

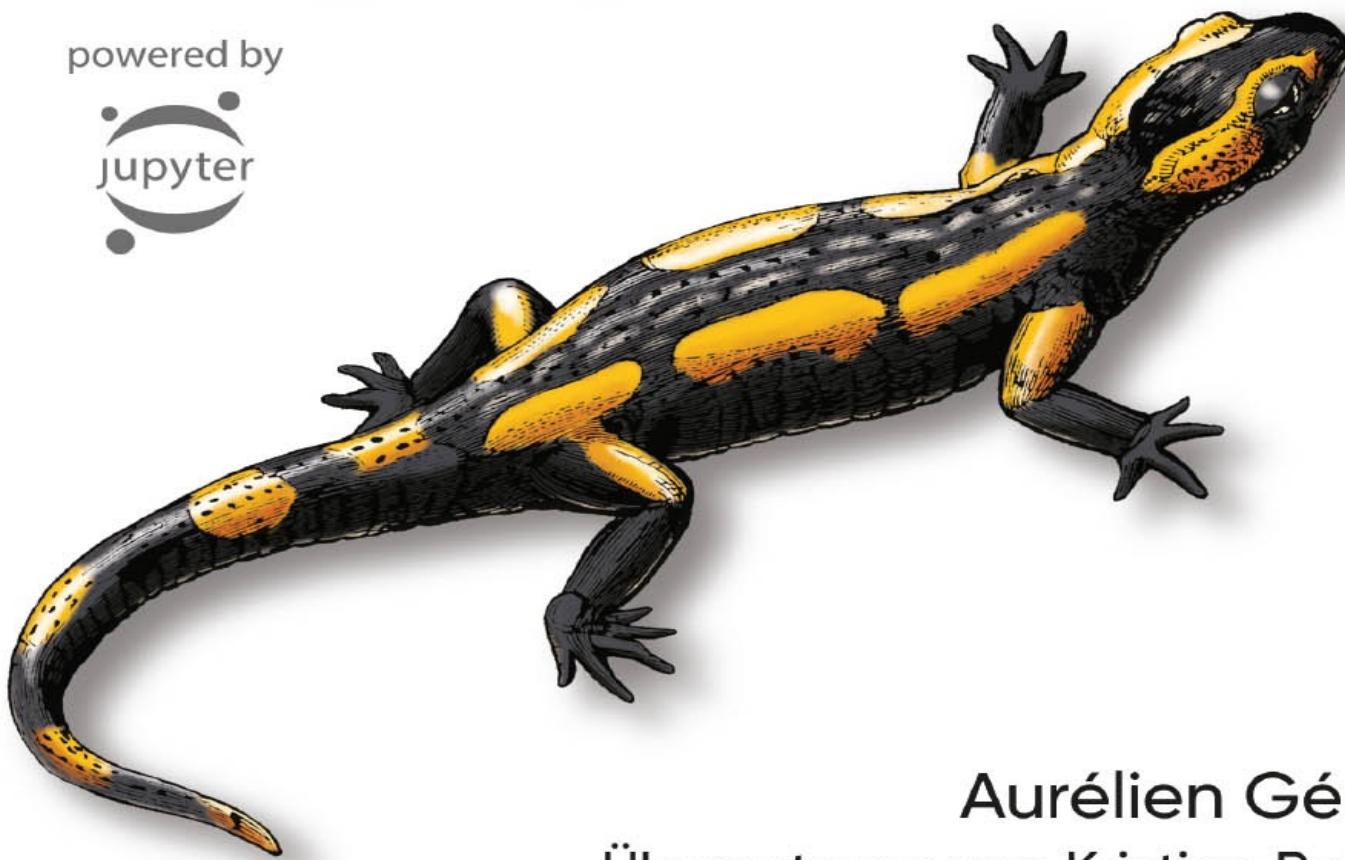
O'REILLY®

2. Auflage
Aktuell zu
TensorFlow 2

Praxiseinstieg Machine Learning mit Scikit-Learn, Keras und TensorFlow

Konzepte, Tools und Techniken
für intelligente Systeme

powered by



Aurélien Géron

Übersetzung von Kristian Rother
und Thomas Demmig

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

2. AUFLAGE

Praxiseinstieg Machine Learning mit Scikit-Learn, Keras und TensorFlow

*Konzepte, Tools und Techniken
für intelligente Systeme*

Aurélien Géron

*Deutsche Übersetzung von
Kristian Rother & Thomas Demmig*

O'REILLY®

Aurélien Géron

Lektorat: Alexandra Follenius

Übersetzung: Kristian Rother, Thomas Demmig

Korrektorat: Sibylle Feldmann, www.richtiger-text.de

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

Herstellung: Stefanie Weidner

Umschlaggestaltung: Karen Montgomery, Michael Oréal, www.oreal.de

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 978-3-96009-124-0

PDF 978-3-96010-339-4

ePub 978-3-96010-340-0

mobi 978-3-96010-341-7

2. Auflage

Translation Copyright für die deutschsprachige Ausgabe © 2020 dpunkt.verlag GmbH

Wieblinger Weg 17

69123 Heidelberg

Authorized German translation of the English edition of *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*, ISBN 9781492032649 © 2019 Kiwisoft S.A.S. 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.

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.



Hinweis:

Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:

Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: kommentar@oreilly.de.

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.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag noch Übersetzer können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Inhalt

Vorwort

Teil I Die Grundlagen des Machine Learning

1 Die Machine-Learning-Umgebung

- Was ist Machine Learning?
- Warum wird Machine Learning verwendet?
- Anwendungsbeispiel
- Unterschiedliche Machine-Learning-Systeme
 - Überwachtes/unüberwachtes Lernen
 - Batch- und Online-Learning
 - Instanzbasiertes versus modellbasiertes Lernen
- Die wichtigsten Herausforderungen beim Machine Learning
 - Unzureichende Menge an Trainingsdaten
 - Nicht repräsentative Trainingsdaten
 - Minderwertige Daten
 - Irrelevante Merkmale
 - Overfitting der Trainingsdaten
 - Underfitting der Trainingsdaten
 - Zusammenfassung
- Testen und Validieren
 - Hyperparameter anpassen und Modellauswahl
 - Datendiskrepanz
- Übungen

2 Ein Machine-Learning-Projekt von A bis Z

- Der Umgang mit realen Daten
- Betrachte das Gesamtbild
 - Die Aufgabe abstecken
 - Wähle ein Qualitätsmaß aus
 - Überprüfe die Annahmen
- Beschaffe die Daten
 - Erstelle eine Arbeitsumgebung
 - Die Daten herunterladen
 - Wirf einen kurzen Blick auf die Datenstruktur
 - Erstelle einen Testdatensatz

Erkunde und visualisiere die Daten, um Erkenntnisse zu gewinnen
Visualisieren geografischer Daten
Suche nach Korrelationen
Experimentieren mit Kombinationen von Merkmalen
Bereite die Daten für Machine-Learning-Algorithmen vor
Aufbereiten der Daten
Bearbeiten von Text und kategorischen Merkmalen
Eigene Transformer
Skalieren von Merkmalen
Pipelines zur Transformation
Wähle ein Modell aus und trainiere es
Trainieren und Auswerten auf dem Trainingsdatensatz
Bessere Auswertung mittels Kreuzvalidierung
Optimiere das Modell
Gittersuche
Zufällige Suche
Ensemble-Methoden
Analysiere die besten Modelle und ihre Fehler
Evaluiere das System auf dem Testdatensatz
Nimm das System in Betrieb, überwache und warte es
Probieren Sie es aus!
Übungen

3 Klassifikation

MNIST
Trainieren eines binären Klassifikators
Qualitätsmaße
Messen der Genauigkeit über Kreuzvalidierung
Konfusionsmatrix
Relevanz und Sensitivität
Die Wechselbeziehung zwischen Relevanz und Sensitivität
Die ROC-Kurve
Klassifikatoren mit mehreren Kategorien
Fehleranalyse
Klassifikation mit mehreren Labels
Klassifikation mit mehreren Ausgaben
Übungen

4 Trainieren von Modellen

Lineare Regression
Die Normalengleichung
Komplexität der Berechnung
Das Gradientenverfahren
Batch-Gradientenverfahren
Stochastisches Gradientenverfahren

- Mini-Batch-Gradientenverfahren
- Polynomielle Regression
- Lernkurven
- Regularisierte lineare Modelle
 - Ridge-Regression
 - Lasso-Regression
 - Elastic Net
 - Early Stopping
- Logistische Regression
 - Abschätzen von Wahrscheinlichkeiten
 - Trainieren und Kostenfunktion
 - Entscheidungsgrenzen
 - Softmax-Regression
- Übungen

5 Support Vector Machines

- Lineare Klassifikation mit SVMs
 - Soft-Margin-Klassifikation
- Nichtlineare SVM-Klassifikation
 - Polynomieller Kernel
 - Ähnlichkeitsbasierte Merkmale
 - Der gaußsche RBF-Kernel
 - Komplexität der Berechnung
- SVM-Regression
- Hinter den Kulissen
 - Entscheidungsfunktion und Vorhersagen
 - Zielfunktionen beim Trainieren
 - Quadratische Programme
 - Das duale Problem
 - Kernel-SVM
 - Online-SVMs
- Übungen

6 Entscheidungsbäume

- Trainieren und Visualisieren eines Entscheidungsbaums
- Vorhersagen treffen
- Schätzen von Wahrscheinlichkeiten für Kategorien
- Der CART-Trainingsalgorithmus
- Komplexität der Berechnung
- Gini-Unreinheit oder Entropie?
- Hyperparameter zur Regularisierung
- Regression
- Instabilität
- Übungen

7 Ensemble Learning und Random Forests

Abstimmverfahren unter Klassifikatoren

Bagging und Pasting

 Bagging und Pasting in Scikit-Learn

 Out-of-Bag-Evaluation

Zufällige Patches und Subräume

Random Forests

 Extra-Trees

 Wichtigkeit von Merkmalen

Boosting

 AdaBoost

 Gradient Boosting

Stacking

Übungen

8 Dimensionsreduktion

Der Fluch der Dimensionalität

Die wichtigsten Ansätze zur Dimensionsreduktion

 Projektion

 Manifold Learning

Hauptkomponentenzerlegung (PCA)

 Erhalten der Varianz

 Hauptkomponenten

 Die Projektion auf d Dimensionen

 Verwenden von Scikit-Learn

 Der Anteil erklärter Varianz

 Auswählen der richtigen Anzahl Dimensionen

 PCA als Komprimierungsverfahren

 Randomisierte PCA

 Inkrementelle PCA

Kernel-PCA

 Auswahl eines Kernels und Optimierung der Hyperparameter

LLE

Weitere Techniken zur Dimensionsreduktion

Übungen

9 Techniken des unüberwachten Lernens

Clustering

 K-Means

 Grenzen von K-Means

 Bildsegmentierung per Clustering

 Vorverarbeitung per Clustering

 Clustering für teilüberwachtes Lernen einsetzen

DBSCAN

 Andere Clustering-Algorithmen

Gaußsche Mischverteilung

Anomalieerkennung mit gaußschen Mischverteilungsmodellen

Die Anzahl an Clustern auswählen

Bayessche gaußsche Mischverteilungsmodelle

Andere Algorithmen zur Anomalie- und Novelty-Erkennung

Übungen

Teil II Neuronale Netze und Deep Learning

10 Einführung in künstliche neuronale Netze mit Keras

Von biologischen zu künstlichen Neuronen

Biologische Neuronen

Logische Berechnungen mit Neuronen

Das Perzeptron

Mehrschichtiges Perzeptron und Backpropagation

Regressions-MLPs

Klassifikations-MLPs

MLPs mit Keras implementieren

TensorFlow 2 installieren

Einen Bildklassifikator mit der Sequential API erstellen

Ein Regressions-MLP mit der Sequential API erstellen

Komplexe Modelle mit der Functional API bauen

Dynamische Modelle mit der Subclassing API bauen

Ein Modell sichern und wiederherstellen

Callbacks

TensorBoard zur Visualisierung verwenden

Feinabstimmung der Hyperparameter eines neuronalen Netzes

Anzahl verborgener Schichten

Anzahl Neuronen pro verborgene Schicht

Lernrate, Batchgröße und andere Hyperparameter

Übungen

11 Trainieren von Deep-Learning-Netzen

Das Problem schwindender/explodierender Gradienten

Initialisierung nach Glorot und He

Nicht sättigende Aktivierungsfunktionen

Batchnormalisierung

Gradient Clipping

Wiederverwenden vortrainierter Schichten

Transfer Learning mit Keras

Unüberwachtes Vortrainieren

Vortrainieren anhand einer Hilfsaufgabe

Schnellere Optimierer

Momentum Optimization

- Beschleunigter Gradient nach Nesterov
- AdaGrad
- RMSProp
- Adam-Optimierung
- Scheduling der Lernrate
- Vermeiden von Overfitting durch Regularisierung
 - ℓ_1 - und ℓ_2 -Regularisierung
 - Drop-out
 - Monte-Carlo-(MC)-Drop-out
 - Max-Norm-Regularisierung
- Zusammenfassung und praktische Tipps
- Übungen

12 Eigene Modelle und Training mit TensorFlow

- Ein kurzer Überblick über TensorFlow
- TensorFlow wie NumPy einsetzen
 - Tensoren und Operationen
 - Tensoren und NumPy
 - Typumwandlung
 - Variablen
 - Andere Datenstrukturen
- Modelle und Trainingsalgorithmen anpassen
 - Eigene Verlustfunktion
 - Modelle mit eigenen Komponenten sichern und laden
 - Eigene Aktivierungsfunktionen, Initialisierer, Regularisierer und Constraints
 - Eigene Metriken
 - Eigene Schichten
 - Eigene Modelle
 - Verlustfunktionen und Metriken auf Modell-Interna basieren lassen
 - Gradienten per Autodiff berechnen
 - Eigene Trainingsschleifen
- Funktionen und Graphen in TensorFlow
 - AutoGraph und Tracing
 - Regeln für TF Functions
- Übungen

13 Daten mit TensorFlow laden und vorverarbeiten

- Die Data-API
 - Transformationen verketten
 - Daten durchmischen
 - Daten vorverarbeiten
 - Alles zusammenbringen
 - Prefetching
 - Datasets mit tf.keras verwenden
- Das TFRecord-Format

- Komprimierte TFRecord-Dateien
- Eine kurze Einführung in Protocol Buffer
- TensorFlow-Protobufs
- Examples laden und parsen
- Listen von Listen mit dem SequenceExample-Protobuf verarbeiten
- Die Eingabemerkmale vorverarbeiten
 - Kategorische Merkmale mit One-Hot-Vektoren codieren
 - Kategorische Merkmale mit Embeddings codieren
 - Vorverarbeitungsschichten von Keras
- TF Transform
- Das TensorFlow-Datasets-(TFDS-)Projekt
- Übungen

14 Deep Computer Vision mit Convolutional Neural Networks

- Der Aufbau des visuellen Cortex
- Convolutional Layers
 - Filter
 - Stapeln mehrerer Feature Maps
 - Implementierung in TensorFlow
 - Speicherbedarf
- Pooling Layers
 - Implementierung in TensorFlow
- Architekturen von CNNs
 - LeNet-5
 - AlexNet
 - GoogLeNet
 - VGGNet
 - ResNet
 - Xception
 - SENet
- Ein ResNet-34-CNN mit Keras implementieren
- Vortrainierte Modelle aus Keras einsetzen
- Vortrainierte Modelle für das Transfer Learning
- Klassifikation und Lokalisierung
- Objekterkennung
 - Fully Convolutional Networks
 - You Only Look Once (YOLO)
- Semantische Segmentierung
- Übungen

15 Verarbeiten von Sequenzen mit RNNs und CNNs

- Rekurrente Neuronen und Schichten
 - Gedächtniszellen
 - Ein- und Ausgabesequenzen
- RNNs trainieren

- Eine Zeitserie vorhersagen
 - Grundlegende Metriken
 - Ein einfaches RNN implementieren
 - Deep RNNs
 - Mehrere Zeitschritte vorhersagen
- Arbeit mit langen Sequenzen
 - Gegen instabile Gradienten kämpfen
 - Das Problem des Kurzzeitgedächtnisses
- Übungen

16 Natürliche Sprachverarbeitung mit RNNs und Attention

- Shakespearesche Texte mit einem Character-RNN erzeugen
 - Den Trainingsdatensatz erstellen
 - Wie ein sequenzieller Datensatz aufgeteilt wird
 - Den sequenziellen Datensatz in mehrere Fenster unterteilen
 - Das Char-RNN-Modell bauen und trainieren
 - Das Char-RNN-Modell verwenden
 - Einen gefälschten Shakespeare-Text erzeugen
 - Zustandsbehaftetes RNN
- Sentimentanalyse
 - Maskieren
 - Vortrainierte Embeddings wiederverwenden
- Ein Encoder-Decoder-Netzwerk für die neuronale maschinelle Übersetzung
 - Bidirektionale RNNs
 - Beam Search
- Attention-Mechanismen
 - Visuelle Attention
 - Attention Is All You Need: Die Transformer-Architektur
- Aktuelle Entwicklungen bei Sprachmodellen
- Übungen

17 Representation Learning und Generative Learning mit Autoencodern und GANs

- Effiziente Repräsentation von Daten
- Hauptkomponentenzerlegung mit einem untvollständigen linearen Autoencoder
- Stacked Autoencoder
 - Einen Stacked Autoencoder mit Keras implementieren
 - Visualisieren der Rekonstruktionen
 - Den Fashion-MNIST-Datensatz visualisieren
 - Unüberwachtes Vortrainieren mit Stacked Autoencoder
 - Kopplung von Gewichten
 - Trainieren mehrerer Autoencoder nacheinander
- Convolutional Autoencoder
- Rekurrente Autoencoder
- Denoising Autoencoder

Sparse Autoencoder
Variational Autoencoder
 Fashion-MNIST-Bilder erzeugen
Generative Adversarial Networks
 Schwierigkeiten beim Trainieren von GANs
 Deep Convolutional GANs
 Progressive wachsende GANs
 StyleGANs
Übungen

18 Reinforcement Learning

Lernen zum Optimieren von Belohnungen
Suche nach Policies
Einführung in OpenAI Gym
Neuronale Netze als Policies
Auswerten von Aktionen: Das Credit-Assignment-Problem
Policy-Gradienten
Markov-Entscheidungsprozesse
Temporal Difference Learning
Q-Learning
 Erkundungspolicies
 Approximatives Q-Learning und Deep-Q-Learning
Deep-Q-Learning implementieren
Deep-Q-Learning-Varianten
 Feste Q-Wert-Ziele
 Double DQN
 Priorisiertes Experience Replay
 Dueling DQN
Die TF-Agents-Bibliothek
 TF-Agents installieren
 TF-Agents-Umgebungen
 Umgebungsspezifikationen
 Umgebungswrapper und Atari-Vorverarbeitung
 Trainingsarchitektur
 Deep-Q-Netz erstellen
 DQN-Agenten erstellen
 Replay Buffer und Beobachter erstellen
 Trainingsmetriken erstellen
 Collect-Fahrer erstellen
 Dataset erstellen
 Trainingsschleife erstellen
Überblick über beliebte RL-Algorithmen
Übungen

19 TensorFlow-Modelle skalierbar trainieren und deployen

Ein TensorFlow-Modell ausführen
TensorFlow Serving verwenden
Einen Vorhersageservice auf der GCP AI Platform erstellen
Den Vorhersageservice verwenden
Ein Modell auf ein Mobile oder Embedded Device deployen
Mit GPUs die Berechnungen beschleunigen
Sich eine eigene GPU zulegen
Eine mit GPU ausgestattete virtuelle Maschine einsetzen
Colaboratory
Das GPU-RAM verwalten
Operationen und Variablen auf Devices verteilen
Paralleles Ausführen auf mehreren Devices
Modelle auf mehreren Devices trainieren
Parallelisierte Modelle
Parallelisierte Daten
Mit der Distribution Strategies API auf mehreren Devices trainieren
Ein Modell in einem TensorFlow-Cluster trainieren
Große Trainingsjobs auf der Google Cloud AI Platform ausführen
Black Box Hyperparameter Tuning auf der AI Platform
Übungen
Vielen Dank!

- A Lösungen zu den Übungsaufgaben**
 - B Checkliste für Machine-Learning-Projekte**
 - C Das duale Problem bei SVMs**
 - D Autodiff**
 - E Weitere verbreitete Architekturen neuronaler Netze**
 - F Spezielle Datenstrukturen**
 - G TensorFlow-Graphen**
- Index**

Vorwort

Der Machine-Learning-Tsunami

Im Jahr 2006 erschien ein Artikel (<https://homl.info/136>) 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. Ein Deep Neural Network ist ein (sehr) vereinfachtes Modell unseres zerebralen Kortex, und es besteht aus einer Folge von Schichten mit künstlichen Neuronen. 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 auflieben, 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 dem 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!

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 an 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.

- Betrügerische Transaktionen mit hoher Wahrscheinlichkeit erkennen.
- Den Unternehmensgewinn im nächsten Jahr vorhersagen.

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 von 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. Mit ihr können Sie sehr große neuronale Netze effizient trainieren und ausführen, indem Sie die Berechnungen auf bis zu Hunderte von Servern mit mehreren GPUs (*Graphics Processing Units*) verlagern. TensorFlow (TF) wurde von Google entwickelt und läuft in vielen großflächigen Machine-Learning-Anwendungen. Die Bibliothek wurde im November 2015 als Open Source veröffentlicht.
- Keras (<https://keras.io/>) ist eine High-Level-Deep-Learning-API, die das Trainieren und Ausführen neuronaler Netze sehr einfach macht. Sie kann auf TensorFlow, Theano oder Microsoft Cognitive Toolkit (früher bekannt als CNTK) aufsetzen. TensorFlow bringt seine eigene Implementierung dieser API namens *tf.keras* mit, die einige der fortgeschrittenen TensorFlow-Features unterstützt (zum Beispiel die Möglichkeit, Daten effizient zu laden).

Dieses Buch verfolgt einen praxisorientierten Ansatz, bei dem Sie ein intuitives Verständnis von Machine Learning entwickeln, indem Sie sich mit konkreten Beispielen 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-ml2> 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 dafür interessieren, was hinter den Kulissen passiert, sollten Sie ein Grundverständnis von Oberstufenmathematik haben (Analysis, lineare Algebra, Wahrscheinlichkeiten und Statistik).

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

Falls Sie Jupyter noch nie verwendet haben, führt Sie [Kapitel 2](#) durch die Installation und die Grundlagen: Es ist ein leistungsfähiges Werkzeug in Ihrem Werkzeugkasten.

Und für den Fall, dass 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 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 Herausforderungen beim Machine Learning, insbesondere Underfitting und Overfitting (das Gleichgewicht zwischen Bias und Varianz).
- Die verbreitetsten Lernalgorithmen: lineare und polynomische Regression, logistische Regression, k-nächste Nachbarn, Support Vector Machines, Entscheidungsbäume, Random Forests und Ensemble-Methoden.
- Dimensionsreduktion der Trainingsdaten, um dem »Fluch der Dimensionalität« etwas entgegenzusetzen.
- Andere Techniken des unüberwachten Lernens, unter anderem Clustering, Dichteabschätzung und Anomalieerkennung.

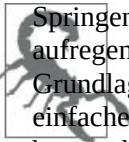
[Teil II](#) widmet sich diesen Themen:

- Was sind neuronale Netze? Wofür sind sie geeignet?
- Erstellen und Trainieren neuronaler Netze mit TensorFlow und Keras.
- Die wichtigsten Architekturen neuronaler Netze: Feed-Forward-Netze für Tabellendaten, Convolutional Neural Networks zur Bilderkennung, rekurrente Netze und Long-Short-Term-Memory-(LSTM-)Netze zur Sequenzverarbeitung, Encoder/Decoder und Transformer für die Sprachverarbeitung, Autoencoder und Generative Adversarial

Networks (GANs) zum generativen Lernen.

- Techniken zum Trainieren von Deep-Learning-Netzen.
- Wie man einen Agenten erstellt (zum Beispiel einen Bot in einem Spiel), der durch Versuch und Irrtum gute Strategien erlernt und dabei Reinforcement Learning einsetzt.
- Effizientes Laden und Vorverarbeiten großer Datenmengen.
- Trainieren und Deployen von TensorFlow-Modellen im großen Maßstab.

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



Springen Sie nicht zu schnell ins tiefe Wasser: Auch wenn Deep Learning zweifelsohne eines der aufregendsten Teilgebiete des 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.

Änderungen in der zweiten Auflage

Diese zweite Auflage hat sechs zentrale Ziele:

1. Die Behandlung zusätzlicher ML-Themen: weitere Techniken zum unüberwachten Lernen (unter anderem Clustering, Anomalieerkennung, Dichteabschätzung und Mischmodelle), zusätzliche Techniken zum Trainieren von Deep Networks (einschließlich sich selbst normalisierender Netze), weitere Techniken der Bilderkennung (unter anderem Xception, SENet, Objekterkennung mit YOLO und semantische Segmentierung mit R-CNN), das Verarbeiten von Sequenzen mit Convolutional Neural Networks (CNNs, einschließlich WaveNet), Verarbeitung natürlicher Sprache mit rekurrenten neuronalen Netzen (RNNs), CNNs und Transformers, GANs.
2. Zusätzliche Bibliotheken und APIs (Keras, die Data-API, TF-Agents für das Reinforcement Learning) sowie das Trainieren und Deployen von TF-Modellen im großen Maßstab mithilfe der Distribution Strategies API, TF Serving und Google Cloud AI Platform; zudem eine kurze Vorstellung von TF Transform, TFLite, TF Addons/Seq2Seq und TensorFlow.js.
3. Vorstellen einiger der neuesten Forschungsergebnisse aus dem Deep Learning.
4. Migrieren aller TensorFlow-Kapitel nach TensorFlow 2 und Verwenden der TensorFlow-Implementierung der Keras-API (`tf.keras`), wann immer das möglich ist.
5. Aktualisieren der Codebeispiele auf die neuesten Versionen von Scikit-Learn, NumPy, pandas, Matplotlib und anderer Bibliotheken.
6. Anpassen einiger Abschnitte zum besseren Verständnis und Beheben von Fehlern dank sehr vieler Rückmeldungen von Lesern.

Es wurden ein paar Kapitel hinzugefügt, andere wurden umgeschrieben, und ein paar wurden neu angeordnet. Unter <https://homl.info/changes2> finden Sie detailliertere Angaben darüber, was sich in der zweiten Auflage geändert hat.

Ressourcen im Netz

Es gibt viele ausgezeichnete Ressourcen, mit deren Hilfe sich Machine Learning erlernen lässt. Der ML-Kurs auf Coursera (<https://homl.info/ngcourse>) von Andrew Ng ist faszinierend, auch wenn er einen beträchtlichen Zeitaufwand bedeutet (in Monaten).

Darüber hinaus finden Sie viele interessante Webseiten über Machine Learning, darunter natürlich den ausgezeichneten User Guide (<https://homl.info/skdoc>) von Scikit-Learn. Auch Dataquest (<https://www.dataquest.io/>), das sehr ansprechende Tutorials und ML-Blogs bietet, sowie die auf Quora (<https://homl.info/1>) aufgeführten ML-Blogs könnten Ihnen gefallen. Schließlich sind auf der Deep-Learning-Website (<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* (<https://www.oreilly.de/buecher/13335/9783960091233-einf%C3%BChrung-in-data-science.html>) (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 & 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 und Scikit-Learn und TensorFlow: Das umfassende 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).
- François Chollet, *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek* (mitp Professional). Ein sehr praxisnahes Buch, das klar und präzise viele Themen behandelt – wie Sie es vom Autor der ausgezeichneten Keras-Bibliothek erwarten können. Es zieht Codebeispiele der mathematischen Theorie vor.
- Andriy Burkov, *Machine Learning kompakt: Alles, was Sie wissen müssen* (mitp Professional). Dieses sehr kurze Buch behandelt ein beeindruckendes Themenspektrum gut verständlich, scheut dabei aber nicht vor mathematischen Gleichungen zurück.
- Yaser S. Abu-Mostafa, Malik Magdon-Ismail und Hsuan-Tien Lin, *Learning from Data* (MLBook). Als eher theoretische Abhandlung von ML enthält dieses Buch sehr tiefgehende Erkenntnisse, insbesondere zum Gleichgewicht zwischen Bias und Varianz (siehe [Kapitel 4](#)).
- Stuart Russell und 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) (<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.

Codebeispiele

Es gibt eine Reihe von Jupyter-Notebooks voll mit zusätzlichem Material, wie Codebeispielen und Übungen, die zum Herunterladen unter <https://github.com/ageron/handson-ml2> bereitstehen.

Einige der Codebeispiele im Buch lassen sich wiederholende Abschnitte oder Details weg, die offensichtlich sind oder nichts mit Machine Learning zu tun haben. Das sorgt dafür, dass Sie sich auf die wichtigen Teile des Codes konzentrieren können, und spart Platz, um mehr Themen unterzubringen. Wollen Sie sich die vollständigen Codebeispiele betrachten, finden Sie diese in den Jupyter-Notebooks.

Geben die Codebeispiele etwas aus, wird dies mit Python-Prompts (`>>>` und `...`) wie in einer Python-Shell dargestellt, um den Code deutlich von den Ausgaben trennen zu können. So

definiert beispielsweise folgender Code die Funktion `square()`, rechnet dann damit und gibt das Quadrat von 3 aus:

```
>>> def square(x):
...     return x ** 2
...
>>> result = square(3)
>>> result
9
```

Gibt Code nichts aus, werden keine Prompts verwendet. Aber das Ergebnis wird manchmal als Kommentar angegeben, wie zum Beispiel hier:

```
def square(x):
    return x ** 2

result = square(3) # Ergebnis ist 9
```

Verwenden von Codebeispielen

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, Keras und TensorFlow* von Aurélien Géron (O'Reilly). Copyright 2020, ISBN 978-3-96009-124-0.«

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

In meinen wildesten Träumen hätte ich mir niemals vorgestellt, dass die erste Auflage dieses Buchs solch eine Verbreitung finden würde. Ich habe so viele Nachrichten von Lesern erhalten – oft mit Fragen, manche mit freundlichen Hinweisen auf Fehler und die meisten mit ermutigenden Worten. Ich kann gar nicht sagen, wie dankbar ich all diesen Lesern für ihre Unterstützung bin. Vielen, vielen Dank! Scheuen Sie sich nicht, sich auf GitHub (<https://homl.info/issues2>) zu melden, wenn Sie Fehler in den Codebeispielen finden (oder einfach etwas fragen wollen) oder um auf Fehler im Text aufmerksam (<https://homl.info/errata2>) zu machen. Manche Leser haben mir auch geschrieben, wie dieses Buch ihnen dabei geholfen hat, ihren ersten Job zu bekommen oder ein konkretes Problem zu lösen, an dem sie gearbeitet haben. Ich finde ein solches Feedback unglaublich motivierend. Hat Ihnen dieses Buch geholfen, würde ich mich freuen, wenn Sie mir Ihre Geschichte erzählen würden – entweder privat (zum Beispiel über LinkedIn (<https://www.linkedin.com/in/aurelien-geron/>)) oder öffentlich (beispielsweise in einem Tweet oder über ein Amazon-Review (<https://homl.info/amazon2>)).

Ich bin all den fantastischen Menschen unglaublich dankbar, die in ihrem geschäftigen Leben die Zeit gefunden haben, mein Buch im Detail gegenzulesen. Insbesondere möchte ich François Chollet dafür danken, dass er alle Kapitel zu Keras und TensorFlow kontrolliert und mir großartiges und detailliertes Feedback gegeben hat. Da Keras eine meiner wichtigsten Ergänzungen dieser zweiten Auflage ist, war die Review durch den Autor unbezahlbar. Ich empfehle Ihnen François' Buch *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek* (mitp Professional): Es bietet die Präzision, Klarheit und Tiefe, die auch die Keras-Bibliothek selbst besitzt. Besonderer Dank geht ebenfalls an Ankur Patel, der jedes Kapitel dieser zweiten Auflage begutachtet und mir ausgezeichnetes Feedback gegeben hat, insbesondere zu [Kapitel 9](#), das sich um unüberwachtes Lernen dreht. Er könnte glatt ein ganzes Buch zu dem Thema schreiben ... Moment mal, das hat er ja! Schauen Sie sich mal *Praxisbuch Unsupervised Learning: Machine-Learning-Anwendungen für ungelabelte Daten mit Python programmieren* (<https://www.oreilly.de/buecher/13534/9783960091271-praxisbuch-unsupervised-learning.html>) (O'Reilly) an. Ein großes Dankeschön auch an Olzhass Akpambetov, der alle Kapitel im zweiten Teil des Buchs begutachtet, sehr viel Code getestet und viele großartige Verbesserungsvorschläge gemacht hat. Ich bin dankbar, dass Mark Daoust, Jon Krohn, Dominic Monn und Josh Patterson den zweiten Teil des Buchs so genau begutachtet und ihre Erfahrungen eingebracht haben. Sie ließen keinen Stein auf dem anderen und lieferten ausgezeichnete Hinweise.

Beim Schreiben dieser zweiten Auflage hatte ich das Glück, sehr viel Hilfe von Mitgliedern des TensorFlow-Teams zu bekommen, insbesondere von Martin Wicke, der unermüdlich Dutzende meiner Fragen beantwortet und den Rest an die richtigen Leute weitergeleitet hat, unter anderen an Karmel Allison, Paige Bailey, Eugene Brevdo, William Chargin, Daniel »Wolff« Dobson, Nick Felt, Bruce Fontaine, Goldie Gadde, Sandeep Gupta, Priya Gupta, Kevin Haas, Konstantinos Katsiapis, Viacheslav Kovalevskyi, Allen Lavoie, Clemens Mewald, Dan Moldovan, Sean Morgan, Tom O'Malley, Alexandre Passos, André Susano Pinto, Anthony Platanios, Oscar Ramirez, Anna Revinskaya, Saurabh Saxena, Ryan Sepassi, Jiri Simska, Xiaodan Song, Christina Sorokin, Dustin Tran, Todd Wang, Pete Warden (der auch die erste Auflage begutachtet hat), Edd Wilder-James und Yuefeng Zhou, die alle außerordentlich hilfreich waren.

Ein großer Dank an euch alle und auch an alle anderen Mitglieder des TensorFlow-Teams – nicht nur für eure Hilfe, sondern auch dafür, dass ihr solch eine tolle Bibliothek geschaffen habt. Ein besonderer Dank geht an Irene Giannoumis und Robert Crowe vom TFCX-Team, die die [Kapitel 13](#) und [19](#) im Detail durchgearbeitet haben.

Ich danke auch den fantastischen Menschen bei O'Reilly, insbesondere Nicole Taché für ihr aufschlussreiches, immer freundliches, ermutigendes und hilfreiches Feedback. Eine bessere Lektorin hätte ich mir nicht vorstellen können. Ein großer Dank geht an Michele Cronin, die zu Beginn dieser zweiten Auflage sehr hilfreich (und geduldig) war, und an Kristen Brown, Production Editor für die zweite Auflage, die sie auf allen Schritten begleitet hat (sie hat auch Korrekturen und Aktualisierungen jedes Nachdrucks der ersten Auflage koordiniert). Ich danke Rachel Monaghan und Amanda Kersey für ihr umfassendes Copyediting (der ersten bzw. zweiten Auflage) und Johnny O'Toole, der die Beziehung zu Amazon gemanagt und viele meiner Fragen beantwortet hat. Dank geht an 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 technischen Fragen zu Formatierung, AsciiDoc und LaTeX sowie Nick Adams, Rebecca Demarest, Rachel Head, Judith McConville, Helen Monroe, Karen Montgomery, Rachel Roumeliotis und allen bei O'Reilly, die zu diesem Buch beigetragen haben.

Ich möchte auch meinen früheren 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 die erste Auflage 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 und Rich Washington. Und danke an alle anderen, mit denen ich bei YouTube und in den großartigen Google-Forschungsteams in Mountain View zusammengearbeitet habe. Vielen Dank auch an Martin Andrews, Sam Witteveen und Jason Zaman, dass sie mich in ihre Google-Developer-Experts-Gruppe in Singapur aufgenommen haben – mit freundlicher Unterstützung durch Soonson Kwon – und für all die tollen Diskussionen über Deep Learning und TensorFlow. Jeder, der in Singapur an Deep Learning interessiert ist, sollte auf jeden Fall das Deep Learning Singapore Meetup (<https://homl.info/meetupsg>) besuchen. Jason verdient besonderen Dank für sein TFLite-Know-how, das in [Kapitel 19](#) eingeflossen ist!

Nie werde ich all die netten Leute vergessen, die die erste Auflage dieses Buchs Korrektur gelesen haben, unter anderem David Andrzejewski, Lukas Biewald, Justin Francis, Vincent Guilbeau, Eddy Hung, Karim Matrah, Grégoire Mesnil, Salim Sémaoune, Iain Smears, Michel Tessier, Ingrid von Glehn, Pete Warden und natürlich mein lieber Bruder Sylvain. Ein besonderer Dank geht an Haesun Park, der mir sehr viel ausgezeichnetes Feedback gab und viele Fehler fand, während er die koreanische Übersetzung der ersten Auflage schrieb. Er hat auch die Jupyter-Notebooks ins Koreanische übersetzt, nicht zu vergessen die Dokumentation von TensorFlow. Ich spreche kein Koreanisch, aber in Anbetracht der Qualität seines Feedbacks müssen alle seine Übersetzungen wirklich ausgezeichnet sein. Darüber hinaus hat Haesun freundlicherweise ein paar der Lösungen zu den Übungen in dieser zweiten Auflage beigetragen.

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 haben. Auch danke ich ihnen für ihre unersättliche Neugier: Indem ich einige der schwierigsten Konzepte in diesem Buch meiner Frau und meinen Kindern erklärt habe, konnte ich meine Gedanken ordnen und viele Teile direkt verbessern. Und sie haben mir sogar Kekse und Kaffee vorbeigebracht. Was kann man sich noch mehr wünschen?

TEIL I

Die Grundlagen des Machine Learning

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 etliche weitere 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 Learning zu erforschen, werfen wir einen Blick auf die Landkarte und lernen die wichtigsten Regionen und Orientierungspunkte kennen: überwachtes und unüberwachtes Lernen, Online- und Batch-Learning, 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 in 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

Ihr Spamfilter ist ein maschinelles Lernprogramm, das aus Beispielen für Spam-E-Mails (z.B. vom Nutzer markierten) und gewöhnlichen E-Mails (Nicht-Spam, auch »Ham« genannt) lernt, 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 (siehe 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. Möglicherweise würden Ihnen auch weitere Muster im Namen des Absenders, dem Text und in anderen Teilen der E-Mail 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.

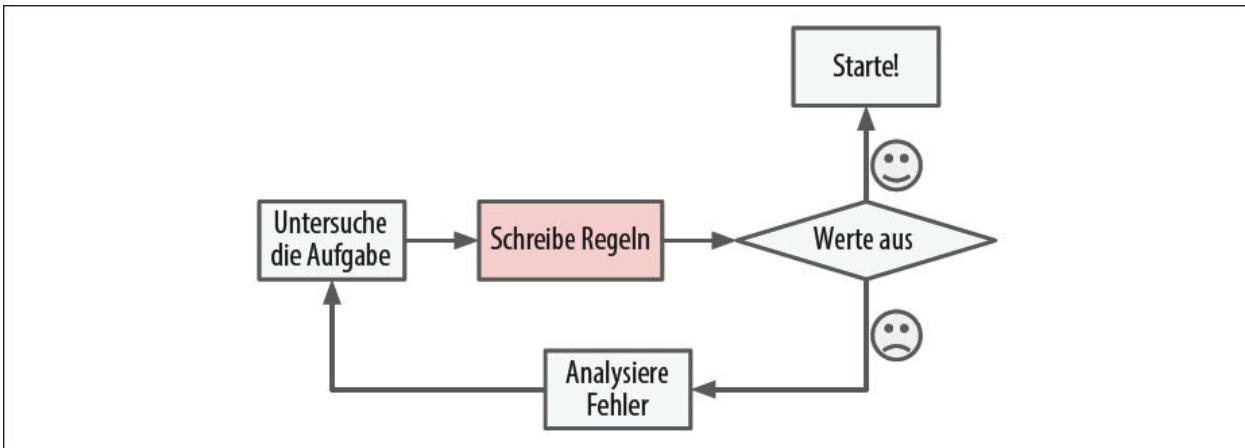


Abbildung 1-1: Die herkömmliche Herangehensweise

Da diese Aufgabe schwierig ist, wird Ihr Programm vermutlich eine lange Liste komplexer Regeln beinhalten – und ganz schön schwer 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 Spambeispielen erkennt (siehe Abbildung 1-2). Das Programm wird viel kürzer, leichter zu warten und wahrscheinlich auch treffsicherer.

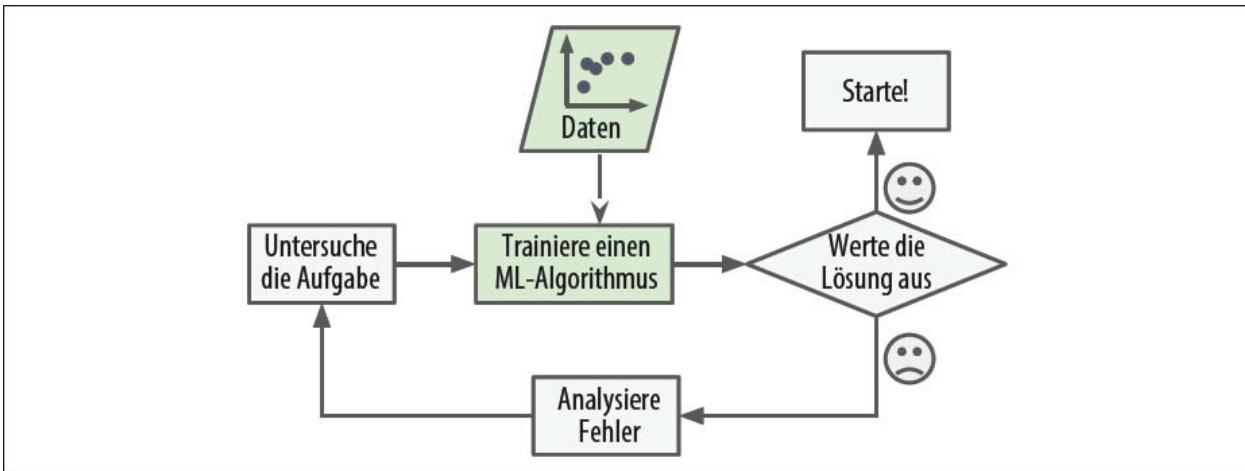


Abbildung 1-2: Der Machine-Learning-Ansatz

Wenn außerdem die Spammer bemerken, dass alle ihre E-Mails mit »Für Sie« geblockt werden, könnten sie stattdessen auf »4U« umsatteln. Ein mit traditionellen Programmietechniken 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 (siehe Abbildung 1-3).

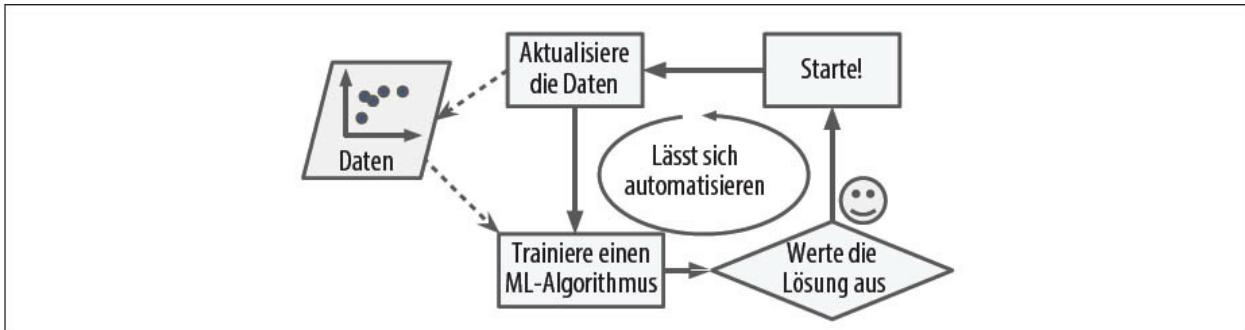


Abbildung 1-3: Automatisches Einstellen auf Änderungen

Machine Learning brilliert außerdem 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, das die Wörter »one« und »two« unterscheiden kann. Sie bemerken, dass das Wort »two« mit einem hochfrequenten Ton (»T«) beginnt, und könnten demnach einen Algorithmus hartcodieren, 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 von 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 unterstützen (siehe 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*.

Zusammengefasst, ist Machine Learning hervorragend geeignet für:

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

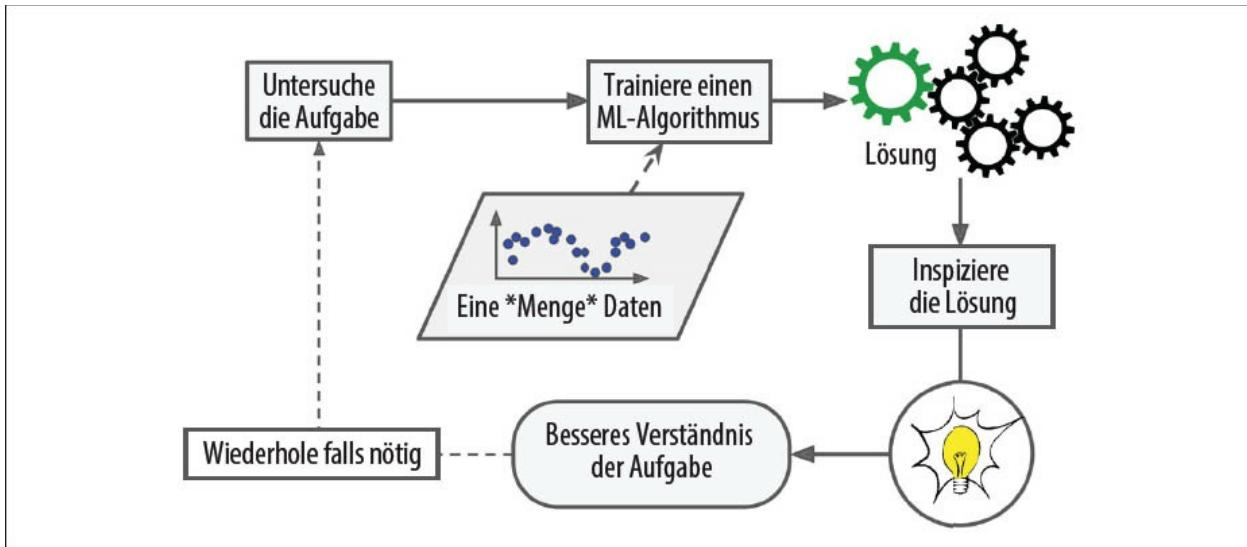


Abbildung 1-4: Machine Learning hilft Menschen beim Lernen.

Anwendungsbeispiel

Schauen wir uns ein paar konkrete Beispiele für Aufgaben des Machine Learning und die dabei eingesetzten Techniken an:

Produktbilder in der Herstellung analysieren, um sie automatisch zu klassifizieren

Dies ist Bildklassifikation, meist ausgeführt durch Convolutional Neural Networks (CNNs, siehe [Kapitel 14](#)).

Tumoren in Gehirnscans erkennen

Das ist eine semantische Segmentierung, bei der jedes Pixel im Bild klassifiziert wird (da wir die genaue Position und Form des Tumors bestimmen wollen), meist ebenfalls mithilfe von CNNs.

Nachrichtenartikel automatisch klassifizieren

Dies ist linguistische Datenverarbeitung (NLP, Natural Language Processing) und, spezifischer, Textklassifikation, die sich über rekurrente neuronale Netze (RNNs), CNNs oder Transformer angehen lässt (siehe [Kapitel 16](#)).

Beleidige Kommentare in Diskussionsforen automatisch markieren

Dabei handelt es sich ebenfalls um Textklassifikation mit den gleichen NLP-Tools.

Automatisch lange Dokumente zusammenfassen

Dies ist ein Zweig der NLP namens Textextrahierung, ebenfalls mit den gleichen Tools.

Einen Chatbot oder persönlichen Assistenten erstellen

Dazu gehören viele NLP-Komponenten, unter anderem das Verstehen natürlicher Sprache (Natural Language Understanding, NLU) und Module zum Beantworten von Fragen.

Den Umsatz Ihrer Firma für das nächste Jahr basierend auf vielen Performancemetriken vorhersagen

Dies ist eine Regressionsaufgabe (also das Vorhersagen von Werten), die über ein

Regressionsmodell wie ein lineares oder polynomisches Regressionsmodell (siehe [Kapitel 4](#)), eine Regressions-SVM (siehe [Kapitel 5](#)), einen Regressions-Random-Forest (siehe [Kapitel 7](#)) oder ein künstliches neuronales Netzwerk (siehe [Kapitel 10](#)) angegangen werden kann. Wollen Sie Zeitreihen vergangener Metriken mit einbeziehen, können Sie RNNs, CNNs oder Transformer nutzen (siehe die [Kapitel 15](#) und [16](#)).

Kreditkartenmissbrauch erkennen

Dies ist Anomalieerkennung (siehe [Kapitel 9](#)).

Kunden anhand ihrer Einkäufe segmentieren, sodass Sie für jeden Bereich unterschiedliche Marketingstrategien entwerfen können

Das ist Clustering (siehe [Kapitel 9](#)).

Einen komplexen, hochdimensionalen Datensatz in einem klaren und verständlichen Diagramm darstellen

Hier geht es um Datenvisualisierung, meist unter Verwendung von Techniken zur Datenreduktion (siehe [Kapitel 8](#)).

Einem Kunden basierend auf dessen bisherigen Käufen ein Produkt empfehlen, das ihn interessieren könnte

Dies ist ein Empfehlungssystem. Ein Ansatz ist, Käufe aus der Vergangenheit (und andere Informationen über den Kunden) in ein künstliches neuronales Netzwerk einzuspeisen (siehe [Kapitel 10](#)) und es dazu zu bringen, den wahrscheinlichsten nächsten Kauf auszugeben. Dieses neuronale Netzwerk würde typischerweise mit bisherigen Abfolgen von Käufen aller Kunden trainiert werden.

Einen intelligenten Bot für ein Spiel bauen

Dies wird oft durch Reinforcement Learning (RL, siehe [Kapitel 18](#)) angegangen. Dabei handelt es sich um einen Zweig des Machine Learning, der Agenten trainiert (zum Beispiel Bots), um die Aktionen auszuwählen, die mit der Zeit in einer gegebenen Umgebung (wie dem Spiel) ihre Belohnungen maximieren (beispielsweise kann ein Bot immer dann eine Belohnung erhalten, wenn der Spieler Lebenspunkte verliert). Das berühmte AlphaGo-Programm, das den Weltmeister im Go geschlagen hat, wurde mithilfe von RL gebaut.

Diese Liste könnte immer weiter fortgeführt werden, aber hoffentlich haben Sie auch so schon einen Eindruck von der unglaublichen Breite und Komplexität der Aufgaben erhalten, die Machine Learning angehen kann, und die Art von Techniken, die Sie dafür verwenden würden.

Unterschiedliche Machine-Learning-Systeme

Es gibt so viele verschiedene 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-Learning gegenüber Batch-Learning).
- Ob sie einfach neue Datenpunkte mit den bereits bekannten Datenpunkten vergleichen

oder stattdessen Muster in den Trainingsdaten erkennen, um ein Vorhersagemodell 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 Onlinelernsystem.

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* (siehe Abbildung 1-5).

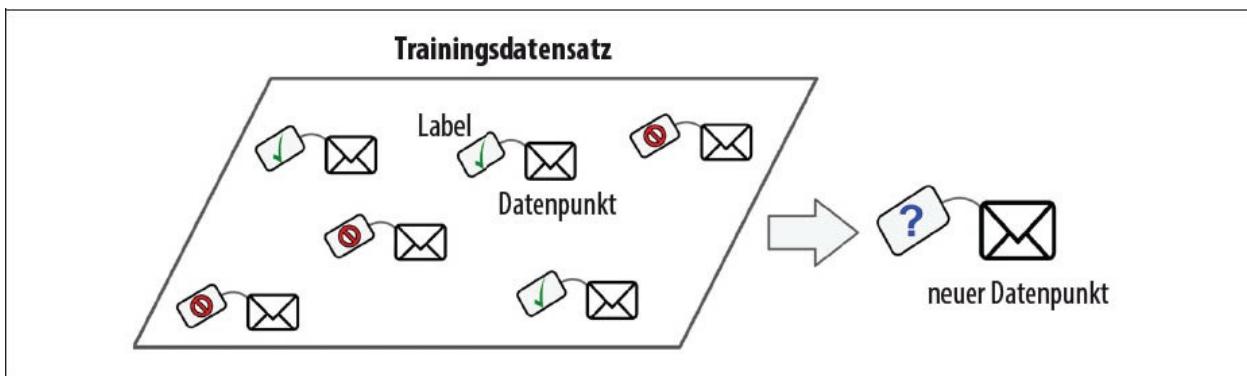


Abbildung 1-5: Ein Trainingsdatensatz mit Labels zur Spamerkennung (ein Beispiel für überwachtes Lernen)

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* (siehe 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.

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

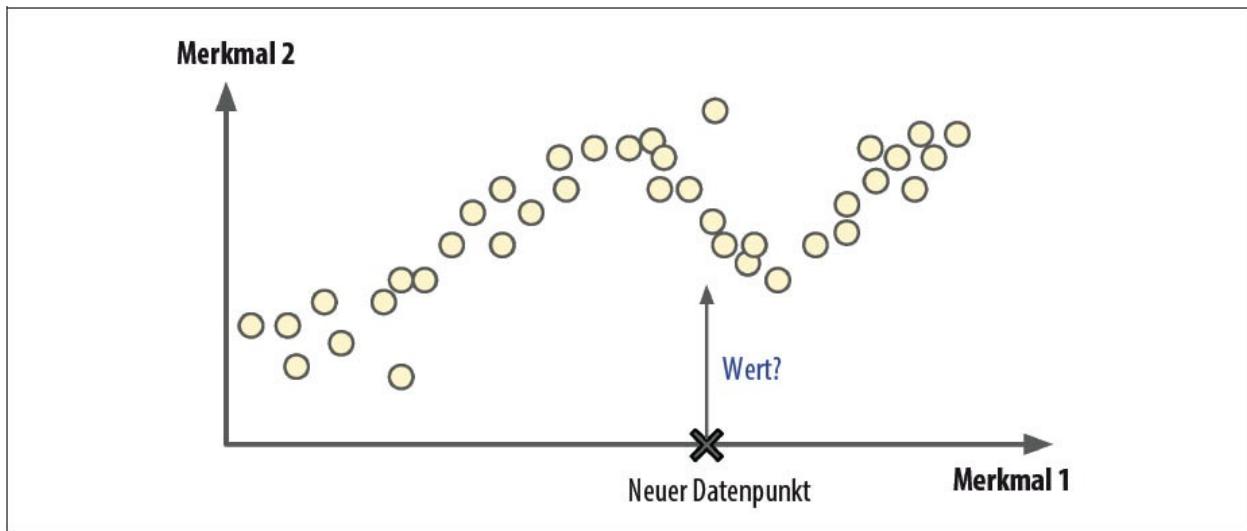


Abbildung 1-6: Ein Regressionsproblem: Sage mithilfe eines Eingangsmerkmals einen Wert voraus (es gibt meist mehrere Eingangsmerkmale und manchmal auch mehrere Ausgangsmerkmale).

Hier sind ein paar 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

Unüberwachtes Lernen

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

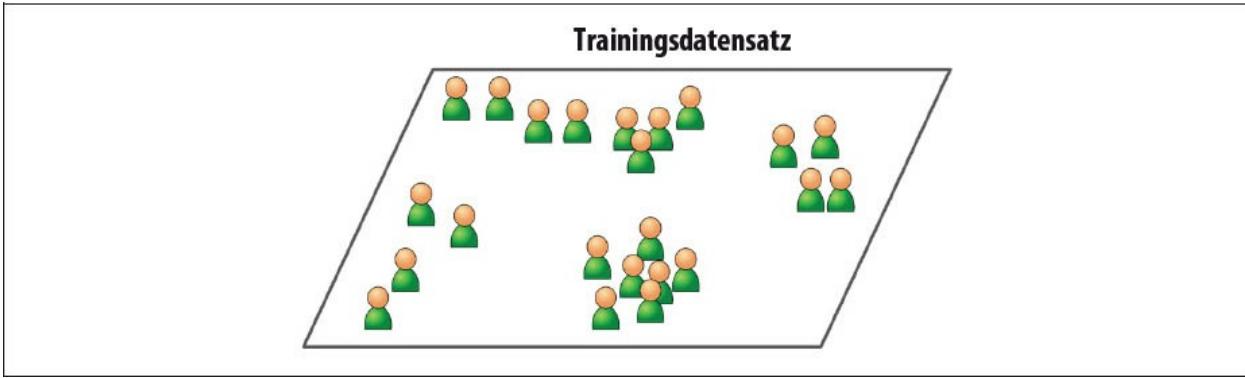


Abbildung 1-7: Ein Trainingsdatensatz ohne Labels für unüberwachtes Lernen

Hier sind ein paar der wichtigsten unüberwachten Lernalgorithmen (wir werden die Dimensionsreduktion in [Kapitel 8](#) und [Kapitel 9](#) behandeln):

- Clustering
 - k-Means
 - DBSCAN
 - hierarchische Cluster-Analyse (HCA)
- Anomalieerkennung und Novelty Detection
 - One-Class-SVM
 - Isolation Forest
- 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 (siehe [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. Wenn Sie ein *hierarchisches Cluster-Verfahren* verwenden, können Sie sogar jede Gruppe in weitere Untergruppen zerlegen, was hilfreich sein kann, wenn Sie Ihre Blogartikel auf diese Zielgruppen zuschneiden möchten.

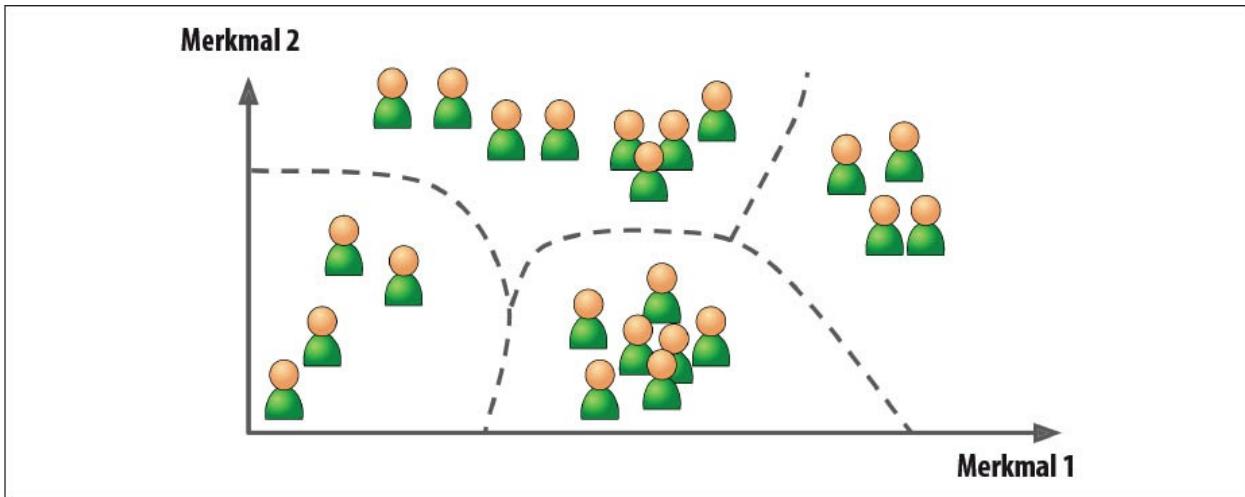


Abbildung 1-8: Clustering

Algorithmen zur *Visualisierung* sind ebenfalls ein gutes Beispiel für unüberwachtes Lernen: Sie übergeben diesen eine Menge komplexer Daten ohne Labels und erhalten eine 2-D- oder 3-D-Repräsentation der Daten, die Sie leicht grafisch darstellen können (siehe [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.

Eine verwandte Aufgabe ist die *Dimensionsreduktion*, bei der das Ziel die Vereinfachung der Daten ist, ohne dabei allzu viele Informationen zu verlieren. Dazu lassen sich mehrere korrelierende Merkmale zu einem vereinigen. Beispielsweise korreliert der Kilometerstand eines Autos stark 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*.

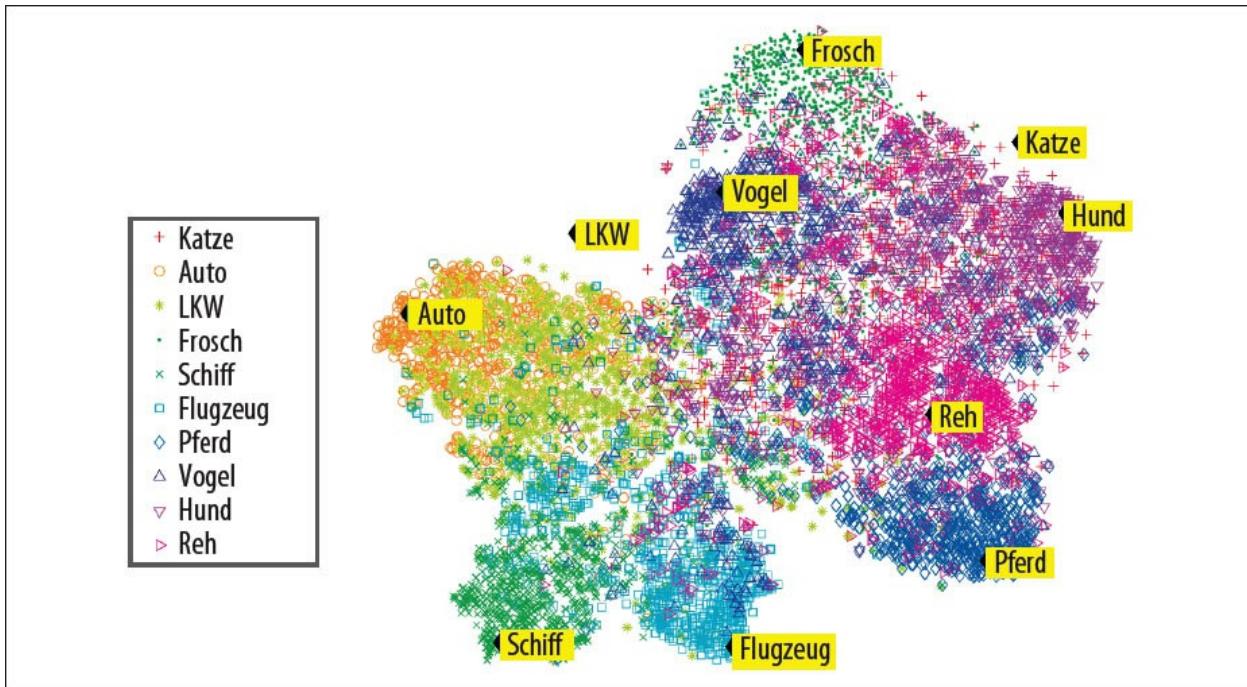


Abbildung 1-9: Beispiel für eine t-SNE-Visualisierung semantischer Cluster³

* 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. Dem System werden beim Training vor allem gewöhnliche Datenpunkte präsentiert, sodass es lernt, sie zu erkennen. Sieht es dann einen neuen Datenpunkt, kann es entscheiden, ob dieser wie ein normaler Punkt oder eine Anomalie aussieht (siehe Abbildung 1-10). Eine sehr ähnliche Aufgabe ist die *Novelty Detection*: Ihr Ziel ist es, neue Instanzen zu erkennen, die anders als alle anderen Instanzen im Trainingsdatensatz aussehen. Dafür brauchen Sie einen sehr »sauber« Trainingsdatensatz, der von allen Instanzen befreit ist, die der Algorithmus erkennen soll. Haben Sie beispielsweise Tausende von Bildern mit Hunden und sind nur auf 1% dieser Bilder Chihuahuas zu sehen, sollte ein Algorithmus zur Novelty Detection neue Bilder von Chihuahuas nicht als Besonderheiten behandeln. Ein Algorithmus zur Anomalieerkennung mag diese Hunde allerdings als so selten ansehen – und als so anders als andere Hunde –, dass er sie als Anomalien klassifizieren würde (nichts gegen Chihuahuas).

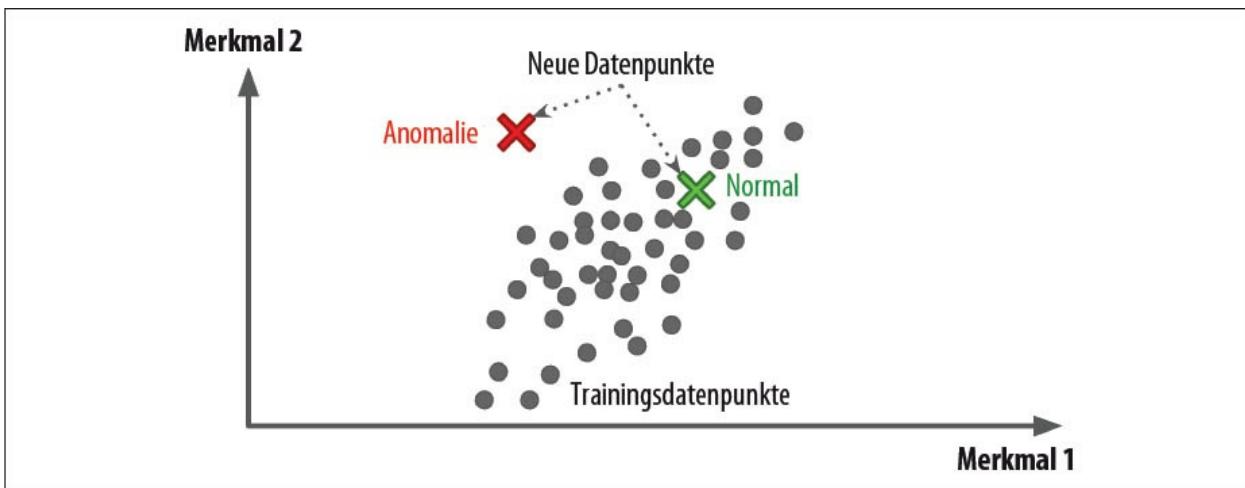


Abbildung 1-10: Erkennen von Anomalien

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

Da das Labeling normalerweise zeitaufwendig und teuer ist, werden Sie oftmals sehr viele ungelabelte und wenige gelabelte Instanzen haben. Einige Algorithmen können mit nur teilweise gelabelten Trainingsdaten arbeiten. Dies bezeichnet man als *halbüberwachtes Lernen* (siehe Abbildung 1-11).

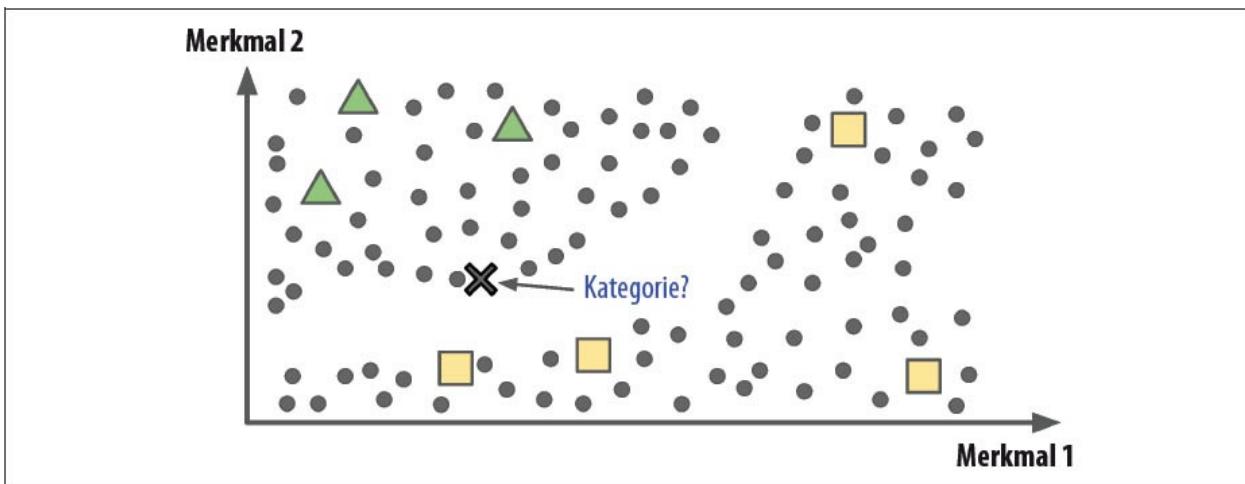


Abbildung 1-11: Halbüberwachtes Lernen mit zwei Klassen (Dreiecke und Quadrate): Die ungelabelten Beispiele (Kreise) helfen dabei, eine neue Instanz (das Kreuz) in die Dreiecksklasse statt in die Quadratkategorie einzurordnen, auch wenn sie näher an den gelabelten Quadraten ist.

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 äußerst nützlich ist.

Die meisten Algorithmen für halbüberwachtes Lernen sind Kombinationen aus 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 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.

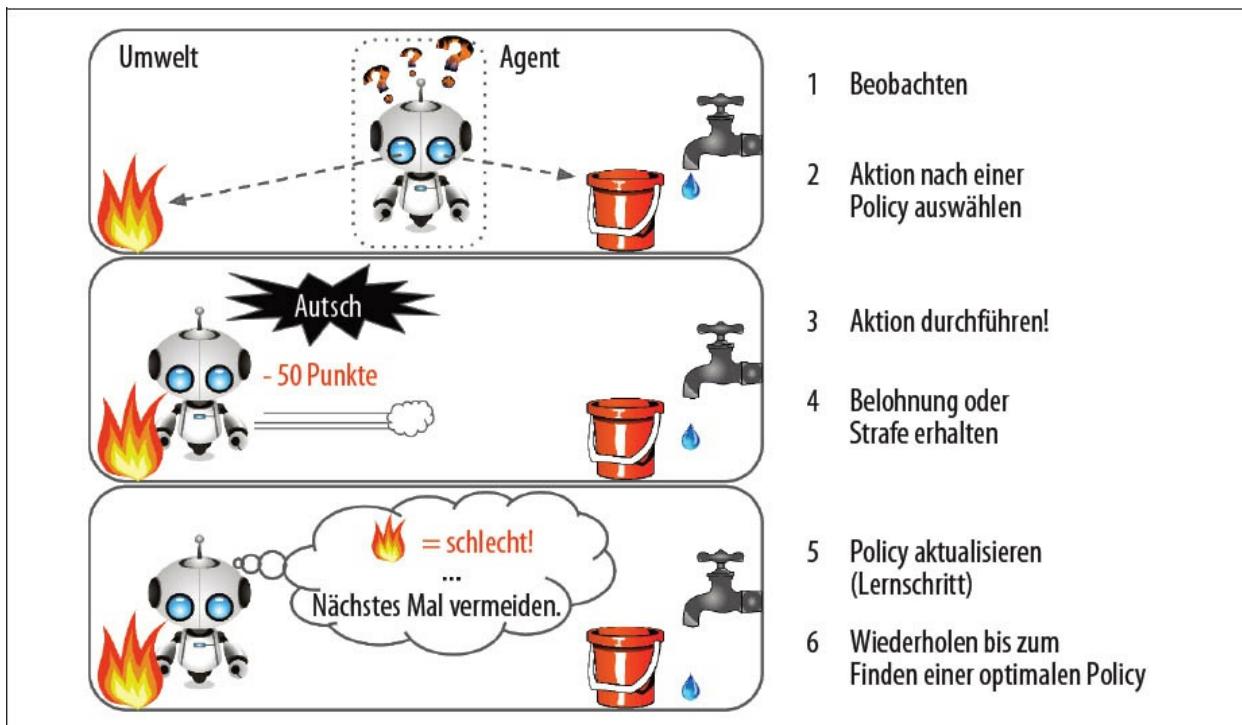


Abbildung 1-12: Reinforcement Learning

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.

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 neuartigen Spam), müssen Sie eine neue Version des Systems ein weiteres Mal 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), ist das Herumschleppen großer Mengen an 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. Aktienkursen), das sich entweder schnell oder autonom an Veränderungen anpassen muss. Auch wenn Ihnen nur 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 (es sei denn, Sie möchten in der Lage sein, zu einem früheren Zustand zurückzukehren und den Datenstrom erneut »abzuspielen«). Dies kann enorme Mengen an Speicherplatz einsparen.

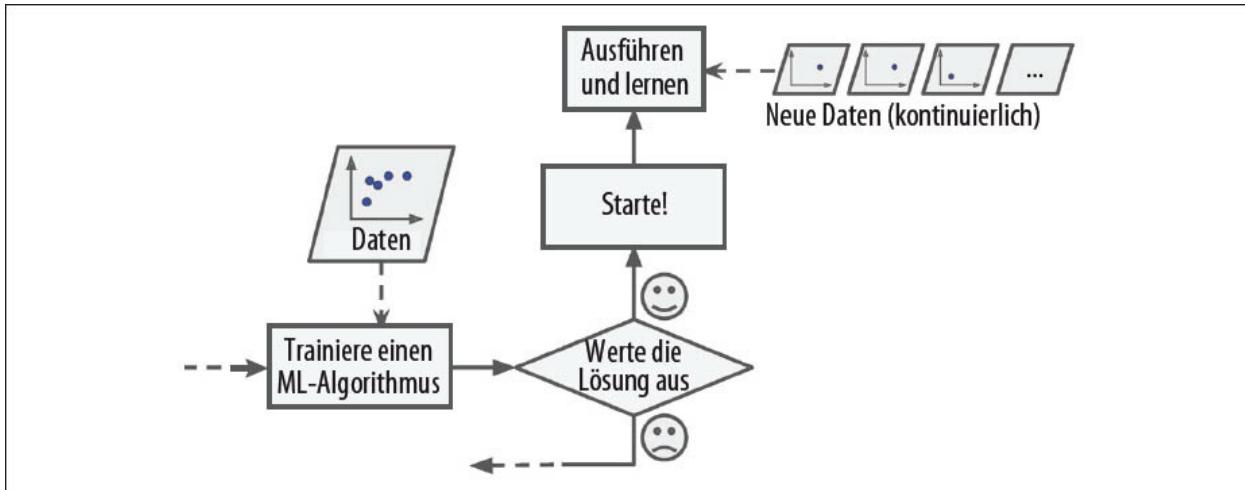


Abbildung 1-13: Beim Online-Learning wird ein Modell trainiert und in den Produktivbetrieb übernommen, wo es mit neu eintreffenden Daten weiterlernt.

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).



Out-of-Core-Lernen 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.

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 sich Ihr System 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 (Outlier).

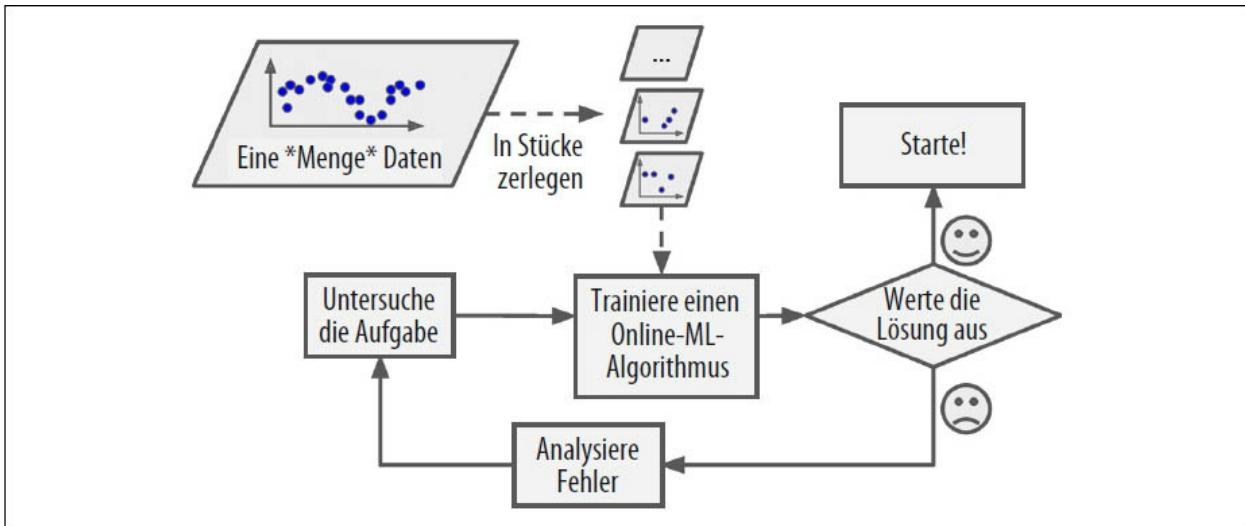


Abbildung 1-14: Verwenden von Online-Learning zum Bewältigen riesiger Datensätze

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 auch von 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 Möglichkeit, maschinelle Lernverfahren zu kategorisieren, 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 erstellten, würde dieser einfach alle E-Mails aussortieren, die mit bereits von Nutzern markierten E-Mails identisch sind – nicht die schlechteste Lösung, aber sicher nicht die beste.

Anstatt einfach mit bekannten Spam-E-Mails identische Nachrichten zu markieren, könnte Ihr

Spamfilter auch so programmiert sein, dass darüber hinaus 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, wobei es sie mit den gelernten Beispielen (oder einer Untermenge davon) vergleicht. So würden beispielsweise in [Abbildung 1-15](#) die neuen Instanzen als Dreieck klassifiziert werden, weil die Mehrheit der ähnlichsten Instanzen zu dieser Klasse gehört.

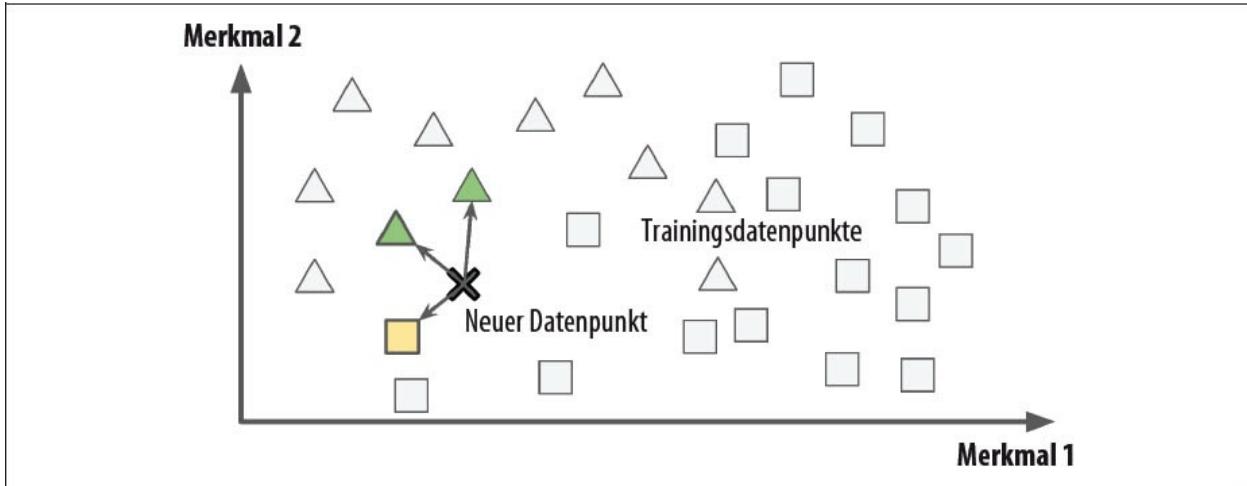


Abbildung 1-15: *Instanzbasiertes Lernen*

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. Das wird als *modellbasiertes Lernen* bezeichnet (siehe [Abbildung 1-16](#)).

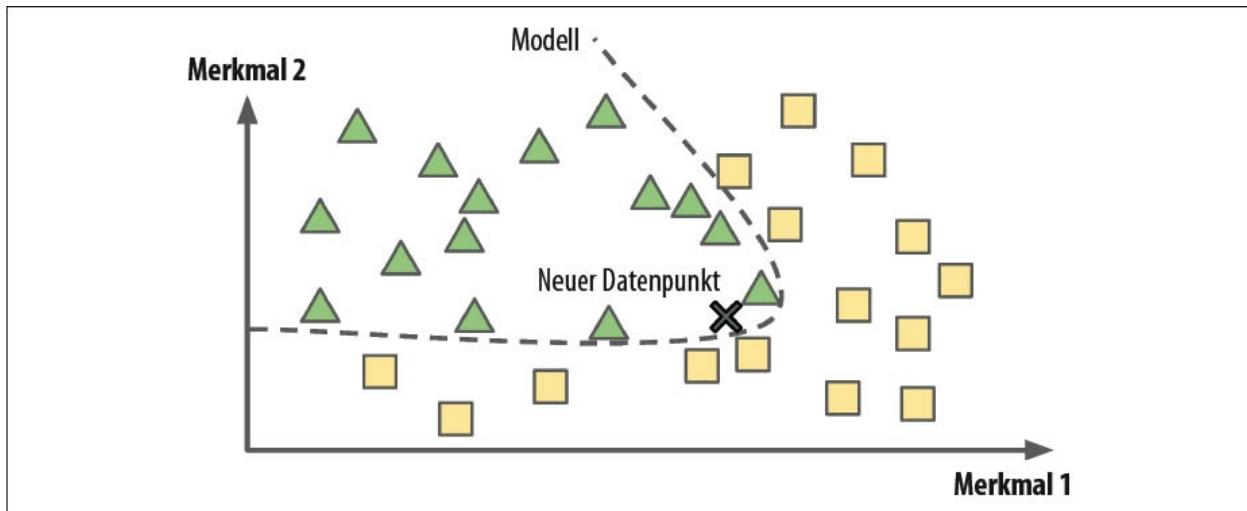


Abbildung 1-16: Modellbasiertes Lernen

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://homl.info/4>) herunter und Statistiken zum Pro-Kopf-Bruttoinlandsprodukt (BIP) von der Webseite des IMF (<https://homl.info/5>). Anschließend führen Sie beide Tabellen 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 die Daten für einige dieser Länder dar (siehe Abbildung 1-17).

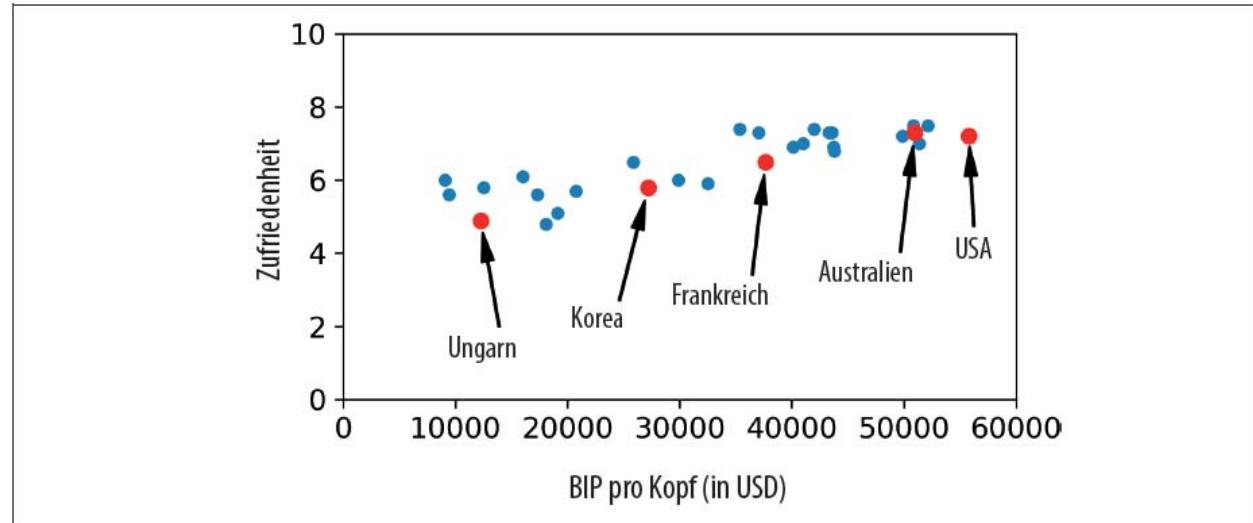


Abbildung 1-17: Sehen Sie hier eine Tendenz?

Es scheint so etwas wie einen Trend zu geben! Auch wenn die Daten *verrauscht* sind (also teilweise zufällig), sieht es so aus, als steige die Zufriedenheit mehr oder weniger mit dem Pro-Kopf-BIP des Landes linear an. 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 (siehe 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.

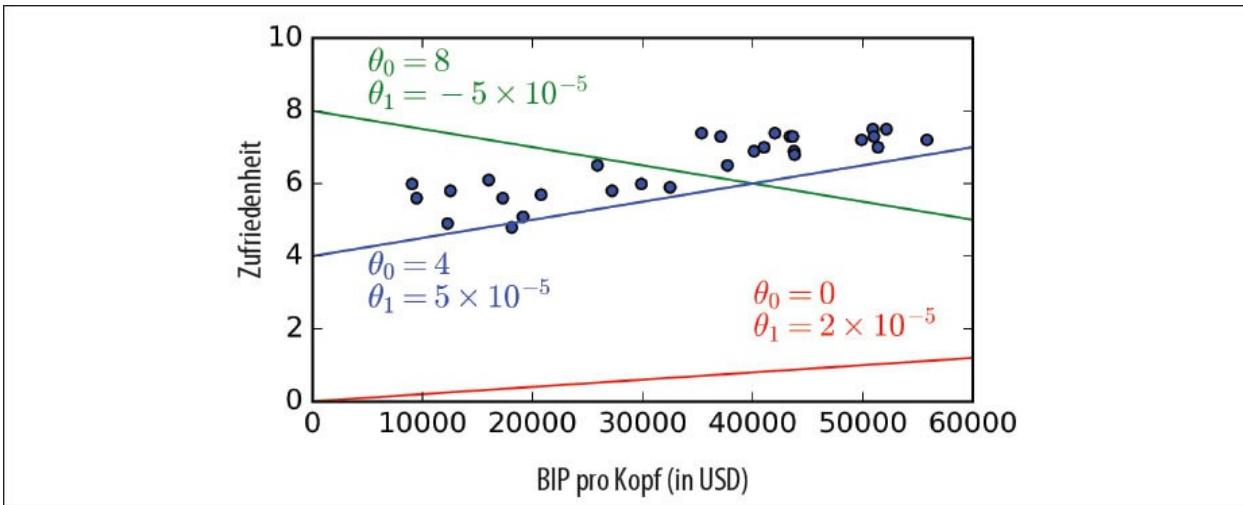


Abbildung 1-18: Einige mögliche lineare Modelle

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.

 Verwirrenderweise kann sich das gleiche Wort »Modell« auf eine Art von Modell (zum Beispiel lineare Regression), auf eine vollständig spezifizierte Modellarchitektur (zum Beispiel lineare Regression mit einer Eingabe und einer Ausgabe) oder auf das abschließende trainierte Modell, das für Vorhersagen genutzt werden kann (zum Beispiel lineare Regression mit einer Eingabe und einer Ausgabe und $\theta_0 = 4,85$ und $\theta_1 = 4,91 \times 10^{-5}$), verwenden werden. Die Modellauswahl besteht darin, die Art des Modells auszuwählen und seine Architektur vollständig zu definieren. Beim Trainieren eines Modells geht es darum, einen Algorithmus laufen zu lassen, der die Modellparameter findet, mit denen es am besten zu den Trainingsdaten passt (und hoffentlich gute Vorhersagen für neue Daten trifft).

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

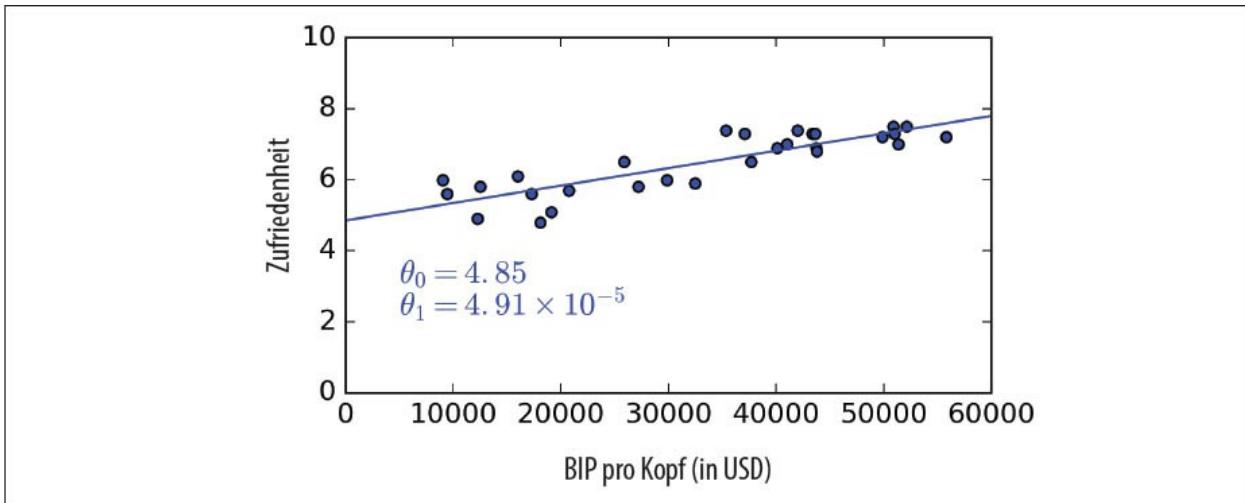


Abbildung 1-19: Das an die Trainingsdaten optimal angepasste lineare Modell

Sie sind nun endlich so weit, 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.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model

# 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]]

```



Hätten Sie einen instanzbasierten Lernalgorithmus verwendet, 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 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 dieser beiden Zeilen:

```

import sklearn.linear_model
model = sklearn.linear_model.LinearRegression()

```

durch diese zwei:

```

import sklearn.neighbors
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 Ähnliches), 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« sowie »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 ist eine Vielzahl an 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 (es sei denn, Sie können Teile eines existierenden Modells wiederverwenden).

Die unverschämte Effektivität von Daten

In einem berühmten Artikel (<https://homl.info/6>) 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).

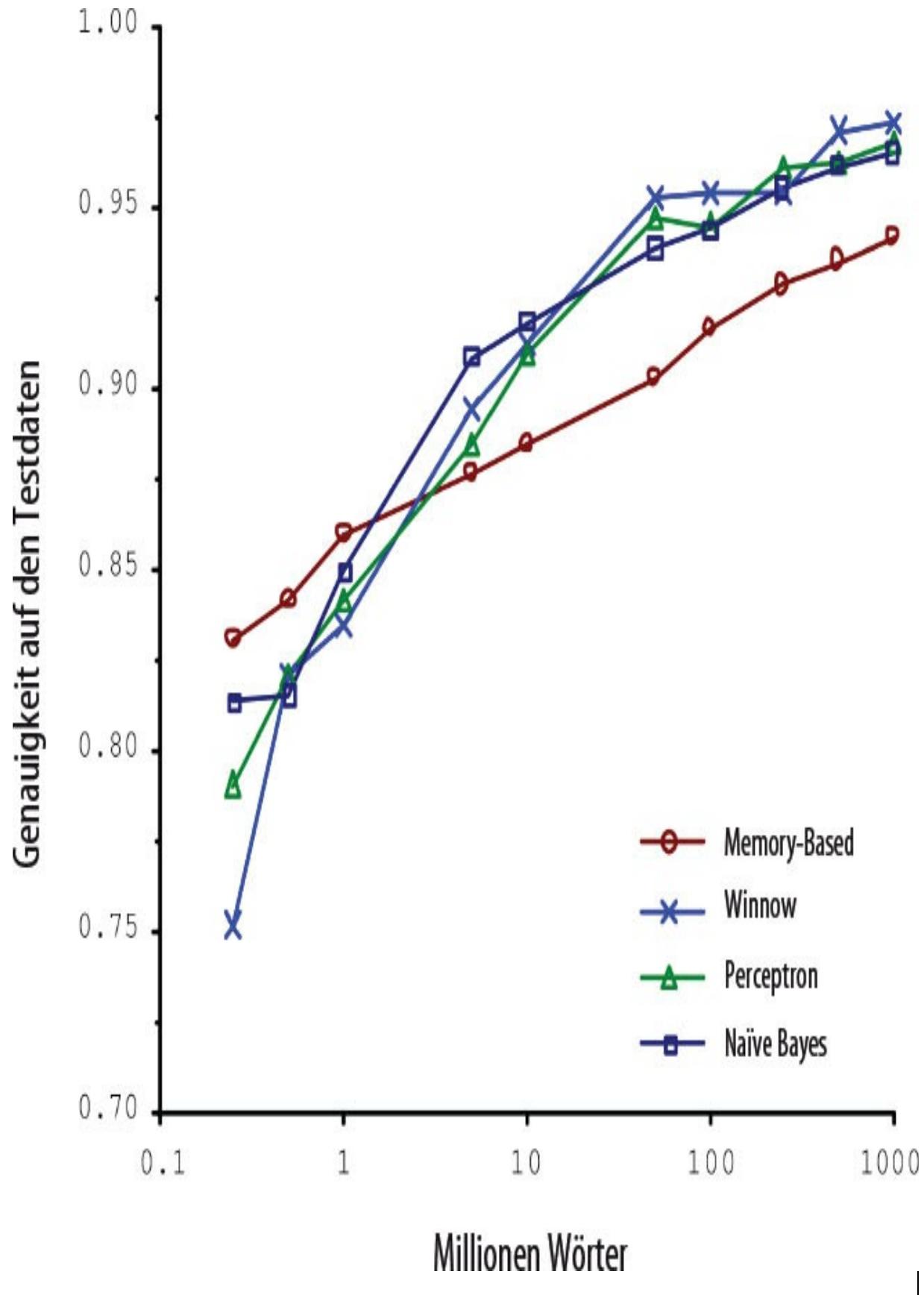


Abbildung 1-20: Die Wichtigkeit der Daten im Vergleich zum Algorithmus⁹

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« (<https://homl.info/7>), veröffentlicht im Jahr 2009, weiter thematisiert.¹⁰ 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 als auch 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.

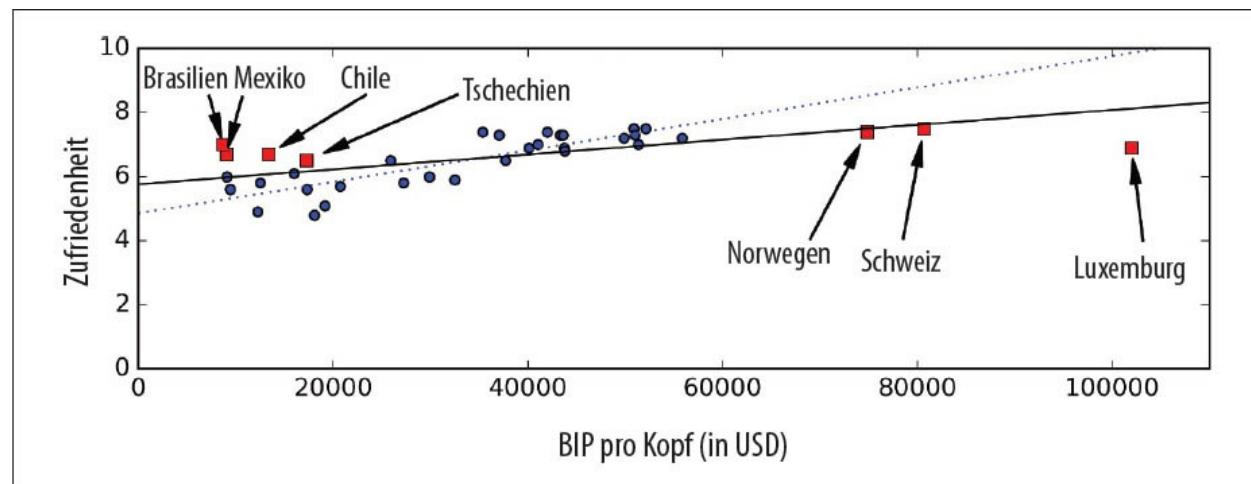


Abbildung 1-21: Ein repräsentativerer Trainingsdatensatz

Wenn Sie mit diesen Daten ein lineares Modell trainieren, erhalten Sie die durchgezogene Linie. Das alte Modell ist durch die gepunktete Linie gekennzeichnet. 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 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 ungenaue

Vorhersagen, besonders bei sehr armen und sehr reichen Ländern.

Es ist daher wichtig, einen Trainingsdatensatz zu verwenden, in dem die zu verallgemeinernden Fälle abgebildet sind. Oft ist dies schwieriger, als es sich anhört: 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*.

Beispiele für Stichprobenverzerrungen

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. Nach dem Sammeln von 2,4 Millionen Antworten wurde 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:

- Zum einen 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 eher für die Republikaner (und damit Landon) stimmen würden.
- Zum anderen 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 potenziell 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 jedoch überrepräsentiert sein (und wenn Sie in Brasilien leben, erhalten Sie eine Menge Videos zu »funk carioca«, die sich überhaupt nicht wie James Brown anhören). Wie aber sonst sollte man 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 es 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 manuell 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 wollen oder 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.

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:

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

Nun, da wir viele Beispiele für schlechte Daten kennengelernt haben, schauen wir uns auch noch 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?

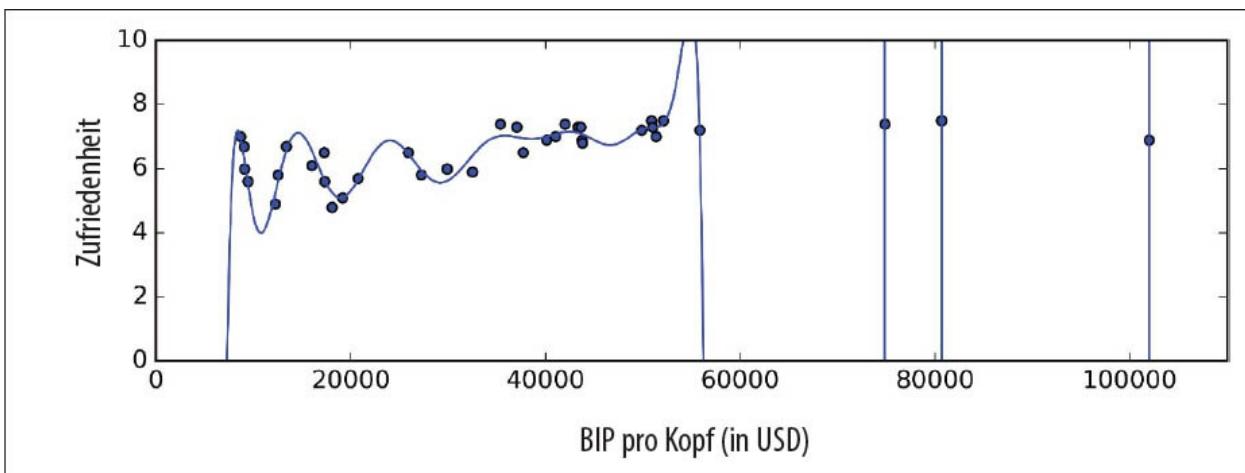


Abbildung 1-22: Overfitting der Trainingsdaten

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 stellten 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 polynomiellen 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 Freiheitsgrade. Das entstehende Modell ist einfacher als das mit zwei Freiheitsgraden, aber komplexer als das mit nur einem. Ihre Aufgabe ist es, 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 den als Kreis dargestellten Ländern trainierte Modell (ohne die als Quadrat dargestellten Länder), die gestrichelte Linie für unser zweites mit allen Ländern (Kreise und Quadrate) trainiertes 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. Dieses Modell passt nicht so gut zu den Trainingsdaten wie das erste Modell, erlaubt aber eine bessere Verallgemeinerung auf neue Beispiele, die es während des Trainings nicht kannte (Quadrate).

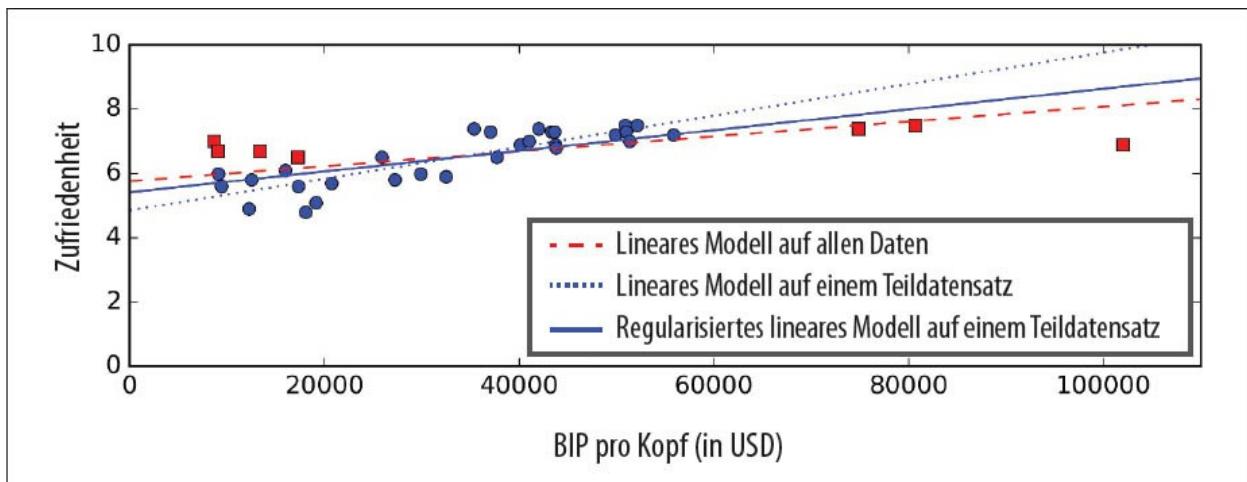


Abbildung 1-23: Regularisierung reduziert das Risiko für Overfitting.

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 einfach 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.
- In einem ML-Projekt sammeln Sie Daten in einem Trainingsdatensatz und speisen diesen in einen Lernalgorithmus ein. Wenn der Algorithmus auf einem Modell basiert, tunt er einige Parameter, um das Modell an die 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 verallgemeinert auf neue Instanzen durch ein Ähnlichkeitsmaß, um sie mit den bekannten Instanzen zu vergleichen.
- Wenn der Trainingsdatensatz zu klein ist oder 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 auch 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. Der Wert verrät Ihnen, wie gut Ihr Modell auf zuvor nicht bekannten Datenpunkten abschneidet.

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

- Es ist üblich, 80% der Daten zum Trainieren zu nehmen und 20% zum Testen *zurückzuhalten*. Das hängt aber auch von der Größe des Datensatzes ab: Enthält er 10 Millionen Instanzen, bedeutet ein Zurückhalten von 1%, dass Ihr Testdatensatz aus 100.000 Instanzen besteht – vermutlich mehr als genug für eine gute Abschätzung des Verallgemeinerungsfehlers.

Hyperparameter anpassen und Modellauswahl

Das Evaluieren eines Modells ist einfach: Verwenden Sie einen Testdatensatz. Aber möglicherweise schwanken Sie zwischen zwei Typen von Modellen (z.B. einem linearen und einem polynomiellen Modell): Wie sollen Sie sich zwischen ihnen 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 wählen Sie den Wert des Regularisierungshyperparameters aus? Natürlich können Sie 100 unterschiedliche Modelle mit 100 unterschiedlichen Werten für diesen Hyperparameter trainieren. Angenommen, 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 nennt sich *Hold-out-Validierung*: Sie halten einfach einen Teil des Trainingsdatensatzes zurück, um die verschiedenen Modellkandidaten zu bewerten, und wählen den besten aus. Dieser neue zurückgehaltene Datensatz wird als *Validierungsdatensatz* (oder manchmal auch als *Entwicklungsdatensatz* oder *Dev Set*) bezeichnet. Genauer gesagt, trainieren Sie mehrere Modelle mit unterschiedlichen Hyperparametern auf dem Trainingsdatensatz und wählen das auf dem Validierungsdatensatz am besten abschneidende Modell und die Hyperparameter aus. Nach diesem Hold-out-Validierungsprozess trainieren Sie das beste Modell mit dem vollständigen Trainingsdatensatz (einschließlich des Validierungsdatensatzes) und erhalten so das endgültige Modell. Schließlich führen Sie einen einzelnen abschließenden Test an diesem finalen Modell mit dem Testdatensatz durch, um den Verallgemeinerungsfehler abzuschätzen.

Dieses Vorgehen funktioniert meist ziemlich gut. Ist aber der Validierungsdatensatz zu klein, wird die Modellbewertung ungenau sein – Sie landen eventuell unabsichtlich bei einem suboptimalen Modell. Ist der Validierungsdatensatz hingegen zu groß, wird der verbleibende

Trainingsdatensatz viel kleiner sein als der vollständige Trainingsdatensatz. Warum ist das schlecht? Nun, da das finale Modell mit dem vollständigen Trainingsdatensatz trainiert werden wird, ist es nicht ideal, Modellkandidaten zu vergleichen, die mit einem viel kleineren Trainingsdatensatz trainiert wurden. Das wäre so, als würden Sie den schnellsten Sprinter auswählen, damit er an einem Marathon teilnimmt. Eine Möglichkeit, dieses Problem zu lösen, ist eine wiederholt durchgeführte *Kreuzvalidierung* mit vielen kleinen Validierungsdatensätzen. Jedes Modell wird einmal per Validierungsdatensatz geprüft, nachdem es mit dem Rest der Daten trainiert wurde. Durch ein Mitteln aller Bewertungen eines Modells können Sie dessen Genauigkeit deutlich besser bestimmen. Es gibt aber einen Nachteil: Die Trainingsdauer wird mit der Anzahl der Validierungsdatensätze multipliziert.

Datendiskrepanz

In manchen Fällen kommt man zwar leicht an große Mengen an Trainingsdaten, diese sind aber keine perfekte Repräsentation der im Produktivumfeld verwendeten Daten. Stellen Sie sich zum Beispiel vor, Sie wollten eine Mobil-App bauen, die Bilder von Blumen aufnimmt und automatisch deren Spezies bestimmt. Sie können problemlos Millionen von Blumenbildern aus dem Web laden, aber sie werden nicht perfekt die Bilder repräsentieren, die tatsächlich mit der App auf einem Smartphone aufgenommen werden. Vielleicht haben Sie nur 10.000 repräsentative Bilder (also solche, die mit der App aufgenommen wurden). In diesem Fall ist es am wichtigsten, daran zu denken, dass der Validierungsdatensatz und der Testdatensatz in Bezug auf die produktiv genutzten Daten so repräsentativ wie möglich sein müssen, daher sollten sie nur aus repräsentativen Bildern bestehen: Sie können sie mischen, die eine Hälfte in den Validierungsdatensatz und die andere in den Testdatensatz stecken (wobei Sie darauf achten müssen, keine Dubletten oder Nahezu-Dubletten in beiden Sets zu haben). Aber beobachten Sie nach dem Trainieren Ihres Modells mit den Bildern aus dem Web, dass die Genauigkeit des Modells beim Validierungsdatensatz enttäuschend ist, werden Sie nicht wissen, ob das daran liegt, dass das Modell bezüglich des Trainingsdatensatzes overfittet oder ob es einfach einen Unterschied zwischen den Bildern aus dem Web und denen aus der App gibt. Eine Lösung ist, einen Teil der Trainingsbilder (aus dem Web) in einen weiteren Datensatz zu stecken, den Andrew Ng als *Train-Dev-Set* bezeichnet. Nachdem das Modell trainiert wurde (mit dem Trainingsdatensatz, *nicht* mit dem Train-Dev-Set), können Sie es auf das Train-Dev-Set loslassen. Funktioniert es dort gut, hat das Modell kein Overfitting bezüglich des Trainingsdatensatzes. Ist die Genauigkeit beim Validierungsdatensatz schlecht, muss das Problem aus der Datendiskrepanz entstanden sein. Sie können versuchen, die Lösung des Problems durch eine Vorverarbeitung der Webbilder anzugehen, sodass sie mehr wie die Bilder aus der App aussehen, und dann das Modell neu zu trainieren. Funktioniert das Modell hingegen schlecht mit dem Train-Dev-Set, muss es bezüglich des Trainingsdatensatzes overfittet sein, und Sie sollten versuchen, das Modell zu vereinfachen oder zu regularisieren sowie mehr Trainingsdaten zu erhalten und diese zu säubern.

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 verwerfen 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://homl.info/8>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 optimale 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.

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 häufig anzutreffende 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 Onlinelernsystem?
10. Was ist Out-of-Core-Lernen?
11. Welche Art Lernalgorithmus beruht auf einem Ähnlichkeitsmaß, um Vorhersagen zu treffen?

12. Worin besteht der Unterschied zwischen einem Modellparameter und einem Hyperparameter eines Lernalgorithmus?
13. Wonach suchen modellbasierte Lernalgorithmen? Welche Strategie führt am häufigsten zum Erfolg? 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 sollte man einen verwenden?
17. Was ist der Zweck eines Validierungsdatensatzes?
18. Was ist das Train-Dev-Set, wann brauchen Sie es, und wie verwenden Sie es?
19. Was kann schiefgehen, wenn Sie Hyperparameter mithilfe der Testdaten einstellen?

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 durchleben. Wir nehmen dazu an, Sie seien ein frisch angeheuerter Data Scientist in einer Immobilienfirma.¹ Dies sind die wichtigsten Schritte, die es zu bearbeiten gilt:

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 Tausende frei verfügbarer Datensätze aus allen möglichen Fachgebieten zur Auswahl. Hier sind einige Quellen, unter denen 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 (<https://registry.opendata.aws/>)
- Metaseiten (Listen von Datenarchiven):
 - Data Portals (<http://dataportals.org/>)
 - OpenDataMonitor (<http://opendatamonitor.eu/>)
 - Quandl (<http://quandl.com/>)
- Andere Seiten, die beliebte offene Datenarchive auflisten:
 - Die Wikipedia-Seite mit Machine-Learning-Datensätzen (<https://homl.info/9>)
 - Quora.com (<https://homl.info/10>)
 - 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 deren 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.

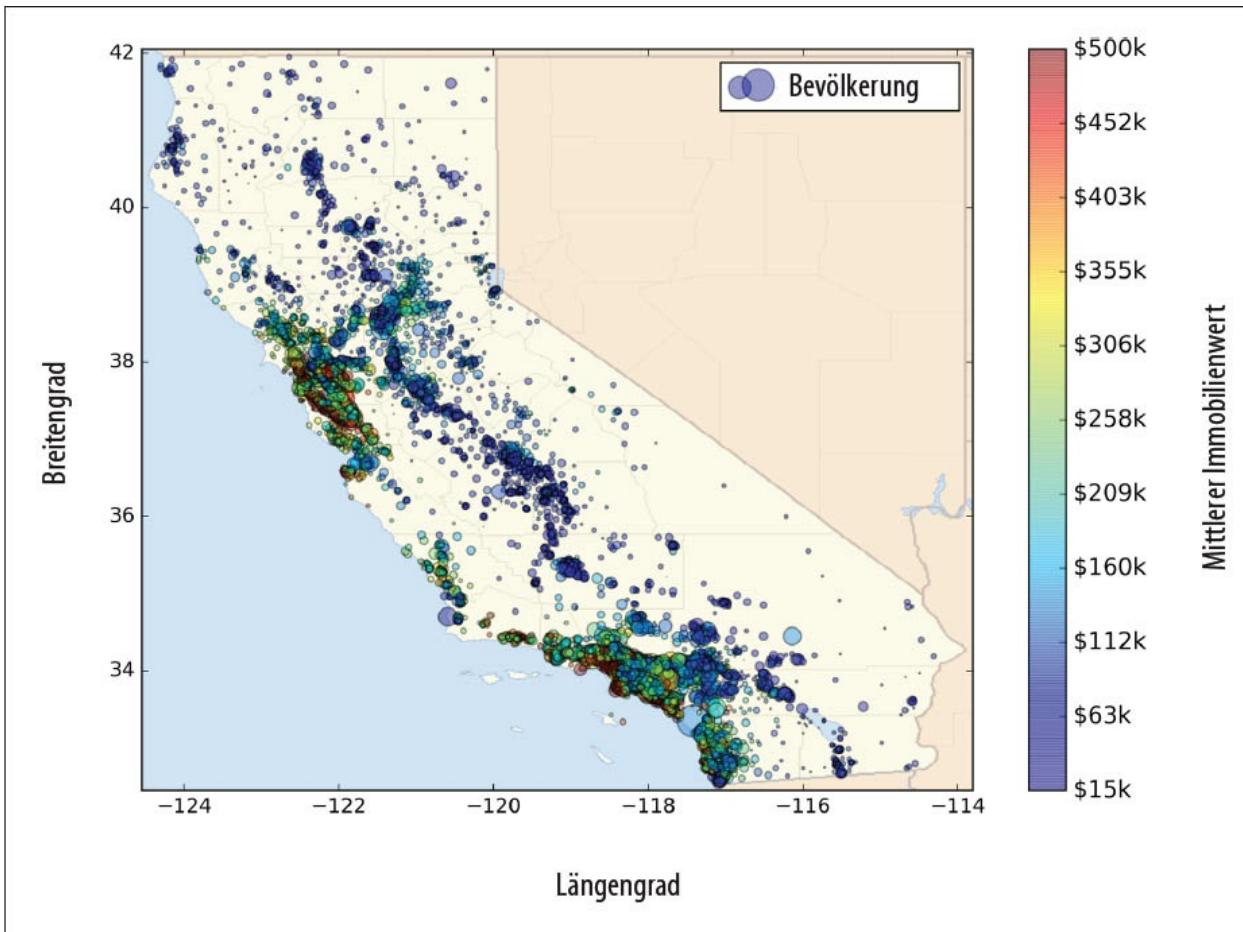


Abbildung 2-1: Immobilienpreise in Kalifornien

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 3.000 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 sei; 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.

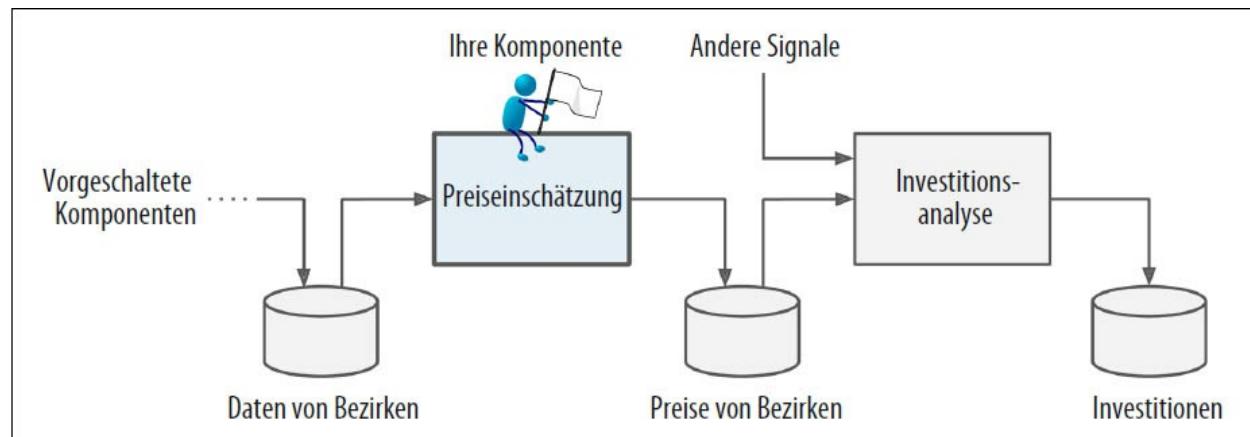


Abbildung 2-2: Eine Machine-Learning-Pipeline für Immobilieninvestitionen

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, die Sie Ihrem Chef stellen sollten, 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 um mehr als 10% danebenlag. Deshalb 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 gut 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 Sie, 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 *multiple Regressionsaufgabe*, da das System mehrere Eigenschaften zum Treffen einer Vorhersage heranziehen wird (die Bevölkerung eines Bezirks, das mittlere Einkommen und so weiter). Gleichzeitig ist es auch eine *univariate Regressionsaufgabe*, da wir nur versuchen, für jeden Bezirk einen einzelnen Wert vorherzusagen. Würden wir versuchen, mehrere Werte pro Bezirk vorherzusagen, handelte es sich um eine *multivariate 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) 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)

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

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 2.000 Bezirken auswerten, dann gilt $m = 2000$.
- $\mathbf{x}^{(i)}$ ist ein Vektor der Werte aller Merkmale (ohne das Label) des i . Datenpunkts im Datensatz, und $y^{(i)}$ ist das dazugehörige Label (der gewünschte Ausgabewert für diesen Datenpunkt).
 - Wenn der erste Bezirk beispielsweise bei $-118,29^\circ$ Länge und $33,91^\circ$ nördlicher Breite liegt, 1.416 Einwohner mit einem mittleren Einkommen von 38.372 USD hat und der mittlere Immobilienpreis 156.400 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 . 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:

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

- 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.
 - Sagt Ihr System beispielsweise im ersten Bezirk einen mittleren Immobilienpreis von 158.400 USD vorher, so ist $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158400$. Der Vorhersagefehler für diesen Bezirk ist dann $\hat{y}^{(1)} - y^{(1)} = 2000$.
- $\text{RMSE}(\mathbf{X}, h)$ 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, sollten Sie in manchen Situationen eine andere Funktion vorziehen. 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.
- 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.

- Allgemeiner ist die ℓ_k -Norm eines Vektors \mathbf{v} mit n Elementen als
$$\|\mathbf{v}\|_k = \left(|v_0|^k + |v_1|^k + \dots + |v_n|^k \right)^{\frac{1}{k}}$$
 definiert. ℓ_0 gibt einfach nur die Anzahl der Elemente ungleich null im Vektor wieder, und ℓ_∞ berechnet den größten Absolutwert im Vektor.
- Je höher der Index einer Norm ist, umso stärker berücksichtigt er große Werte und vernachlässigt kleinere. Deshalb ist der RMSE empfindlicher für Ausreißer als der MAE. Sind Ausreißer aber exponentiell selten (wie in einer Glockenkurve), funktioniert der RMSE sehr gut und ist grundsätzlich vorzuziehen.

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 dem Fall wäre es überhaupt nicht wichtig, den Preis genau vorherzusagen; Ihr System müsste lediglich die Kategorie richtig bestimmen. Ist das der Fall, sollte die Aufgabe als Klassifikation beschrieben werden, nicht als Regression. Zu dieser Art von Erkenntnissen möchte man nicht erst nach monatelanger Arbeit an einem Regressionssystem 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-ml2>.

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
```

Sie benötigen ein paar Python-Module: Jupyter, NumPy, pandas, Matplotlib und Scikit-Learn. Wenn Jupyter bei Ihnen bereits mit all diesen Modulen läuft, können Sie beruhigt bei »[Die Daten herunterladen](#)« auf [Seite 48](#) fortfahren. Sollte noch etwas fehlen, 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:

```
$ python3 -m pip --version  
pip 19.0.2 from [...]/lib/python3.6/site-packages (python 3.6)
```

Sie sollten sicherstellen, dass Sie eine aktuelle Version von pip haben. Um das pip-Modul zu aktualisieren, geben Sie Folgendes ein (die genaue Version mag eine andere sein):⁷

```
$ python3 -m pip install --user -U pip  
Collecting pip  
[...]  
Successfully installed pip-19.0.2
```

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 virtualenv⁸ installieren: (Auch hier gilt wieder: Wollen Sie virtualenv für alle Benutzer auf Ihrem Rechner installieren, entfernen Sie --user und lassen diesen Befehl mit Administratorrechten laufen.)

```
$ python3 -m pip install --user -U virtualenv  
Collecting virtualenv  
[...]  
Successfully installed virtualenv
```

Nun können Sie eine isolierte Python-Umgebung erstellen:

```
$ cd $ML_PATH
```

```
$ virtualenv my_env

Using base prefix '[...]'  
New python executable in [...]/ml/my_env/bin/python3.6  
Also creating executable in [...]/ml/my_env/bin/python  
Installing setuptools, pip, wheel...done.
```

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

```
$ cd $ML_PATH  
$ source my_env/bin/activate # unter Linux oder macOS  
$ .\my_env\Scripts\activate # unter Windows
```

Um diese Umgebung zu deaktivieren, geben Sie **deactivate** ein. 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 (verwenden Sie keine `virtualenv`, benötigen Sie die Option `--user` oder Administratorrechte):

```
$ python3 -m pip install --upgrade jupyter matplotlib numpy pandas scipy scikit-learn  
  
Collecting jupyter  
  Downloading jupyter-1.0.0-py2.py3-none-any.whl  
  
Collecting matplotlib  
[...]
```

Haben Sie eine `virtualenv` installiert, müssen Sie sie für Jupyter registrieren und ihr einen Namen geben:

```
$ $ python3 -m ipykernel install --user --name=python3
```

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 das 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« ① und wählen Sie die gewünschte Python-Version aus ②⁹ (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, und drittens wird das Notebook in einem neuen Unterfenster geöffnet. Zu Beginn sollten Sie das Notebook in »Housing« umbenennen ① (siehe Abbildung 2-4), indem Sie auf »Untitled« klicken und den neuen Namen eingeben (damit wird auch die Datei automatisch zu *Housing.ipynb* umbenannt).

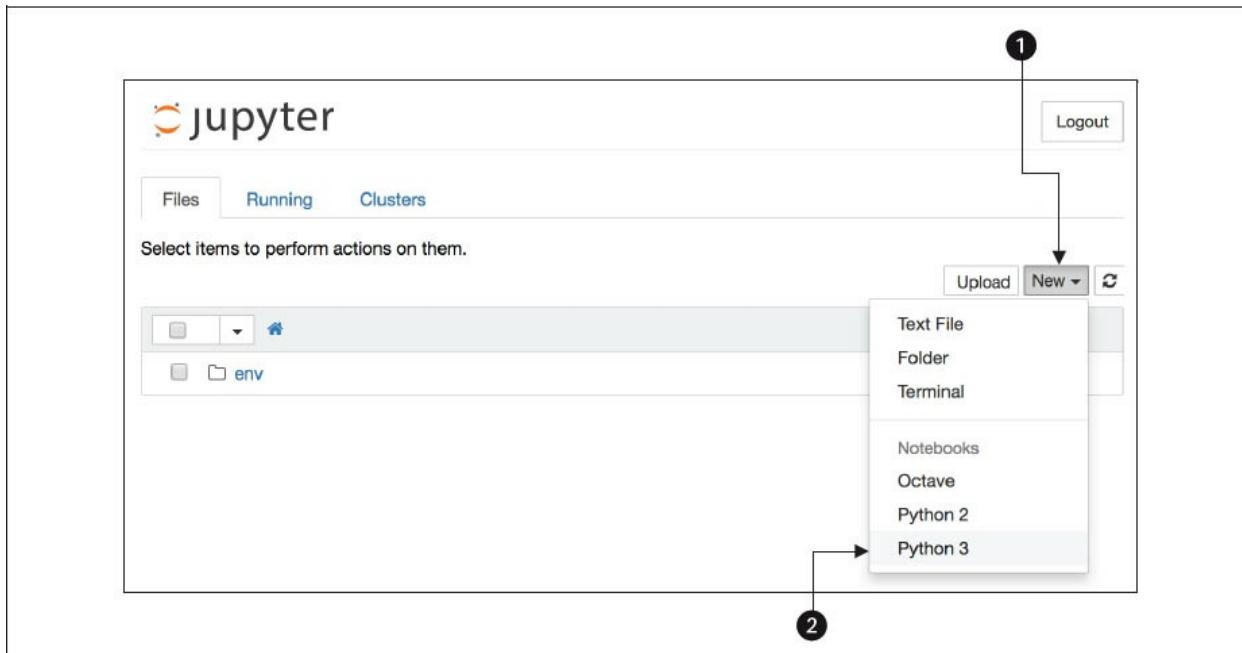


Abbildung 2-3: Ihre Arbeitsumgebung in Jupyter

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 Codezelle mit dem

Label *In [1]*: Schreiben Sie in die Zelle `print("Hello world!")` ② und drücken Sie dann auf den Play-Button ③ (siehe Abbildung 2-4) oder drücken Sie Umschalt-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 oder Dateien verteilt. Um auf sie zuzugreifen, müssten Sie zuerst Zugriffsrechte und Passwörter erhalten¹⁰ 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 nutzen und anschließend `tar xzf housing.tgz` zum Entpacken und Extrahieren der CSV-Datei eingeben, es ist aber besser, dazu eine kleine Funktion zu schreiben. Eine solche Funktion 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:¹¹

```
import os  
  
import tarfile  
  
import urllib
```

```

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"

HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    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` wird heruntergeladen, und die Datei `housing.csv` wird in dieses Verzeichnis entpackt.

Nun laden Sie die Daten mit pandas. Auch diesmal sollten Sie eine kleine Funktion zum Laden der Daten 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](#)).

In [5]:	housing = load_housing_data() housing.head()																																										
Out[5]:	<table border="1"> <thead> <tr> <th></th><th>longitude</th><th>latitude</th><th>housing_median_age</th><th>total_rooms</th><th>total_bedrooms</th><th>population</th></tr> </thead> <tbody> <tr> <td>0</td><td>-122.23</td><td>37.88</td><td>41.0</td><td>880.0</td><td>129.0</td><td>322.0</td></tr> <tr> <td>1</td><td>-122.22</td><td>37.86</td><td>21.0</td><td>7099.0</td><td>1106.0</td><td>2401.0</td></tr> <tr> <td>2</td><td>-122.24</td><td>37.85</td><td>52.0</td><td>1467.0</td><td>190.0</td><td>496.0</td></tr> <tr> <td>3</td><td>-122.25</td><td>37.85</td><td>52.0</td><td>1274.0</td><td>235.0</td><td>558.0</td></tr> <tr> <td>4</td><td>-122.25</td><td>37.85</td><td>52.0</td><td>1627.0</td><td>280.0</td><td>565.0</td></tr> </tbody> </table>		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
	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. Dies sind insbesondere die Anzahl der Zeilen, der Typ jedes Attributs und die Anzahl der Werte ungleich null (siehe Abbildung 2-6).

In [6]:	housing.info()
	<pre><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</pre>

Abbildung 2-6: Informationen zu Immobilien

Im Datensatz gibt es 20.640 Datenpunkte. Damit ist er für Machine-Learning-Verhältnisse eher klein, für den Anfang ist das aber ausgezeichnet! Beachten Sie, dass das Merkmal total_bedrooms nur 20.433 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 (siehe 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	20640.000000	20433.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	3
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1000.000000
min	-124.350000	32.540000	1.000000	2.000000	1.000000	0
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	0
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	0
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	0
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	0

Abbildung 2-7: Zusammenfassung aller numerischen Merkmale

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*, die die Streuung der Werte angibt.¹² 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 (wie im folgenden Codebeispiel) und für jedes numerische Merkmal ein Histogramm erhalten (siehe Abbildung 2-8):

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

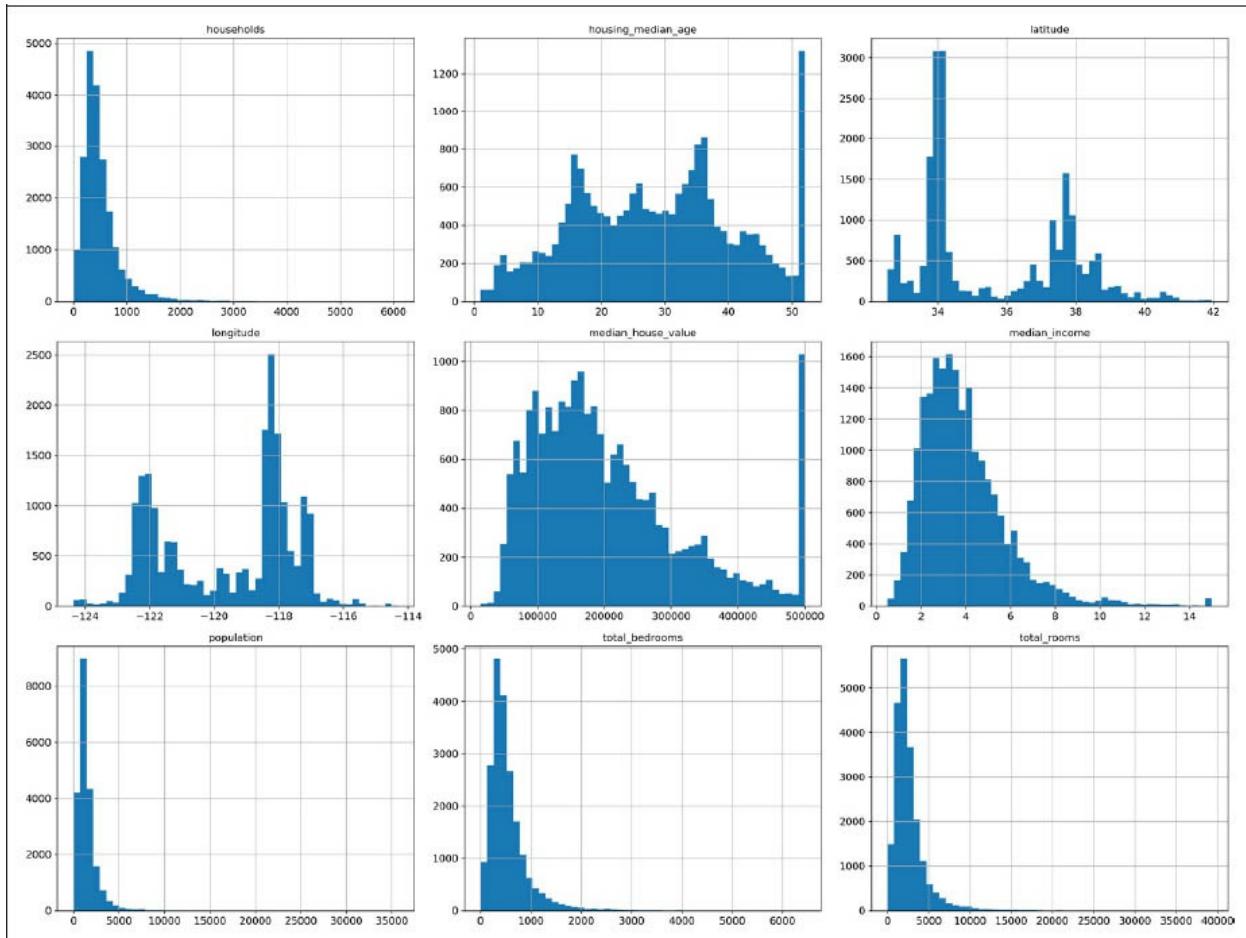


Abbildung 2-8: Ein Histogramm für jedes numerische Attribut

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` einzusetzen. Dieses weist Jupyter an, Matplotlib zu nutzen, 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 einiges zu sehen:

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. Die Zahlen stehen ungefähr für 10.000 USD (eine 3 bedeutet also etwa 30.000 USD). 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 das ein Problem darstellt oder nicht. Wenn Ihnen erklärt wird, dass auch jenseits von 500.000 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 50.000 USD vorhersagt).
3. Diese Attribute haben sehr unterschiedliche Wertebereiche. Wir werden das weiter unten in diesem 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 das *Data-Snooping-Bias*.

Einen Testdatensatz zu erstellen, ist theoretisch einfach: Wählen Sie zufällig einige Datenpunkte aus, meist 20% des Datensatzes (oder weniger, wenn Ihr Datensatz sehr groß ist), 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:¹³

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

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 besteht darin, 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)`)¹⁴, bevor Sie `np.random.permutation()` aufrufen, sodass jedes Mal die gleichen durchmischten Indizes generiert werden.

Allerdings scheitern beide Lösungsansätze, sobald Sie einen aktualisierten Datensatz erhalten. Um auch danach über eine stabile Trennung zwischen Trainings- und Testdatensatz zu verfügen, können Sie als Alternative 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 und den Datenpunkt in den Testdatensatz aufzunehmen, falls der Hash kleiner oder gleich 20% des maximalen Hashwerts ist. 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:

```
from zlib import crc32

def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32

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

Leider gibt es im Immobiliendatensatz keine Identifikatorpalte. Die Lösung ist, den Zeilenindex als ID zu nutzen:

```
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 das nicht möglich ist, können Sie immer noch versuchen, einen eindeutigen Identifikator aus den stabilsten Merkmalen zu entwickeln. Beispielsweise werden geografische Länge und Breite garantiert für die nächsten paar Millionen Jahre stabil bleiben, daher könnten Sie diese folgendermaßen zu einer ID kombinieren:¹⁵

```
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_state`, der den Seed-Wert des Zufallszahlengenerators festlegt. Und zweitens können Sie mehrere Datensätze mit einer identischen Anzahl Zeilen übergeben, die anhand der gleichen Indizes aufgeteilt werden (das 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 1.000 Personen anruft, um diesen Fragen zu stellen, wählt es nicht einfach nur zufällig 1.000 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 hatten Ihnen in einer Unterhaltung erklart, dass das mittlere Einkommen ein sehr wichtiges Merkmal zur Vorhersage des mittleren Immobilienpreises ist. Sie mochten sicherstellen, dass der Testdatensatz die unterschiedlichen im Datensatz enthaltenen Einkommensklassen gut repräsentiert. Da das mittlere Einkommen ein stetiges numerisches Merkmal ist, mussen Sie zuerst ein kategorisches Merkmal fur das Einkommen generieren. Betrachten wir das Histogramm des Einkommens etwas genauer (siehe [Abbildung 2-8](#)): Die meisten mittleren Einkommen liegen bei 1,6 bis 6 (also 15.000 bis 60.000 USD), aber einige mittlere Einkommen liegen deutlich 躡er 6. Es ist wichtig, dass Ihr Datensatz fur jedes Stratum eine genugende Anzahl Datenpunkte enthalt, andernfalls liegt ein Bias fur die Schatzung der Wichtigkeit des Stratoms vor. Das heit, Sie sollten nicht zu viele Strata haben, und jedes Stratum sollte gro genug sein. Der folgende Code nutzt die Funktion `pd.cut()`, um ein kategorisches Merkmal fur das Einkommen mit fnf Kategorien zu erzeugen (mit den Labels 1 bis 5): Kategorie 1 reicht von 0 bis 1,5 (also weniger als 15.000 USD), Kategorie 2 von 1,5 bis 3 und so weiter:

```
labels=[1, 2, 3, 4, 5])
```

Diese Einkommenskategorien sind in [Abbildung 2-9](#) dargestellt:

```
housing["income_cat"].hist()
```

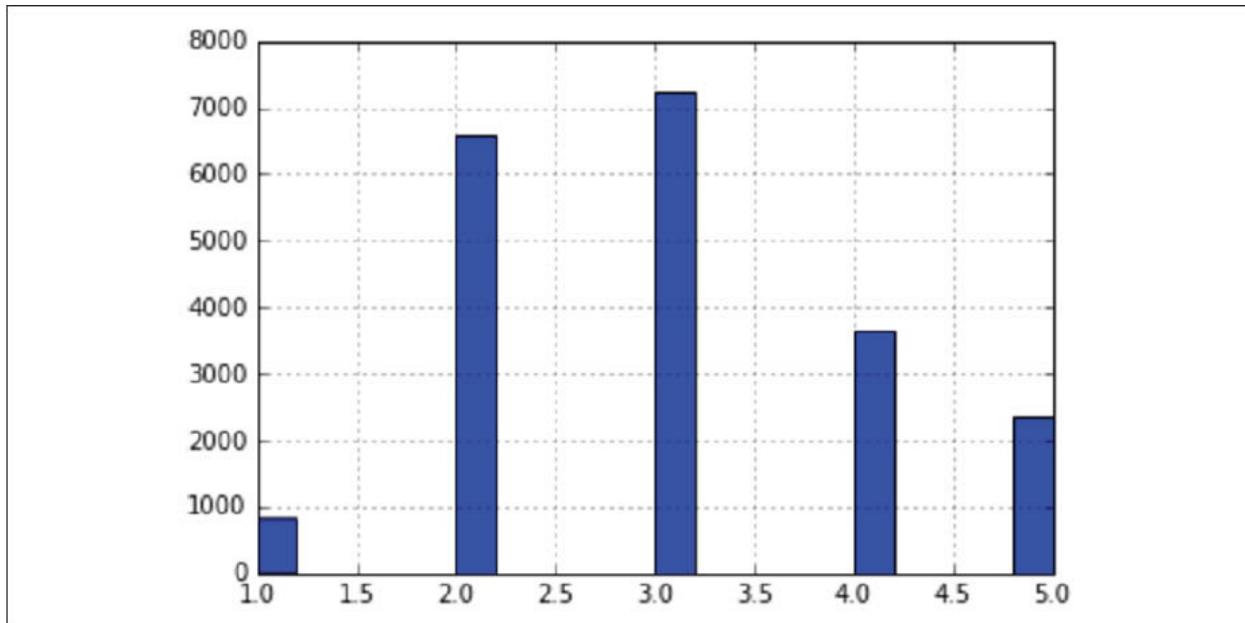


Abbildung 2-9: Histogramm der Einkommenskategorien

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 das wie erwartet funktioniert hat. Zunächst einmal können Sie sich die Anteile der Einkommenskategorien im Testdatensatz ansehen:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)

3      0.350533
2      0.318798
```

```

4    0.176357
5    0.114583
1    0.039729
Name: income_cat, dtype: float64

```

Mit einer ähnlichen Codezeile lassen sich die Anteile der Einkommenskategorien im vollständigen Datensatz 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 Gesamtdatensatz, 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 (siehe [Abbildung 2-11](#)):

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

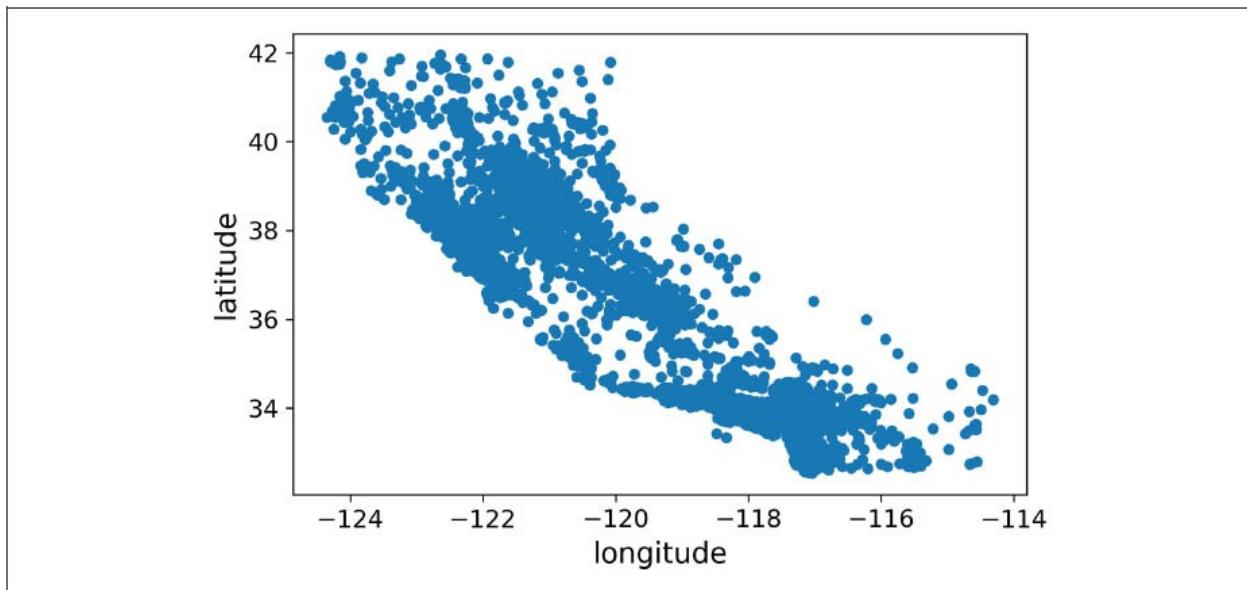


Abbildung 2-11: Ein geografischer Scatterplot der Daten

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 (siehe [Abbildung 2-12](#)):

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

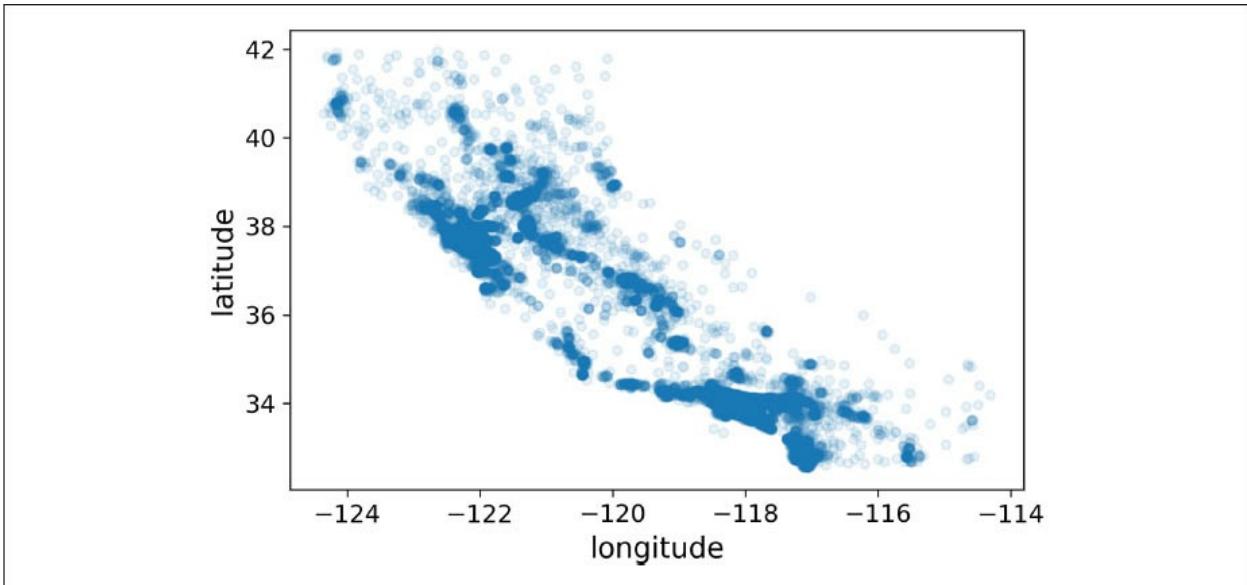


Abbildung 2-12: Eine bessere Darstellung von Gebieten mit hoher Dichte

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.

Unser Gehirn ist 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 (siehe Abbildung 2-13). Der Radius jedes Kreises stellt die Bevölkerung eines Bezirks dar (Option s), und die Farbe repräsentiert den Preis (Option c). Wir verwenden eine vordefinierte Farbskala (Option cmap) namens jet, die von Blau (niedrige Werte) bis Rot (hohe Werte) reicht:¹⁶

```
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()
```

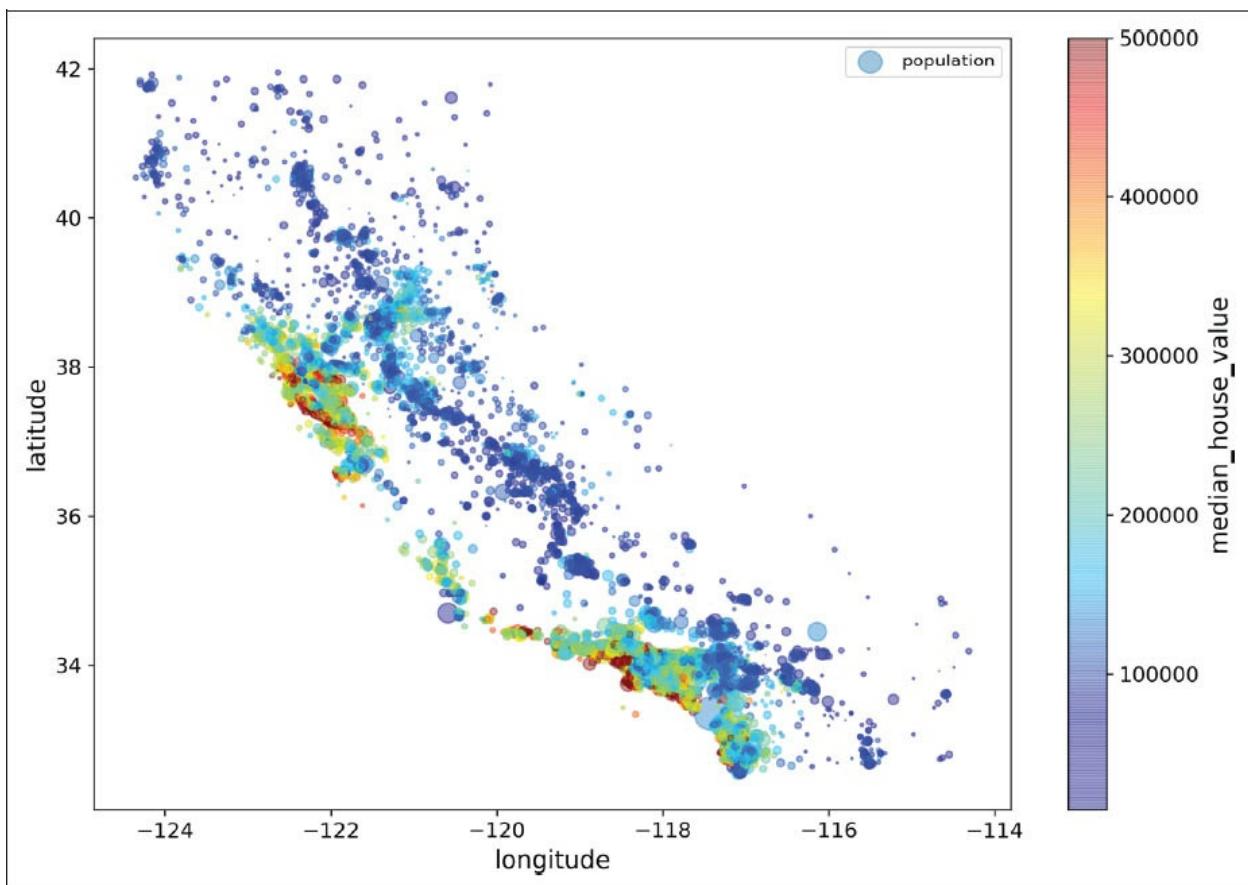


Abbildung 2-13: Immobilienpreise in Kalifornien: Rot ist teuer, Blau bedeutet günstig, größere Kreise stehen für Bereiche mit mehr Bevölkerung.

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.

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; zum Beispiel 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.

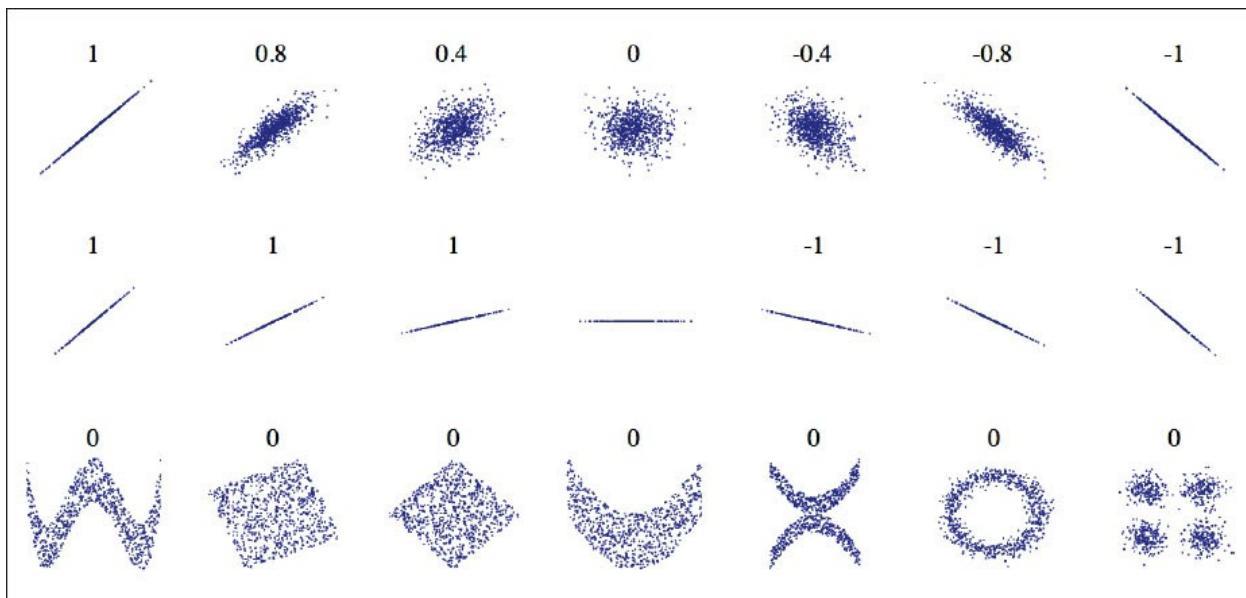


Abbildung 2-14: Korrelationskoeffizienten unterschiedlicher Datensätze (Quelle: Wikipedia; public domain image)

 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. Weil es elf numerische Merkmale gibt, würden Sie $11^2 = 121$ Diagramme erhalten. Da diese aber nicht auf eine Seite passen, konzentrieren wir uns auf einige vielversprechende Merkmale, die am ehesten mit dem mittleren Immobilienpreis korrelieren (siehe [Abbildung 2-15](#)):

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
               "housing_median_age"]

scatter_matrix(housing[attributes], figsize=(12, 8))
```

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).

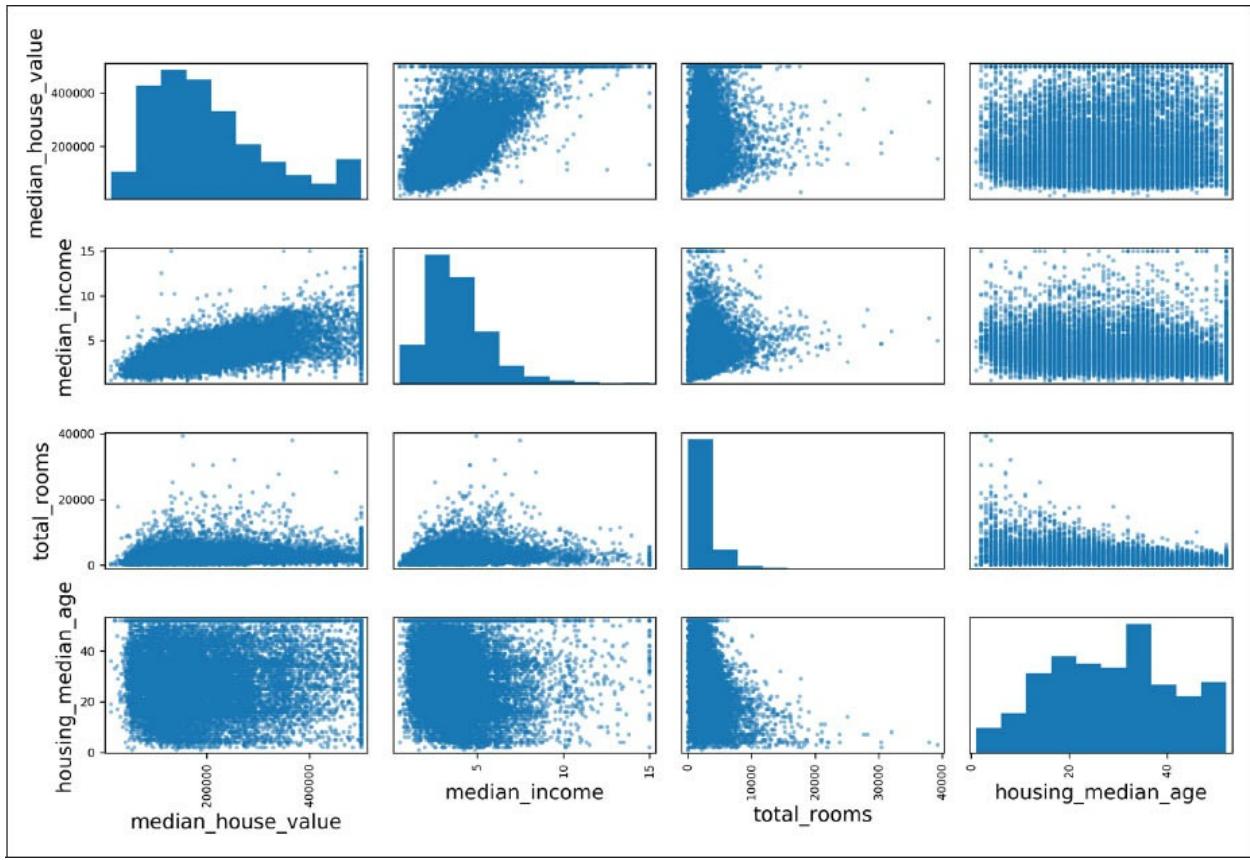


Abbildung 2-15: Diese Scatterplot-Matrix zeigt jedes numerische Attribut gegen jedes andere numerische Attribut und dazu ein Histogramm.

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

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

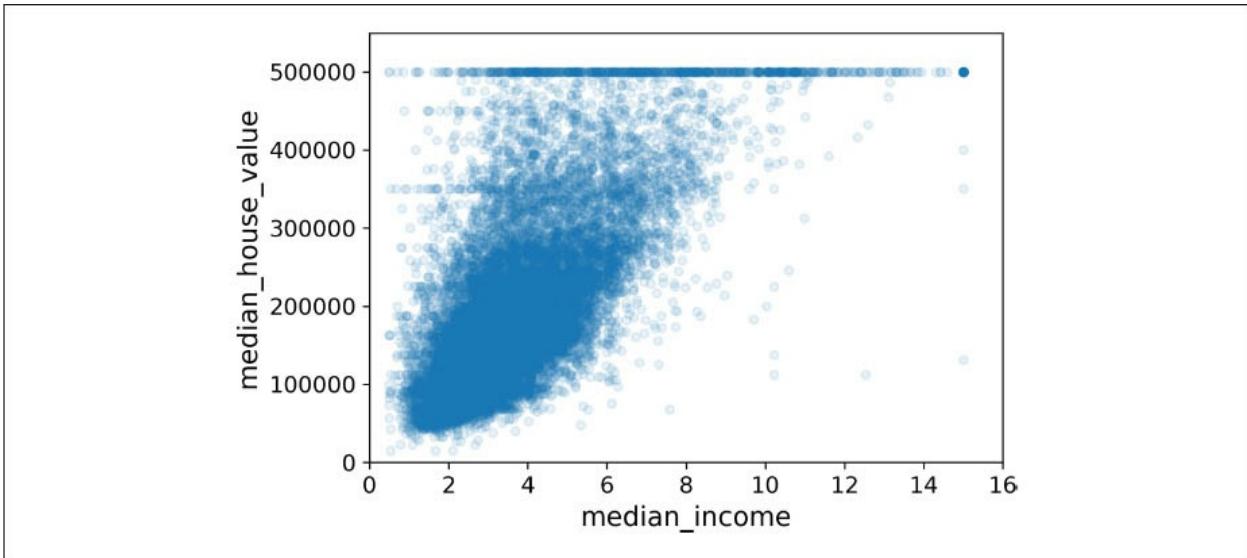


Abbildung 2-16: Mittleres Einkommen gegen mittleren Immobilienwert

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 500.000 USD. Das Diagramm zeigt aber auch weniger offensichtliche gerade Linien: eine horizontale Linie bei etwa 450.000 USD und eine zweite bei 350.000 USD, eventuell auch eine bei 280.000 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.

Experimentieren mit Kombinationen von Merkmalen

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

1. die entsprechenden Bezirke entfernen
2. das Merkmal komplett verwerfen
3. 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 um fehlende Werte in neuen Daten zu ersetzen, sobald Ihr System in Betrieb geht.

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

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

Da der Median sich nur bei numerischen Merkmalen berechnen lässt, müssen wir eine Kopie der Daten unter Ausschluß des Textmerkmals `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 im 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,  
                           index=housing_num.index)
```

Das Design von Scikit-Learn

Das Design der API von Scikit-Learn ist bemerkenswert gut gelungen. Die wichtigsten Designprinzipien (<https://homl.info/11>) sind:¹⁷

- **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 Fall 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 Landes vorher. Jeder Prädiktor besitzt die Methode `predict()`, die einen Datensatz mit neuen Datenpunkten annimmt und einen Satz entsprechender Vorhersagen zurückgibt. Jeder Prädiktor besitzt darüber hinaus die Methode `score()` zum Bestimmen der Vorhersagequalität anhand eines Testdatensatzes (und bei überwachten Lernalgorithmen auch der entsprechenden Labels).¹⁸
- **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 so weit wie möglich wiederverwendet. 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.
- **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

Bisher haben wir uns nur mit numerischen Attributen befasst, jetzt wollen wir uns um Textattribute kümmern. In diesem Datensatz gibt es nur eines: `ocean_proximity`. Schauen wir uns die Werte der ersten zehn Instanzen an:

```
>>> housing_cat = housing[["ocean_proximity"]]

>>> housing_cat.head(10)

ocean_proximity

17606      <1H OCEAN
18632      <1H OCEAN
14650      NEAR OCEAN
3230        INLAND
3555      <1H OCEAN
19480        INLAND
8879      <1H OCEAN
13685        INLAND
4937      <1H OCEAN
4861      <1H OCEAN
```

Das ist kein freier Text – es gibt eine begrenzte Zahl möglicher Werte, von denen jeder eine Kategorie repräsentiert. Daher handelt es sich bei diesem Attribut um ein kategorisches Merkmal. Die meisten Machine-Learning-Algorithmen bevorzugen Zahlen, daher werden wir

diese Kategorien von Text zu Zahlen konvertieren. Dazu verwenden wir die Klasse `OrdinalEncoder`.¹⁹

```
>>> from sklearn.preprocessing import OrdinalEncoder  
>>> ordinal_encoder = OrdinalEncoder()  
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)  
>>> housing_cat_encoded[:10]  
array([[0.],  
       [0.],  
       [4.],  
       [1.],  
       [0.],  
       [1.],  
       [0.],  
       [1.],  
       [0.],  
       [0.]])
```

Die Liste der Merkmale erhalten Sie über die Instanzvariable `categories_`. Dabei handelt es sich um ein eindimensionales Array mit Kategorien für jedes kategorische Merkmal (in diesem Fall enthält die Liste lediglich ein einzelnes Array, da es nur ein kategorisches Merkmal gibt):

```
>>> ordinal_encoder.categories_  
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],  
      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. Das mag in manchen Fällen in Ordnung sein (zum Beispiel für Merkmale mit einer Ordnung wie »schlecht«, »durchschnittlich«, »gut« und »ausgezeichnet«), aber es ist offensichtlich nicht der Fall für die Spalte `ocean_proximity` (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). Die neuen Merkmale werden manchmal als *Dummy*-Merkmale bezeichnet. Scikit-Learn bietet eine Klasse `OneHotEncoder` an, um kategorische Merkmalswerte in One-Hot-Vektoren umzuwandeln:²⁰

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

Beachten Sie, 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,²¹ aber wenn Sie sie wirklich in ein (dichtes) NumPy-Array umwandeln möchten, rufen Sie die Methode `toarray()` auf:

```
>>> housing_cat_1hot.toarray()  
  
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.]])
```

Wieder erhalten Sie eine Liste der Merkmale über die Instanzvariable `categories_` des Encoders:

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

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 # oder aus dem 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>
```

Standardmäßig gibt der `CategoricalEncoder` eine Sparse-Matrix aus, Sie können die Codierung 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-Codierung zu einer großen Zahl von Eingabemerkmalsen. Dies kann das Trainieren verlangsamen und die Leistung verringern. In diesem Fall können Sie die kategorischen Eingabewerte durch sinnvolle numerische Merkmale ersetzen, die damit in Zusammenhang stehen: Beispielsweise könnten Sie das Merkmal `ocean_proximity` durch den Abstand zum Ozean ersetzen (genauso ließe sich ein Ländercode durch die Bevölkerungszahl und das BSP pro Einwohner ersetzen). Alternativ könnten Sie jedes Merkmal durch einen erlernbaren, niedriger dimensionalen Vektor namens *Embedding* austauschen. Jede Repräsentation eines Merkmals würde während des Trainings gelernt werden. Dies ist ein Beispiel für *Representation Learning* (siehe die [Kapitel 13](#) und [17](#)).

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()` (die `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 Parameter `*args` und `**kwargs` 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, households_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):

    def __init__(self, add_bedrooms_per_room = True): # weder *args noch **kwargs
        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[:, households_ix]
```

```

population_per_household = X[:, population_ix] / X[:, households_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 sparen).

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 39.320, 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 Grund nicht 0 bis 1 nutzen 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 Standardabweichung 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 das 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 bis 15 in den Bereich 0 bis 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', SimpleImputer(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 eindeutig sind und keine doppelten Unterstriche enthalten (>__<); wenn es später an das Hyperparameter-Tuning geht, werden sie noch nützlich sein.

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 Fall 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 natürlich auch die Methode `fit_transform()`, die wir verwendet haben).

Bis jetzt haben wir die kategorischen und die numerischen Spalten getrennt behandelt. Es wäre aber praktischer, einen einzelnen Transformer zu haben, der sich um alle Spalten kümmern kann und die passende Transformation auf jede einzelne anwendet. In Version 0.20 hat Scikit-Learn für diesen Zweck den `ColumnTransformer` eingeführt, und der Vorteil ist, dass er sehr gut mit den `DataFrames` von pandas zusammenarbeitet. Nutzen wir ihn, um alle Transformationen auf die Immobiliendaten anzuwenden:

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)

cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", OneHotEncoder(), cat_attribs),
])

housing_prepared = full_pipeline.fit_transform(housing)
```

Zuerst importieren wir die Klasse `ColumnTransformer`, dann holen wir die Listen mit den Namen der numerischen und der kategorischen Spalten, und dann erstellen wir einen `ColumnTransformer`. Der Konstruktor benötigt eine Liste mit Tupeln, von denen jedes einen Namen,²² einen Transformer und eine Liste mit Namen (oder Indizes) von Spalten enthält, auf die der Transformer angewendet werden soll. In diesem Beispiel legen wir fest, dass die numerischen Spalten mit der weiter oben definierten `num_pipeline` transformiert werden sollen, während für die kategorischen Spalten ein `OneHotEncoder` zum Einsatz kommt. Schließlich wenden wir diesen `ColumnTransformer` auf die Immobiliendaten an: Er wendet jeden Transformer auf die entsprechenden Spalten an und verbindet die Ergebnisse entlang der zweiten Achse (die Transformer müssen die gleiche Zahl von Zeilen zurückgeben).

Beachten Sie, dass der `OneHotEncoder` eine Sparse-Matrix zurückgibt, während die `num_pipeline` eine Dense-Matrix liefert. Gibt es solch eine Mischung aus Sparse- und Dense-Matrix, schätzt der `ColumnTransformer` die Dichte der Ergebnismatrix (also das Verhältnis von Zellen, die nicht 0 sind) und gibt eine Sparse-Matrix zurück, wenn die Dichte unter einem

gegebenen Grenzwert liegt (Standard ist `sparse_threshold=0.3`). In diesem Beispiel wird eine Dense-Matrix geliefert. Und das ist alles! Wir haben eine Vorverarbeitungspipeline, die die gesamten Immobiliendaten nimmt und auf jede Spalte die passenden Transformationen anwendet.



Statt einen Transformer zu verwenden, können Sie den String "drop" angeben, wenn Spalten zu verwerfen sind, oder "pass through", wenn die Spalten unverändert belassen werden sollen. Standardmäßig werden die verbleibenden Spalten (also die nicht aufgeführt) verworfen, aber Sie können den Hyperparameter `remainder` auf einen beliebigen Transformer setzen (oder auf "passthrough"), wenn sie anders behandelt werden sollen.

Setzen Sie Scikit-Learn 0.19 oder älter ein, können Sie eine Third-Party-Bibliothek wie `sklearn-pandas` verwenden oder Ihren eigenen Transformer bauen, um die gleiche Funktionalität zu erhalten, wie sie der `ColumnTransformer` bietet. Alternativ nutzen Sie die Klasse `FeatureUnion`, die unterschiedliche Transformer anwenden und deren Ergebnisse verbinden kann. Aber Sie können nicht für jeden Transformer verschiedene Spalten angeben – sie werden alle auf die gesamten Daten angewendet. Es ist möglich, diese Beschränkung zu umgehen, indem Sie einen eigenen Transformer für die Spaltenauswahl nutzen (siehe dazu das Jupyter-Notebook).

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

68628.19819848922

```

In Ordnung, dieser RMSE ist besser als gar nichts, aber bestimmt nicht großartig: Der Wert `median_housing_values` liegt in den meisten Bezirken zwischen 120.000 und 265.000 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 *k-fache Kreuzvalidierung*. Der folgende Code 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) anstelle 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: [70194.33680785 66855.16363941 72432.58244769 70758.73896782
71115.88230639 75585.14172901 70262.86139133 70273.6325285
75366.87952553 71231.65726027]

Mittelwert: 71407.68766037929

Standardabweichung: 2439.4345041191004
```

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 71407, ± 2439 . 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: [66782.73843989 66960.118071    70347.95244419 74739.57052552
68031.13388938 71193.84183426 64969.63056405 68281.61137997
71552.91566558 67665.10082067]
```

Mittelwert: 69052.46136345083

Standardabweichung: 2731.674001798348

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 fast 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  
18603.515021376355  
>>> display_scores(forest_rmse_scores)  
Scores: [49519.80364233 47461.9115823 50029.02762854 52325.28068953  
        49308.39426421 53446.37892622 48634.8036574 47585.73832311  
        53490.10699751 50021.5852922 ]  
Mittelwert: 50182.303100336096  
Standardabweichung: 2097.0810550985693
```

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 zum 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 alle Modelle, mit denen 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 der Bibliothek `joblib` speichern, wobei Letztere große NumPy-Arrays effizienter serialisiert:

```
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`:

```
        return_train_score=True)

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 auf `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, es gibt 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_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=30, n_jobs=None, oob_score=False, random_state=None,
                      verbose=0, warm_start=False)
```



Falls `GridSearchCV` mit `refit=True` initialisiert wird (der Standardeinstellung), wird der

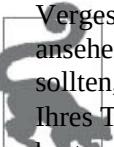
beste über Kreuzvalidierung gefundene Estimator noch einmal mit dem gesamten Trainingsdatensatz trainiert. Dies ist grundsätzlich eine gute Idee, da mehr Daten voraussichtlich die Genauigkeit erhöhen.

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)
...
63669.05791727153 {'max_features': 2, 'n_estimators': 3}
55627.16171305252 {'max_features': 2, 'n_estimators': 10}
53384.57867637289 {'max_features': 2, 'n_estimators': 30}
60965.99185930139 {'max_features': 4, 'n_estimators': 3}
52740.98248528835 {'max_features': 4, 'n_estimators': 10}
50377.344409590376 {'max_features': 4, 'n_estimators': 30}
58663.84733372485 {'max_features': 6, 'n_estimators': 3}
52006.15355973719 {'max_features': 6, 'n_estimators': 10}
50146.465964159885 {'max_features': 6, 'n_estimators': 30}
57869.25504027614 {'max_features': 8, 'n_estimators': 3}
51711.09443660957 {'max_features': 8, 'n_estimators': 10}
49682.25345942335 {'max_features': 8, 'n_estimators': 30}
62895.088889905004 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.14484390074 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.399594730654 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52725.01091081235 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.612956065226 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.51445842374 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

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 49682, was etwas besser als der zuvor mit den voreingestellten

Hyperparametern berechnete Wert ist (dieser betrug 50182). 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. den Hyperparameter `add_bedrooms_per_room` Ihres Transformers `CombinedAttributes Adder`). 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 durchzuprobieren, 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 1.000 Iterationen laufen lassen, werden für jeden Hyperparameter 1.000 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_encoder = full_pipeline.named_transformers_["cat"]  
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])  
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs  
>>> sorted(zip(feature_importances, attributes), reverse=True)  
[(0.3661589806181342, 'median_income'),  
(0.1647809935615905, 'INLAND'),  
(0.10879295677551573, 'pop_per_hhold'),  
(0.07334423551601242, 'longitude'),  
(0.0629090704826203, 'latitude'),  
(0.05641917918195401, 'rooms_per_hhold'),  
(0.05335107734767581, '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'),  
(0.002856474637320158, 'NEAR OCEAN'),  
(0.00196041559947807, 'NEAR BAY'),  
(6.028038672736599e-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. durch Hinzufügen von zusätzlichen Merkmalen oder Entfernen von überflüssigen, von Ausreißern und so weiter).

Evaluiere das System auf dem Testdatensatz

Haben Sie Ihre Modelle über eine gewisse Zeit optimiert, haben Sie ein System, das ausreichend gut funktioniert. Nun kann das endgültige Modell mit dem Testdatensatz evaluiert werden. 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()`), denn Sie wollen nicht an den Testdatensatz anpassen!) 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 47730.2
```

Manchmal wird solch eine Punktschätzung des Verallgemeinerungsfehlers nicht ausreichen, um Sie davon zu überzeugen, das Modell zu übernehmen. Was, wenn es nur 0,1% besser als das aktuell genutzte Modell ist? Vielleicht wollen Sie verstehen, wie exakt diese Schätzung ist. Dazu können Sie mit `scipy.stats.t.interval()` ein 95-%-Konfidenzintervall für den Verallgemeinerungsfehler berechnen.

```
>>> from scipy import stats

>>> confidence = 0.95

>>> squared_errors = (final_predictions - y_test) ** 2

>>> np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
...                           loc=squared_errors.mean(),
...                           scale=stats.sem(squared_errors)))

...
array([45685.10470776, 49691.25001878])
```

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 wo die Grenzen Ihres Systems sind), dokumentieren und ansprechendes Präsentationsmaterial mit klaren Visualisierungen und eingängigen Aussagen (z.B. »das mittlere Einkommen ist der beste Prädiktor des Immobilienpreises«) anbieten. Im Beispiel mit den kalifornischen Immobilienpreisen ist die abschließende Genauigkeit des Systems nicht besser als die Preisschätzungen der Experten, die oft um etwa 20% danebenliegen, aber es kann trotzdem hilfreich sein, es einzusetzen, insbesondere wenn besagte Experten damit Zeit für interessantere und produktivere Aufgaben verschafft bekommen.

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 (zum Beispiel den Code ordentlich machen, Dokumentation und Tests schreiben und so weiter). Dann können Sie Ihr Modell in Ihre Produktivumgebung deployen. Eine Möglichkeit dafür besteht darin, das trainierte Scikit-Learn-Modell zu sichern (beispielsweise mit `joblib`) – einschließlich der vollständigen Vorverarbeitungs- und Vorhersagepipeline –, dann dieses trainierte Modell in Ihrer Produktivumgebung zu laden und es durch den Aufruf der Methode `predict()` zu verwenden. Vielleicht wird das Modell ja auf einer Website genutzt: Der Anwender gibt Daten über einen neuen Bezirk ein und klickt auf den Button *Preis schätzen*. Dadurch wird eine Anfrage mit den Daten an den Webserver geschickt, der diese an Ihre Webanwendung weiterleitet, und Ihr Code ruft dort einfach die Methode `predict()` des Modells auf (Sie werden das Modell nicht bei jedem Einsatz laden wollen, sondern einmalig, wenn der Server startet). Alternativ können Sie das Modell auch in einem eigenen Webservice verpacken, den Ihre Webanwendung über eine REST-API²³ aufrufen kann (siehe Abbildung 2-17). Das erleichtert das Aktualisieren des Modells, ohne dass die Hauptanwendung unterbrochen werden muss. Auch wird das Skalieren so einfacher, da Sie so viele Webservices wie nötig starten und die Requests von Ihrer Webanwendung darauf per Load Balancing verteilen können. Zudem ist es so möglich, Ihre Webanwendung in einer beliebigen Programmiersprache zu entwickeln, statt auf Python angewiesen zu sein.

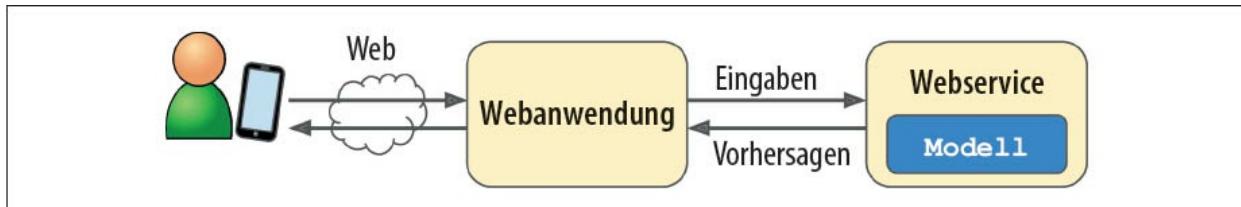


Abbildung 2-17: Ein Modell, das als Webservice deployt und von einer Webanwendung genutzt wird

Eine andere verbreitete Strategie besteht darin, Ihr Modell in die Cloud zu deployen – zum Beispiel auf die Google Cloud AI Platform (früher bekannt als Google Cloud ML Engine): Speichern Sie Ihr Modell einfach mithilfe von `joblib`, laden Sie es auf Google Cloud Storage (GCS), wechseln Sie zur Google Cloud AI Platform, erstellen Sie eine neue Modellversion und lassen Sie diese auf die GCS-Datei zeigen. Das ist alles! Sie erhalten so einen einfachen Webservice, der Ihnen Load Balancing und Skalieren abnimmt. Er erwartet JSON-Requests mit den Eingabedaten (zum Beispiel von einem Bezirk) und gibt JSON-Responses mit den Vorhersagen zurück. Sie können diesen Webservice dann auf Ihrer Website verwenden (oder welche Produktivumgebung Sie sonst so nutzen). Wie wir in [Kapitel 19](#) sehen werden, unterscheidet sich das Deployen von TensorFlow-Modellen auf AI Platform nicht sonderlich vom Deployen von Scikit-Learn-Modellen.

Aber mit dem Deployen ist die Geschichte noch nicht zu Ende. 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 entsteht häufig, da Modelle mit der Zeit dazu tendieren, zu »verrotten«: Die Welt dreht sich weiter, und wenn das Modell mit den Daten aus dem letzten Jahr trainiert wurde, passt es vielleicht nicht zu denen von heute.

Auch ein Modell, das dafür trainiert wurde, Bilder von Katzen und Hunden zu klassifizieren, muss eventuell regelmäßig nachtrainiert werden – nicht weil Katzen und Hunde über Nacht mutieren, sondern weil sich Kameras, Bildformate, Schärfe, Helligkeit oder Seitenverhältnisse ändern. Und vielleicht mögen die Menschen im nächsten Jahr lieber andere Rassen, oder sie entscheiden sich dazu, ihre Haustiere mit kleinen Hütchen zu verkleiden – wer weiß.

Sie müssen also die Leistung Ihres Modells überwachen. Aber wie tun Sie das? Nun, das kommt darauf an. In manchen Fällen kann sie aus Folgemetriken abgeleitet werden. Ist Ihr Modell beispielsweise Teil eines Empfehlungssystems und schlägt es Produkte vor, an denen die Benutzer interessiert sein könnten, ist es leicht, die Anzahl an empfohlenen Produkten zu überwachen, die jeden Tag verkauft werden. Fällt diese Zahl (verglichen mit den nicht empfohlenen Produkten), ist der Haupt verdächtige das Modell. Es kann daran liegen, dass die Datenpipeline defekt ist, oder vielleicht muss das Modell mit frischen Daten neu trainiert werden (wie wir gleich noch besprechen werden).

Aber es ist nicht immer möglich, die Leistung des Modells ohne menschliche Analyse zu bestimmen. Stellen Sie sich beispielsweise vor, Sie hätten ein Modell zur Bildklassifikation trainiert (siehe [Kapitel 3](#)), um defekte Produkte in der Herstellung zu erkennen. Wie können Sie gewarnt werden, wenn die Leistung des Modells abfällt, bevor Tausende von defekten Produkten an Ihre Kunden geliefert werden? Eine Möglichkeit ist, menschlichen Bewertern Beispiele aller der Bilder zu schicken, die das Modell klassifiziert (insbesondere der Bilder, bei denen sich das Modell nicht so sicher ist). Abhängig von der Aufgabe müssen die Bewerter eventuell Experten sein, aber manchmal sind Spezialisten gar nicht notwendig, und es bieten sich Arbeiter auf einer Plattform für Crowdsourcing an (zum Beispiel Amazon Mechanical Turk). Bei manchen Anwendungen kann es sich auch um die Anwender selbst handeln, die beispielsweise über

Umfragen oder umfunktionierte Captchas²⁴ Rückmeldung geben.

Auf jeden Fall müssen Sie ein Überwachungssystem (mit oder ohne menschliche Beurteiler) zum Evaluieren des Modells aufsetzen und auch alle relevanten Prozesse einrichten, um zu definieren, was bei Fehlern zu tun ist und wie man sich auf sie vorbereitet. Leider kann das ziemlich viel Arbeit bedeuten. Tatsächlich ist es meist sogar mehr Arbeit, als ein Modell zu bauen und zu trainieren.

Entwickeln sich die Daten weiter, werden Sie regelmäßig Ihre Datensätze aktualisieren und Ihr Modell neu trainieren müssen. Sie sollten den Prozess so weit wie möglich zu automatisieren versuchen. Hier ein paar Schritte, die Sie automatisieren können:

- Regelmäßig neue Daten einsammeln und labeln (zum Beispiel mithilfe menschlicher Beurteiler).
- Ein Skript schreiben, um das Modell zu trainieren und die Hyperparameter automatisch genauer anzupassen. Das Skript kann abhängig von Ihren Anforderungen automatisiert laufen – zum Beispiel jeden Tag oder jede Woche.
- Ein weiteres Skript schreiben, das sowohl das neue wie auch das alte Modell gegen den aktualisierten Testdatensatz evaluiert und das Modell in die Produktivumgebung deployt, wenn die Leistung nicht nachgelassen hat (und wenn sie nachlässt, stellen Sie sicher, dass Sie auch ermitteln, woran das liegt).

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. Sie könnten beispielsweise eine Warnung ausgeben lassen, wenn immer mehr Eingangsdaten ein Merkmal fehlt, wenn deren Mittelwert oder Standardabweichung zu weit vom Trainingsdatensatz wegdriftet oder wenn ein kategorisches Merkmal plötzlich neue Kategorien besitzt.

Und stellen Sie auch sicher, dass Sie Backups von jedem erstellten Modell haben und die Prozesse und Tools bereitstehen, um schnell zu einem früheren Modell zurückkehren zu können, falls das neue Modell aus irgendwelchen Gründen fehlerhaft arbeitet. Durch das Vorhandensein von Backups können Sie auch leicht neue Modelle mit bestehenden vergleichen. Und Sie sollten Backups jeder Version Ihrer Datensätze aufheben, sodass Sie zu einem älteren Datensatz zurückkehren können, wenn das neue einmal fehlerhaft ist (weil beispielsweise die neu hinzugefügten Daten nur noch aus Ausreißern bestehen). Mit den Backups Ihrer Datensätze können Sie außerdem jedes Modell gegen jeden früheren Datensatz evaluieren.



Sie können diverse Untermengen des Testdatensatzes erzeugen, um zu ermitteln, wie gut Ihr Modell mit bestimmten Teilen der Daten funktioniert. So kann es beispielsweise sinnvoll sein, eine Untermenge nur mit den aktuellsten Daten zu besitzen oder einen Testdatensatz für bestimmte Eingabearten (zum Beispiel Bezirke im Landesinneren im Gegensatz zu denen in Küstennähe). Damit erhalten Sie ein besseres Verständnis der Stärken und Schwächen Ihres Modells.

Wie Sie sehen, gehört zu Machine Learning ziemlich viel Infrastruktur – seien Sie also nicht überrascht, wenn Ihr erstes ML-Projekt viel Aufwand und Zeit erfordert, um es in den Produktivbetrieb zu bringen. Ist die Infrastruktur einmal eingerichtet, wird es zum Glück deutlich schneller werden, von einer Idee bis in die Produktion zu kommen.

Probieren Sie es aus!

Dieses Kapitel hat Ihnen hoffentlich einen Eindruck davon 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.

Wenn Sie es also ohnehin nicht bereits getan haben, ist es 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 Website für Wettbewerbe wie <http://kaggle.com/>: Dort finden Sie Datensätze zum Ausprobieren, klare Zielvorgaben, und Sie können Erfahrungen austauschen.

Viel Spaß!

Übungen

Die folgenden Aufgaben basieren alle auf dem Immobilien-Datensatz aus diesem Kapitel:

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-ml2>.

KAPITEL 3

Klassifikation

In [Kapitel 1](#) haben wir erwähnt, dass die am häufigsten zu findenden 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 70.000 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_openml  
>>> mnist = fetch_openml('mnist_784', version=1)  
>>> mnist.keys()  
  
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details',  
          'categories', 'url'])
```

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 70.000 Bilder, und 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:

```
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap="binary")
plt.axis("off")
plt.show()
```



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

```
>>> y[0]
'5'
```

Beachten Sie, dass das Label ein String ist. Die meisten ML-Algorithmen erwarten Zahlen, daher wollen wir `y` in einen Integer casten:

```
>>> y = y.astype(np.uint8)
```

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

Einen Moment noch! Sie sollten stets einen Testdatensatz erstellen und ihn vor dem genaueren Betrachten der Daten beiseitelegen. Der MNIST-Datensatz ist bereits in Trainingsdaten (die ersten 60.000 Bilder) und Testdaten (die letzten 10.000 Bilder) unterteilt:

```
x_train, x_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

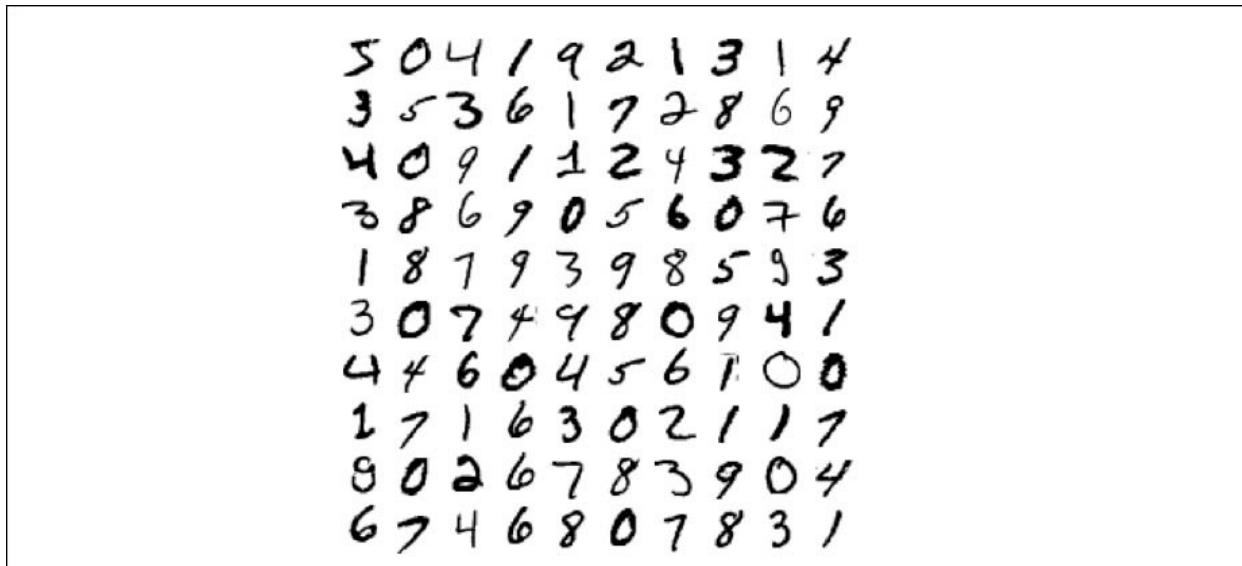


Abbildung 3-1: Ziffern aus dem MNIST-Datensatz

Die Trainingsdaten sind schon für uns gemischt, was gut ist, denn 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 ein paar 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

Trainieren eines binären Klassifikators

Für den Anfang werden wir die Aufgabe vereinfachen und lediglich versuchen, eine Ziffer zu erkennen – beispielsweise die Ziffer 5. Dieser »5-Detektor« ist ein Beispiel 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 das *Online-Learning* gut geeignet ist). Erstellen wir zunächst einen `SGDClassifier` und trainieren wir 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])
```

Der Klassifikator meint, dass dieses Bild eine 5 darstellt (`True`). Es sieht so aus, als läge er in diesem Fall richtig! 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. 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([0.96355, 0.93795, 0.95615])
```

Wow! Eine *Genauigkeit* über 93% (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([0.91125, 0.90855, 0.90915])
```

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 der 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 treffen, wir lassen das 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()` einsetzen:

```
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([[53057, 1522],  
       [1325, 4096]])
```

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*): 53.057 davon wurden korrekt als Nicht-5en klassifiziert (man nennt diese *richtig Negative*), die übrigen 1.522 wurden fälschlicherweise als 5en klassifiziert (*falsch Positive*). Die zweite Zeile betrachtet die Bilder mit 5en (die *positive Kategorie*): 1.325 Bilder wurden fälschlicherweise als Nicht-5en vorhergesagt (*falsch Negative*), die restlichen 4.096 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:

```
>>> y_train_perfect_predictions = y_train_5 # als ob wir perfekt wären  
  
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)  
  
array([[54579,     0],  
       [     0, 5421]])
```

Die Konfusionsmatrix gibt uns eine Menge an Informationen, aber manchmal wünscht man sich ein etwas kompakteres Qualitätsmaß. Ein interessantes Maß ist die Genauigkeit der positiven Vorhersagen; dies nennt man auch die *Relevanz* (engl. Precision) des Klassifikators (siehe [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 (siehe 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.

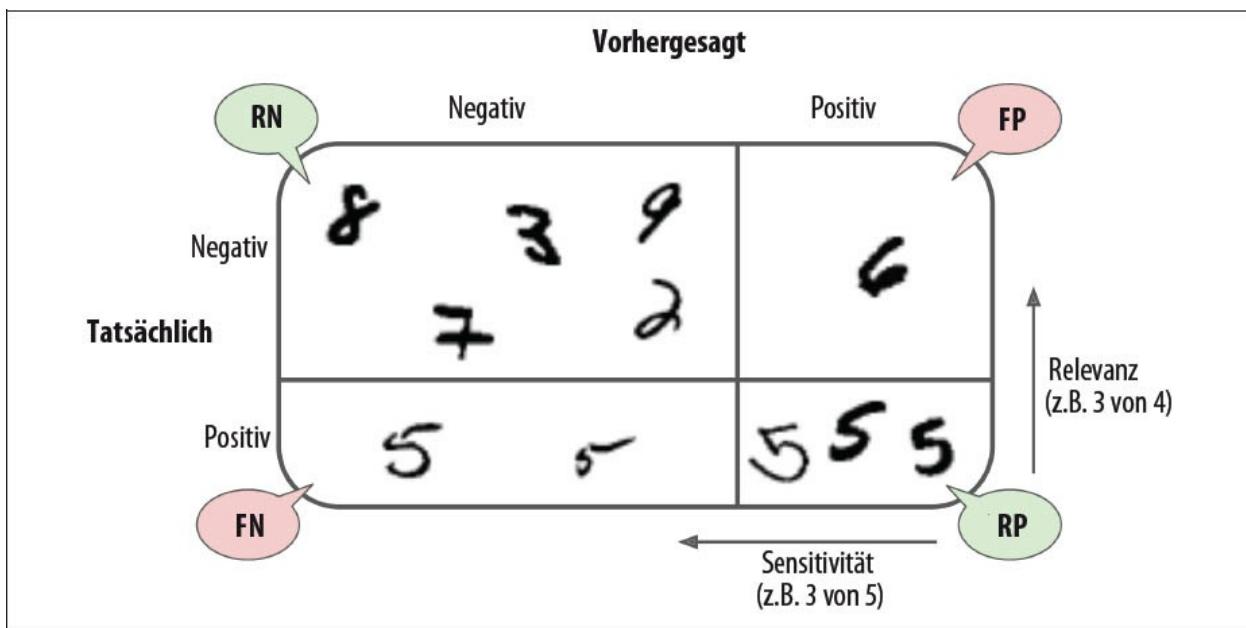


Abbildung 3-2: Eine illustrierte Konfusionsmatrix

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) # == 4096 / (4096 + 1522)
0.7290850836596654
>>> recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
```

0.7555801512636044

Damit glänzt Ihr 5-Detektor nicht mehr so stark, wie es die Genauigkeit zunächst vermuten ließ. Wenn er erklärt, dass ein Bild eine 5 enthält, liegt er nur in 72,9% der Fälle richtig. Außerdem findet er nur 75,6% 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 (siehe [Formel 3-3](#)). Während der gewöhnliche Mittelwert alle Werte gleich behandelt, erhalten beim harmonischen Mittelwert niedrigere Mittelwerte ein weitaus 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

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

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)  
0.7420962043663375
```

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 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).

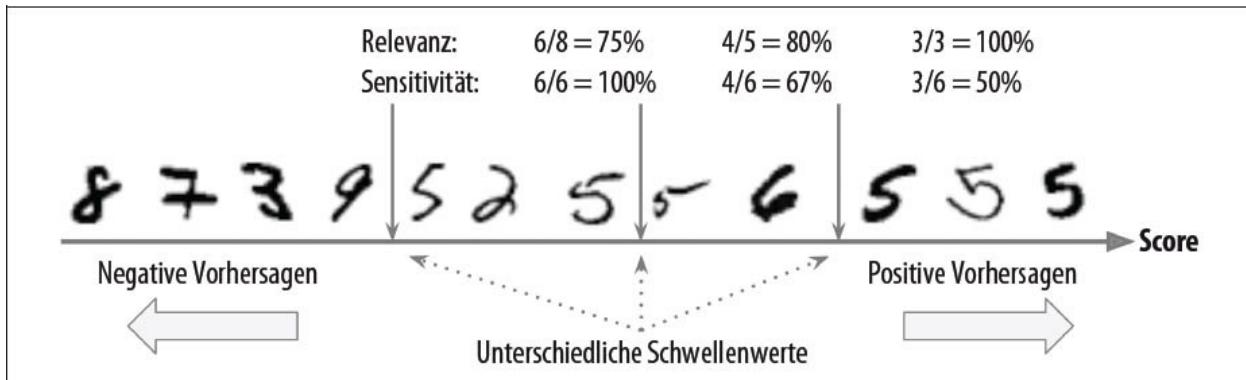


Abbildung 3-3: Bei dieser Wechselbeziehung zwischen Relevanz und Sensitivität werden Bilder aufgrund ihres Klassifikationsscores bewertet – solche oberhalb des gewählten Schwellenwerts betrachtet man als positiv. Je höher der Schwellenwert ist, desto geringer ist die Sensitivität, aber desto höher ist (im Allgemeinen) auch die 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([2412.53175101])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True])
```

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 = 8000
>>> y_some_digit_pred = (y_scores > threshold)
```

```
>>> y_some_digit_pred  
array([False])
```

Das 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 8000 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 (siehe [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")  
    [...] # Schwellenwert hervorheben und Legende, Achsbeschriftung  
          # und Raster ergänzen  
  
    plot_precision_recall_vs_threshold(precisions, recalls, thresholds)  
    plt.show()
```

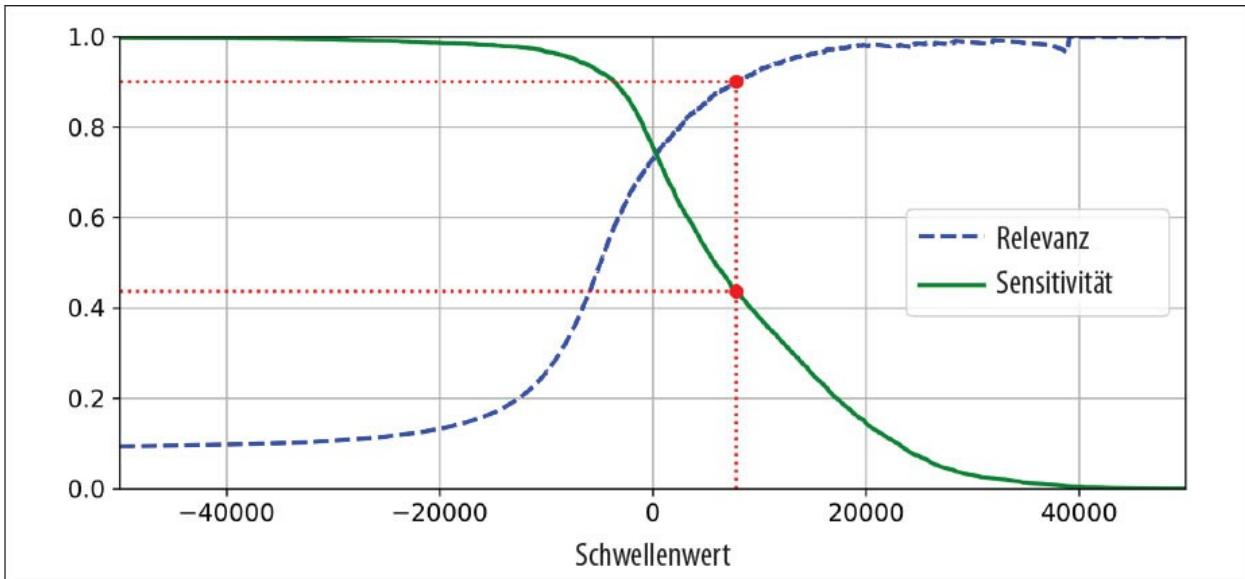


Abbildung 3-4: Relevanz und Sensitivität über dem Schwellenwert zur Entscheidung



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.

Zur Auswahl eines Kompromisses zwischen Relevanz und Sensitivität können Sie auch die Relevanz gegen die Sensitivität plotten, wie in Abbildung 3-5 gezeigt (hier ist der gleiche Schwellenwert wie zuvor hervorgehoben).

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.

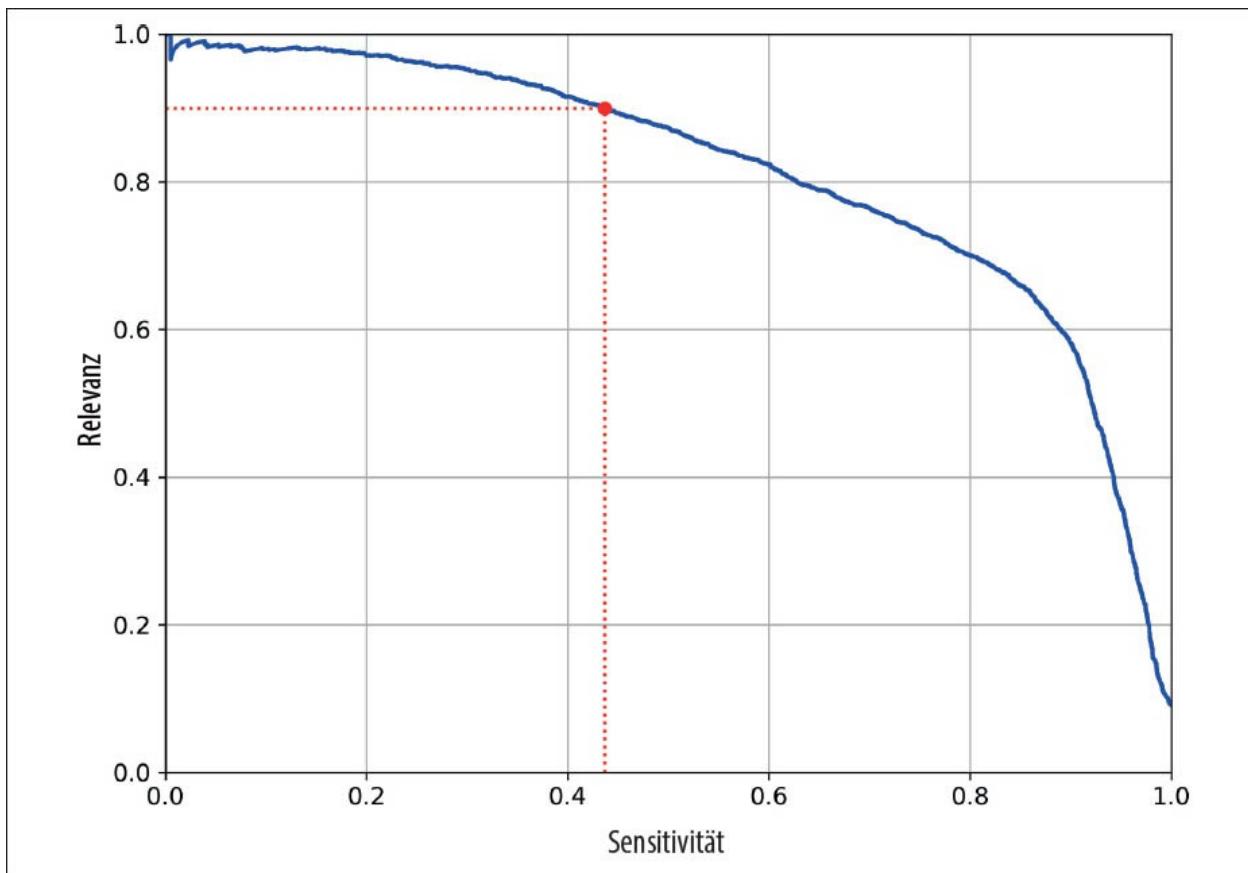


Abbildung 3-5: Relevanz gegen Sensitivität

Nehmen wir an, Sie möchten eine Relevanz von 90% erhalten. Sie schauen im ersten Diagramm nach und finden heraus, dass Sie einen Schwellenwert von etwa 8000 benötigen. Um genauer zu sein, können Sie nach dem niedrigsten Schwellenwert suchen, der Ihnen mindestens 90% Relevanz liefert (`np.argmax()` wird Ihnen den ersten Index des maximalen Werts liefern, was in diesem Fall dem ersten True-Wert entspricht):

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)] # ~7816
```

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 >= threshold_90_precision)
```

Überprüfen wir Relevanz und Sensitivität dieser Vorhersagen:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.9000380083618396
>>> recall_score(y_train_5, y_train_pred_90)
```

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 *Richtignegativ-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 die 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--') # diagonal gestrichelt
    [...] # Achsenbeschriftungen und Raster hinzufügen

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

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).

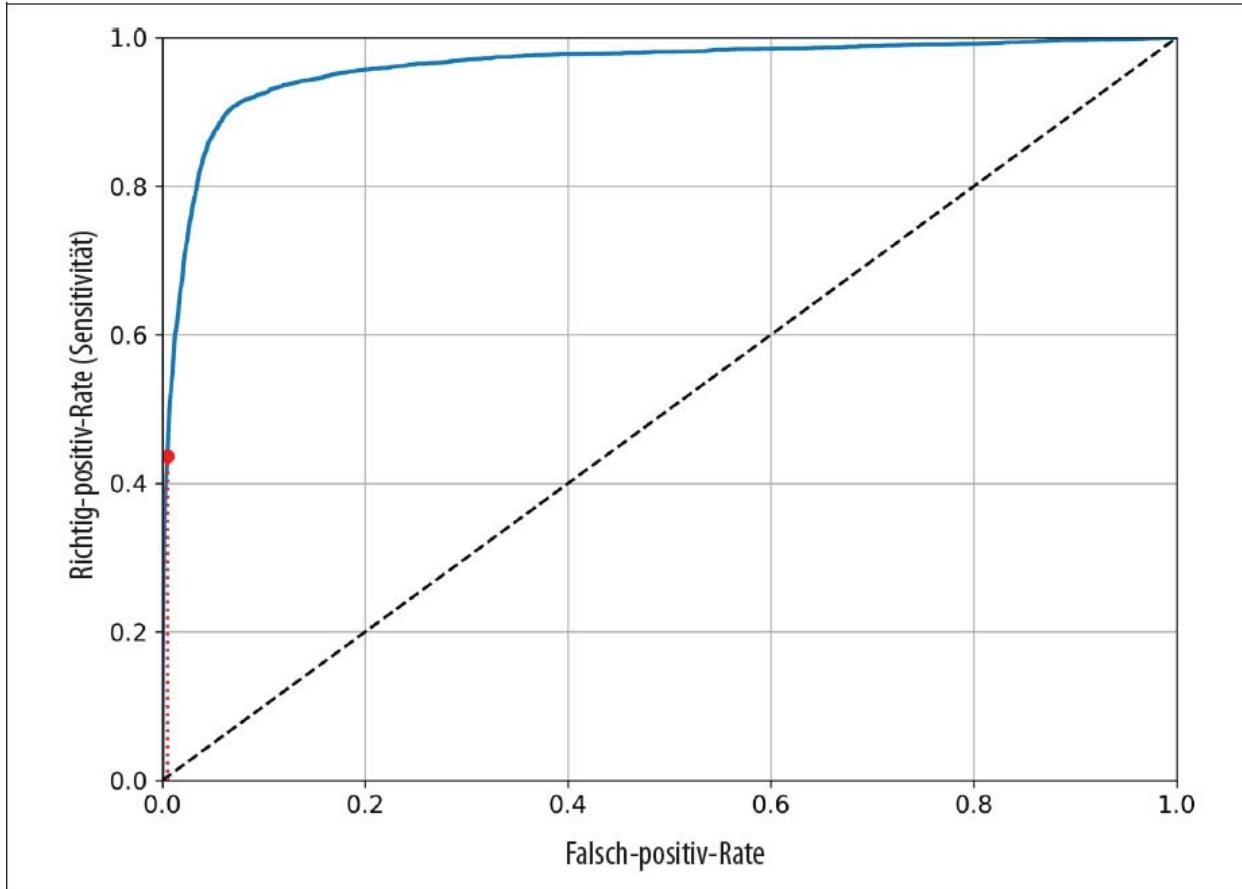
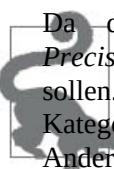


Abbildung 3-6: Diese ROC-Kurve plottet die FPR gegen die TPR für alle möglichen Schwellenwerte – der rote Punkt steht für das gewählte Verhältnis (bei 43,68% Sensitivität).

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)
0.9611778893101814
```

 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 linken 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 die eine oder die andere – oder beide. Die Methode `predict_proba()` liefert ein Array mit einer Zeile pro Datenpunkt und einer Spalte pro Kategorie. Es enthält die Wahrscheinlichkeit, 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")
```

Die Funktion `roc_curve()` erwartet Labels und Scores, aber statt Scores können Sie ihr auch Wahrscheinlichkeiten von Kategorien mitgeben. Nutzen wir die Wahrscheinlichkeit der positiven Kategorie als Score:

```
y_scores_forest = y_probas_forest[:, 1] # score = Wahrscheinlichkeit
                                         # der positiven Kategorie

fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5, y_scores_forest)
```

Nun sind Sie so weit, die ROC-Kurve zu plotten. Dabei ist es hilfreich, zum Vergleich auch die erste ROC-Kurve zu plotten (siehe [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()
```

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)
```

```
0.9983436731328145
```

Versuchen Sie, Relevanz und Sensitivität zu berechnen: Sie sollten eine Relevanz von 99,0% und eine Sensitivität von 86,6% 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.

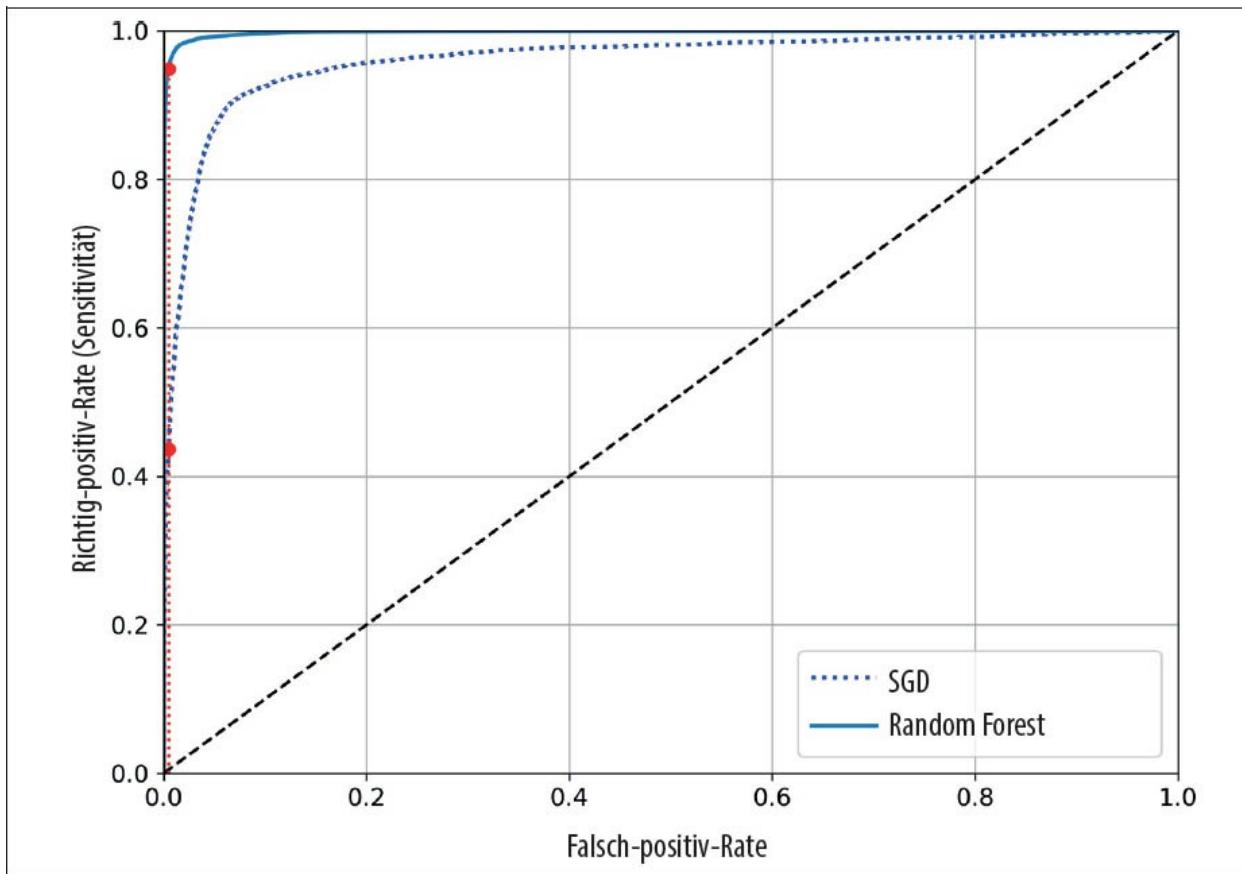


Abbildung 3-7: Vergleich zweier ROC-Kurven: Der Random-Forest-Klassifikator ist besser als der SGD-Klassifikator, weil dessen ROC-Kurve viel näher an der oberen linken Ecke ist und eine größere AUC besitzt.

Klassifikatoren mit mehreren Kategorien

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

Einige Algorithmen (wie SGD-Klassifikatoren, Random-Forest-Klassifikatoren oder naive

Bayes-Klassifikatoren) sind in der Lage, direkt mehrere Kategorien zu berücksichtigen. Andere (wie logistische Regression oder Support-Vector-Machine-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 Entscheidungsscore und wählen die Kategorie aus, deren Klassifikator den höchsten Score lieferte. Dies bezeichnet man als *One-versus-the-Rest*-(*OvR*-)Strategie (oder *One-versus-All*).

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 OvR vorzuziehen.

Scikit-Learn erkennt, wenn Sie einen binären Klassifikationsalgorithmus für eine Aufgabe mit mehreren Kategorien einsetzen, und verwendet abhängig vom Algorithmus automatisch OvR oder OvO. Probieren wir das mit einem SVM-Klassifikator (siehe [Kapitel 5](#)) und der Klasse `sklearn.svm.SVC`:

```
>>> from sklearn.svm import SVC  
  
>>> svm_clf = SVC()  
  
>>> svm_clf.fit(X_train, y_train) # y_train, nicht y_train_5  
  
>>> svm_clf.predict([some_digit])  
  
array([5], dtype=uint8)
```

Das war leicht! Der Code trainiert den SVC auf den Trainingsdaten mit den ursprünglichen Zielkategorien von 0 bis 9 (`y_train`) anstatt mit den Zielkategorien 5-gegen-den-Rest (`y_train_5`). Anschließend wird eine Vorhersage getroffen (in diesem Fall eine richtige). Im Hintergrund nutzt Scikit-Learn die OvO-Strategie: Es trainiert 45 binäre Klassifikatoren, ermittelt deren Entscheidungswerte für das Bild und wählt die Kategorie aus, die die meisten

Duelle gewonnen hat.

Rufen Sie die Methode `decision_function()` auf, werden Sie sehen, dass zehn Scores pro Instanz zurückgegeben werden (statt nur einer). Das ist ein Score pro Kategorie:

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])  
  
>>> some_digit_scores  
  
array([[ 2.92492871,  7.02307409,  3.93648529,   0.90117363,  5.96945908,  
         9.5          ,  1.90718593,  8.02755089, -0.13202708,  4.94216947]])
```

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

```
>>> np.argmax(some_digit_scores)  
  
5  
  
>>> svm_clf.classes_  
  
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)  
  
>>> svm_clf.classes_[5]  
  
5
```

- 💡 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 das One-versus-the-Rest-Verfahren zu verwenden, nutzen Sie dazu die Klassen `OneVsOneClassifier` bzw. `OneVsRestClassifier`. Erstellen Sie einfach eine Instanz, der Sie im Konstruktor einen Klassifikator übergeben (es muss nicht einmal ein binärer Klassifikator sein). Der folgende Code erzeugt beispielsweise einen Klassifikator mit mehreren Kategorien mithilfe der OvR-Strategie und einem SVC:

```
>>> from sklearn.multiclass import OneVsRestClassifier  
  
>>> ovr_clf = OneVsRestClassifier(SVC())  
  
>>> ovr_clf.fit(X_train, y_train)  
  
>>> ovr_clf.predict([some_digit])  
  
array([5], dtype=uint8)  
  
>>> len(ovr_clf.estimators_)
```

Einen `SGDClassifier` (oder einen `RandomForestClassifier`) zu verwenden, ist genauso einfach:

```
>>> sgd_clf.fit(X_train, y_train)
>>> sgd_clf.predict([some_digit])
array([5], dtype=uint8)
```

Diesmal musste Scikit-Learn weder OvR noch OvO anwenden, da SGD-Klassifikatoren Datenpunkte direkt mehreren Kategorien zuordnen können. Die Methode `decision_function()` liefert nun einen Wert pro Kategorie zurück. Schauen wir uns den Score an, den der SGD-Klassifikator jeder Kategorie zugewiesen hat:

```
>>> sgd_clf.decision_function([some_digit])
array([[ -15955.22628, -38080.96296, -13326.66695,    573.52692, -17680.68466,
        2412.53175, -25526.86498, -12290.15705, -7946.05205, -10631.35889]])
```

Sie sehen auch, dass sich der Klassifikator seiner Vorhersage sehr sicher ist: So gut wie alle Scores sind deutlich negativ, aber die Kategorie 5 besitzt einen Score von 2412,5. Bei Kategorie 3 ist sich das Modell nicht sicher, da sie einen Score von 573,5 hat. Diese Klassifikatoren sollten wir nun natürlich auch auswerten. Wie gewöhnlich setzen wir die Kreuzvalidierung ein. Beurteilen wir die Genauigkeit des `SGDClassifier` mit der Funktion `cross_val_score()`:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.8489802, 0.87129356, 0.86988048])
```

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 89%:

```
>>> 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([0.89707059, 0.8960948, 0.90693604])
```

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. 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)

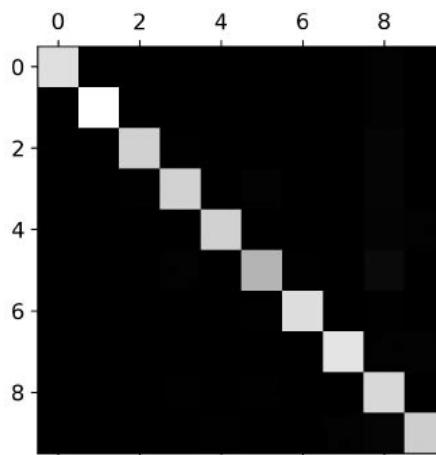
>>> conf_mx

array([[5578,      0,     22,      7,      8,     45,     35,      5,    222,      1],
       [  0,  6410,     35,     26,      4,     44,      4,      8,   198,     13],
       [ 28,     27,  5232,    100,     74,     27,     68,     37,   354,     11],
       [ 23,     18,    115,  5254,      2,    209,     26,     38,   373,     73],
       [ 11,     14,     45,     12,  5219,     11,     33,     26,   299,    172],
       [ 26,     16,     31,    173,     54,  4484,     76,     14,   482,     65],
       [ 31,     17,     45,      2,     42,     98,  5556,      3,   123,      1],
       [ 20,     10,     53,     27,     50,     13,      3,  5696,    173,    220],
       [ 17,     64,     47,    91,      3,    125,     24,     11,  5421,     48],
       [ 24,     18,     29,     67,   116,     39,      1,    174,    329,  5152]])
```

Dies ist 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()
```



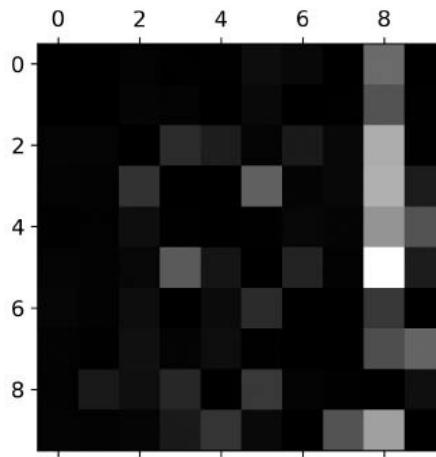
Diese Konfusionsmatrix sieht recht gut aus, da die meisten Bilder auf der Hauptdiagonalen liegen. Dies bedeutet, dass sie korrekt zugeordnet wurden. Die 5en sehen etwas dunkler als die anderen Ziffern aus. Dies könnte bedeuten, dass es weniger Bilder von 5en im Datensatz gibt oder dass der Klassifikator 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 statt 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 von Fehler der Klassifikator begeht. Wie gesagt, die Zeilen stehen für die tatsächlichen Kategorien und die Spalten für die vorhergesagten. Die Spalte der Kategorie 8 ist recht hell, es wurden also recht viele Bilder fälschlicherweise als 8 zugeordnet. Aber die Zeile für Kategorie 8 ist gar nicht so schlecht, was Ihnen zeigt, dass echte 8er im Allgemeinen korrekt als 8er klassifiziert werden. Wie Sie sehen, muss die Konfusionsmatrix nicht notwendigerweise symmetrisch sein. Sie können auch erkennen, dass 3er und 5er häufig verwechselt werden (in beide Richtungen).

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 falschen 8er konzentrieren sollte. Sie könnten also beispielsweise versuchen, zusätzliche Trainingsdaten für Ziffern zu sammeln, die wie 8er aussehen (es aber nicht sind), sodass der Klassifikator lernen kann, sie von den echten 8ern zu unterscheiden. Oder Sie probieren, neue Merkmale 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. Wir könnten etwa 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))

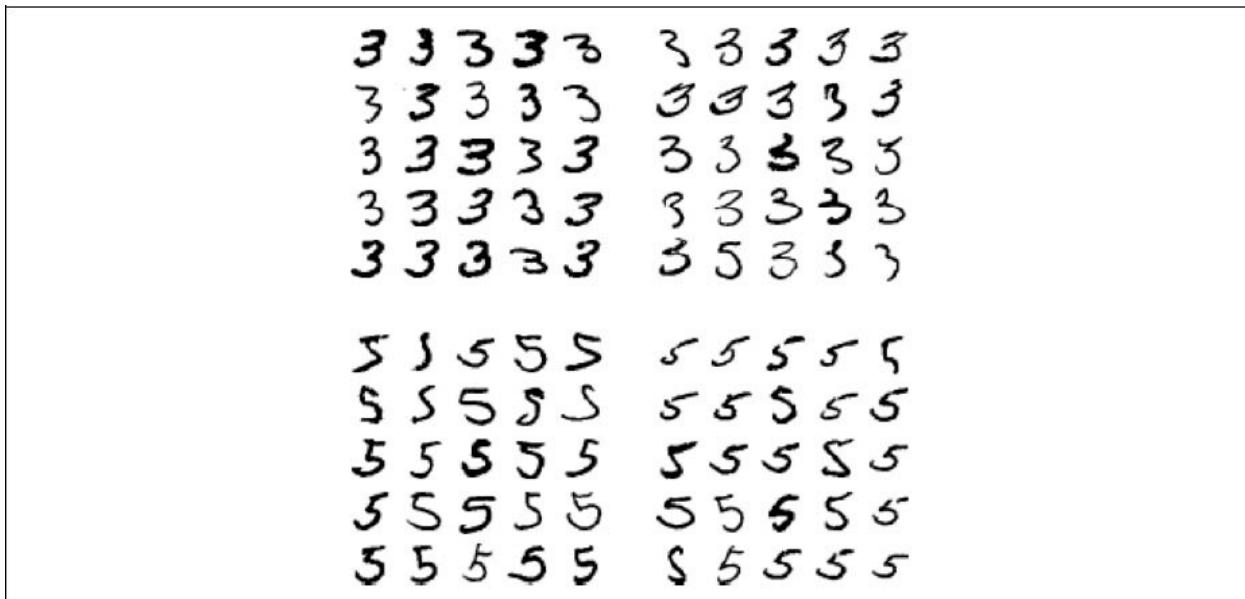
```

```

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 5×5 -Blöcke auf der linken Seite zeigen als 3en klassifizierte Ziffern, und die zwei 5×5 -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 ersten Zeile und zweiten Spalte wirklich wie eine schlecht geschriebene 3 aus). Es ist schwer nachzuvollziehen, warum der Klassifikator diese Fehler gemacht hat.³ Der Grund ist, dass wir einen einfachen SGDClas sifier 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, die die obere Linie mit dem unteren Bogen verbindet. Wenn Sie eine 3 zeichnen und diese Verbindungslinie 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 Tag 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 Tags 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]])
```

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")
0.976410265560605
```

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 das zu erreichen, setzen Sie im obigen Codebeispiel `average="weighted"`.⁴

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 verrauschtes 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 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

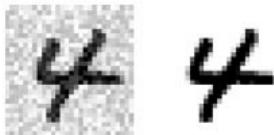
noise = rnd.randint(0, 100, (len(X_test), 784))

X_test_mod = X_test + noise

y_train_mod = y_train

y_test_mod = y_test
```

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



Auf der linken Seite ist das verrauchte 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 durch die Klassifikationsverfahren. Sie wissen nun, 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.

Übungen

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 jede Richtung (links, rechts, oben, unten) um ein Pixel verschieben lässt.⁵ Erstellen Sie anschließend aus sämtlichen Bildern im Trainingsdatensatz vier verschobene Kopien (eine pro Richtung) und fügen Sie diese den Trainingsdaten hinzu. Trainieren Sie dann 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 auszuprobieren, 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 (<https://homl.info/spamassassin>) 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« enthielten, 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« und alle Zahlen durch »NUMBER« ersetzen 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-ml2>.

Trainieren von Modellen

Bisher haben wir Modelle zum Machine Learning und deren Trainingsalgorithmen mehr oder weniger als Black Box 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 polynomiale 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 ein 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 und multiplizieren lassen, wie man sie invertiert 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 Onlinematerialien (<https://github.com/ageron/handson-ml2>) durch. Diejenigen unter Ihnen 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 . Wert des Merkmals.
- θ_j ist der j . Modellparameter (inklusive des Bias-Terms θ_0 und der Gewichte 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)

$$\hat{y} = h_{\theta}(\mathbf{X}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

- $\boldsymbol{\theta}$ ist der *Parametervektor* des Modells mit Bias-Term θ_0 und den Gewichten der Merkmale θ_1 bis θ_n .
- \mathbf{x} ist der *Merkmalsvektor* eines Datenpunkts mit den Werten x_0 bis x_n , wobei x_0 stets 1 beträgt.
- $\boldsymbol{\theta} \cdot \mathbf{x}$ ist das Skalarprodukt der Vektoren $\boldsymbol{\theta}$ und \mathbf{x} , was natürlich $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$ entspricht.
- h_{θ} ist die Hypothesenfunktion unter Verwendung der Modellparameter $\boldsymbol{\theta}$.



Beim Machine Learning werden Vektoren oft als Spaltenvektoren repräsentiert, also als zweidimensionale Arrays mit einer einzelnen Spalte. Handelt es sich bei θ und x um Spaltenvektoren, ist die Vorhersage $\hat{y} = \theta^T x$, wobei es sich bei θ^T um die *Transponierte* von θ handelt (ein Zeilen- statt eines Spaltenvektors) und $\theta^T x$ die Matrixmultiplikation von θ^T und x ist. Das ist natürlich die gleiche Vorhersage, nur dass sie nun als Matrix mit einer Zelle statt als Skalarwert dargestellt wird. In diesem Buch werde ich diese Notation nutzen, um einen Wechsel zwischen Skalarprodukt und Matrixmultiplikationen zu vermeiden.

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

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

Ein Großteil der Notation wurde bereits in [Kapitel 2](#) vorgestellt (siehe »Schreibweisen« auf [Seite 42](#)). 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. Um die Notation zu vereinfachen, werden wir im Folgenden einfach nur $\text{MSE}(\theta)$ anstelle von $\text{MSE}(X, h_\theta)$ 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](#)).

Formel 4-4: Normalengleichung

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

- $\hat{\theta}$ ist der Wert von θ , für den die Kostenfunktion minimal wird.
- 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)

```

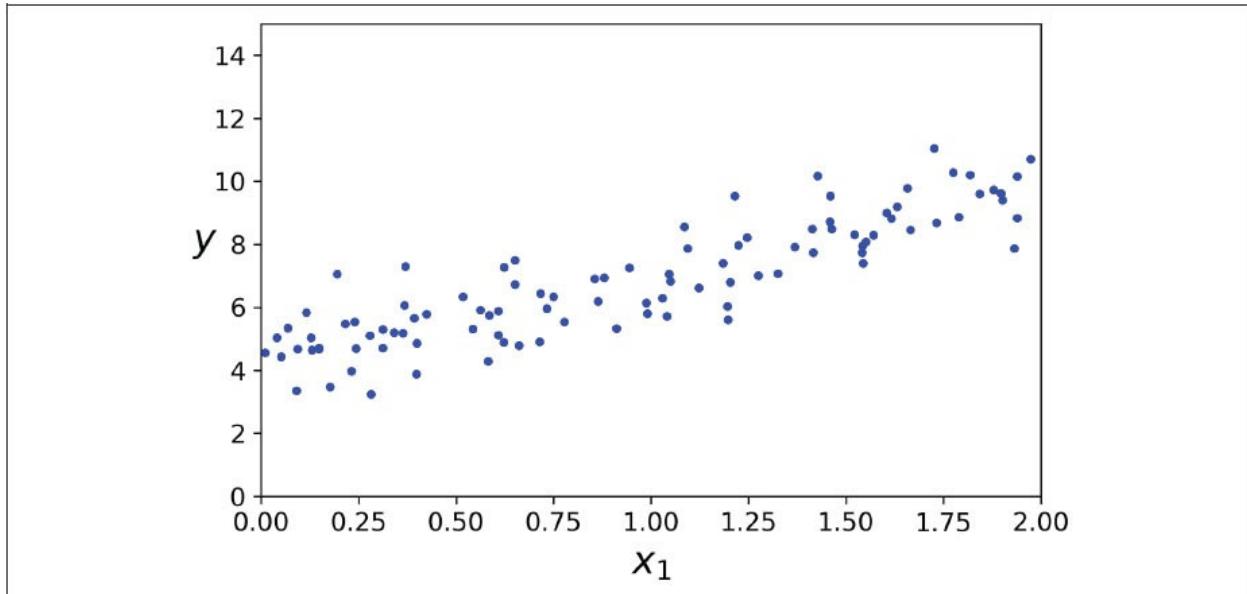


Abbildung 4-1: Zufällig generierter linearer Datensatz

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  $x_0 = 1$  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  $x_0 = 1$  zu jedem Datenpunkt hinzu
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[4.21509616],
       [9.75532293]])

```

Anschließend 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()

```

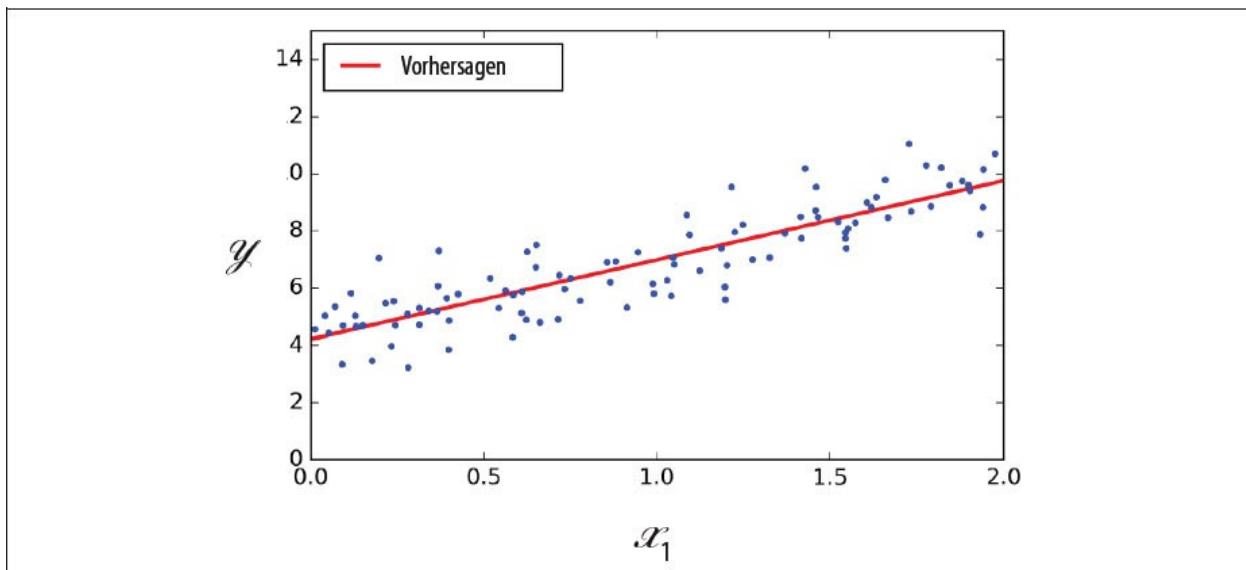


Abbildung 4-2: Vorhersagen des linearen Regressionsmodells

Das Durchführen der linearen Regression mit Scikit-Learn ist einfach:²

```

>>> 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]])

```

Die Klasse `LinearRegression` basiert auf der Funktion `scipy.linalg.lstsq()` (der Name steht für »Least Squares«), die Sie auch direkt aufrufen können:

```

>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)

>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])

```

Diese Funktion berechnet $\hat{\theta} = \mathbf{X}^+ \mathbf{y}$, wobei es sich bei \mathbf{X}^+ um die *Pseudoinverse* von \mathbf{X} handelt (genauer gesagt die Moore-Penrose-Inverse). Sie können `np.linalg.pinv()` verwenden, um sie direkt zu berechnen:

```

>>> np.linalg.pinv(X_b).dot(y)
array([[4.21509616],
       [2.77011339]])

```

Die Pseudoinverse selbst wird mit einer Standard-Matrixfaktorisierungstechnik namens *Singulärwertzerlegung* (SWZ; Singular Value Decomposition, SVD) berechnet, die die Matrix \mathbf{X} mit dem Trainingsdatensatz in die Matrixmultiplikation der drei Matrizen $\mathbf{U}\Sigma\mathbf{V}^T$ zerlegt (siehe `numpy.linalg.svd()`). Die Pseudoinverse wird so berechnet: $\mathbf{X}^+ = \mathbf{V}\Sigma^+\mathbf{U}^T$. Um die Matrix Σ^+ zu berechnen, nimmt der Algorithmus Σ und setzt alle Werte auf null, die kleiner als ein kleiner Schwellenwert sind, dann ersetzt er alle Nicht-null-Werte durch ihr Inverses und transponiert schließlich die so entstandene Matrix. Dieses Vorgehen ist effizienter als das Berechnen der Normalengleichung, zudem kommt sie mit Randfällen besser zurecht: Die Normalengleichung funktioniert eventuell nicht, wenn die Matrix $\mathbf{X}^T\mathbf{X}$ nicht invertierbar (also singulär) ist, wenn beispielsweise $m < n$ oder wenn Features redundant sind. Aber die Pseudoinverse ist immer definiert.

Komplexität der Berechnung

Die Normalengleichung berechnet die Inversion von $\mathbf{X}^T \cdot \mathbf{X}$, eine $(n+1) \times (n+1)$ -Matrix (wobei n die Anzahl Merkmale ist). Die *Komplexität der Berechnung* dieser Matrizeninversion beträgt typischerweise etwa $O(n^{2.4})$ bis $O(n^3)$ (abhängig von der Implementierung). Anders ausgedrückt,

wenn Sie die Anzahl Merkmale verdoppeln, erhöht sich die Rechenzeit etwa um den Faktor $2^{2,4} = 5,3$ bis $2^3 = 8$.

Der SVD-Ansatz der Klasse `LinearRegression` von Scikit-Learn liegt bei $O(n^2)$. Verdoppeln Sie die Anzahl an Merkmalen, multiplizieren Sie die Rechenzeit ungefähr mit 4.

- ☞ Sowohl die Normalengleichung wie auch der SVD-Ansatz werden sehr langsam, wenn die Anzahl Merkmale groß wird (z.B. 100.000). Andererseits sind beide in Bezug auf die Anzahl an Instanzen im Trainingsdatensatz linear ($O(m)$), sodass sie große Trainingsdatensätze effizient verarbeiten können, sofern diese in den Speicher passen.

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 auf die Anzahl Merkmale linear. Anders ausgedrückt, das Treffen einer Vorhersage dauert 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, in 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 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.

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](#)).

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](#)).

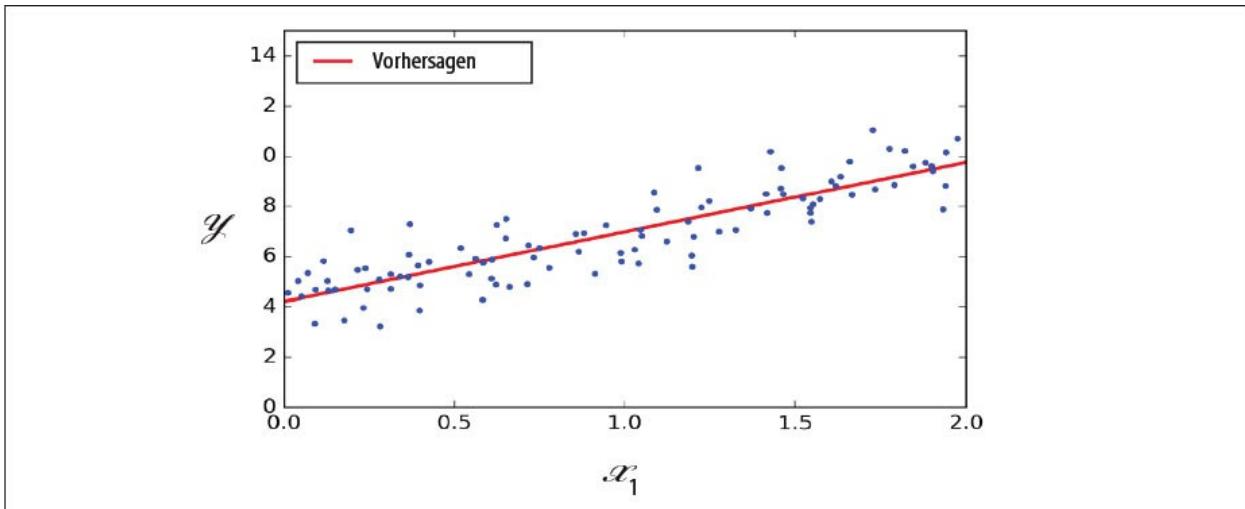


Abbildung 4-3: In dieser Darstellung des Gradientenverfahrens werden die Modellparameter mit Zufallswerten initialisiert und wiederholt angepasst, um die Kostenfunktion zu minimieren – die Größe der Lernschritte ist proportional zur Steigung der Kostenfunktion, sodass die Schritte nach und nach kleiner werden, wenn die Parameter auf das Minimum zusteuern.

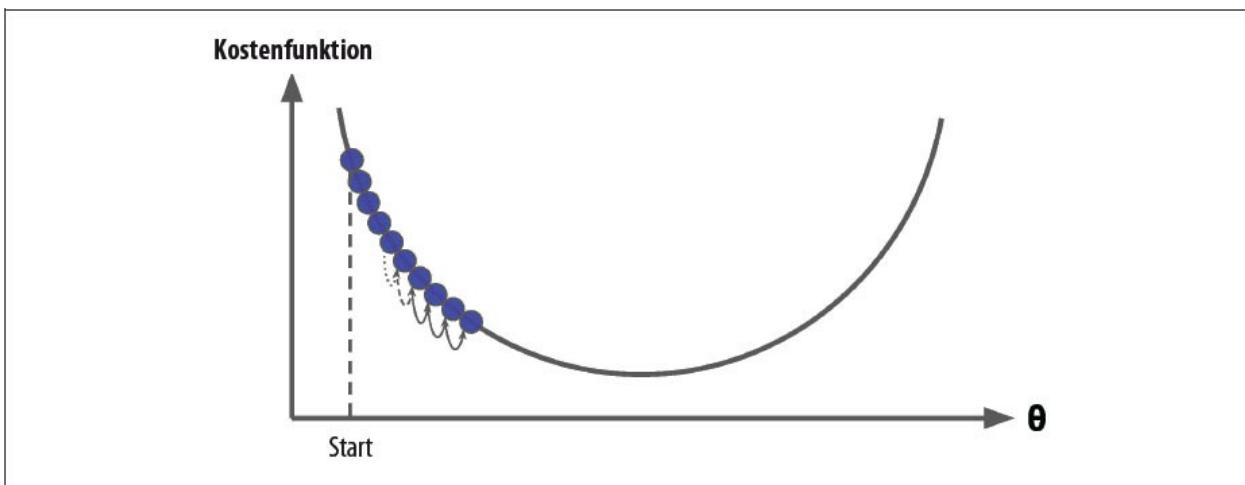


Abbildung 4-4: Die Lernrate ist zu gering.

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).

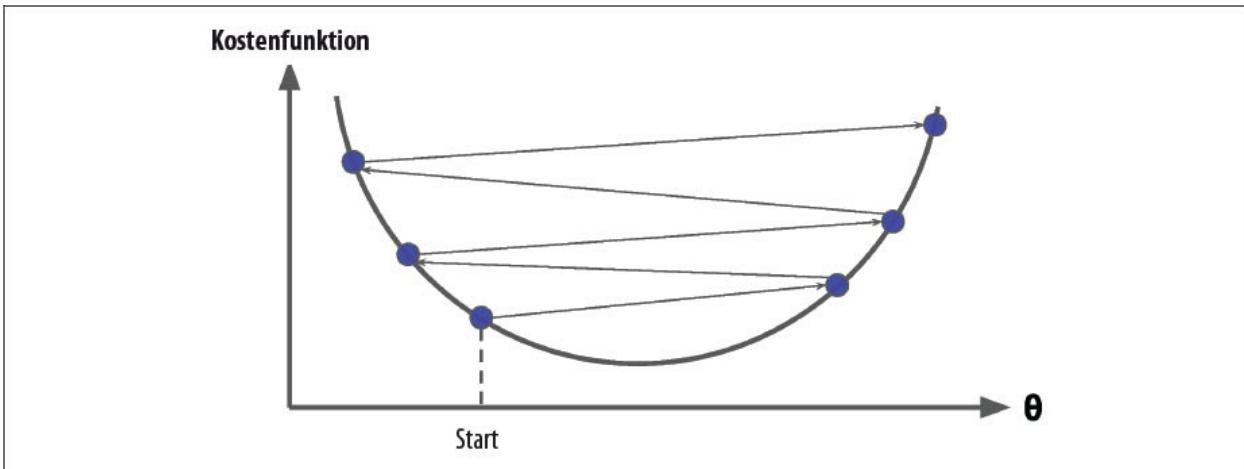


Abbildung 4-5: Die Lernrate ist zu groß.

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 erschweren. [Abbildung 4-6](#) zeigt die zwei wichtigsten Herausforderungen beim Gradientenverfahren: Wenn die zufällige Initialisierung den Algorithmus auf der linken Seite startet, wird er bei einem *lokalen 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.

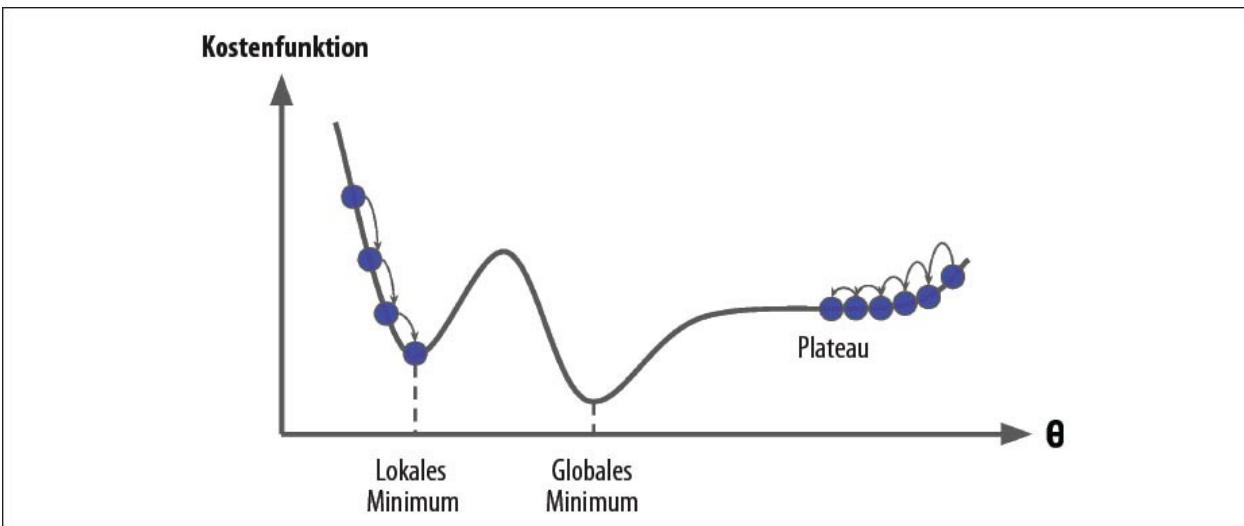


Abbildung 4-6: Fallstricke beim Gradientenverfahren

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.³ 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. Sind die Merkmale sehr unterschiedlich skaliert, 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).⁴

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.

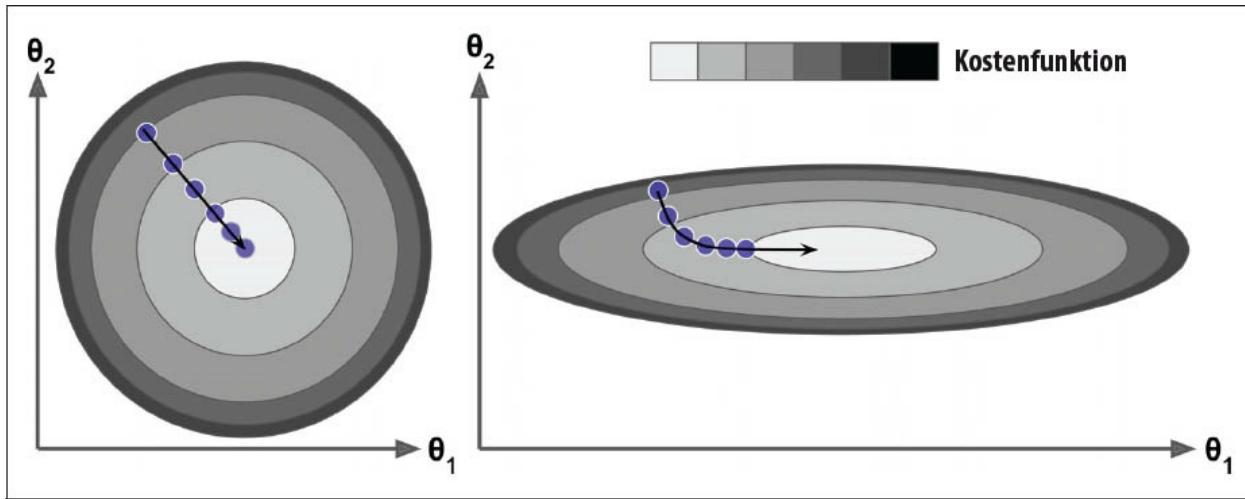


Abbildung 4-7: Gradientenverfahren mit (links) und ohne (rechts) Skalierung von Merkmalen



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 einer Nadel zu suchen, ist viel komplizierter als in drei Dimensionen. Da die Kostenfunktion konvex ist, liegt die Nadel im Fall 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, Sie müssen 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 $\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta})$ schreiben lässt.

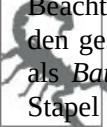
Formel 4-5: Partielle Ableitung der Kostenfunktion

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

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

Formel 4-6: Gradientenvektor der Kostenfunktion

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

 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 (*vollständiges Gradientenverfahren* wäre eigentlich ein besserer Name) 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 von Merkmalen ist mit dem Gradientenverfahren sehr viel schneller als mit der Normalengleichung oder der SVD-Zerlegung.

Sobald Sie den Gradientenvektor ermittelt haben, der bergauf weist, gehen Sie einfach in die entgegengesetzte Richtung, um sich bergab zu bewegen. Dazu müssen Sie $\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$ von $\boldsymbol{\theta}$ abziehen. An dieser Stelle kommt die Lernrate η ins Spiel:⁵ Multiplizieren Sie den Gradientenvektor mit η , um die Größe des Schritts bergab zu ermitteln ([Formel 4-7](#)).

Formel 4-7: Schritt im Gradientenverfahren

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

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).

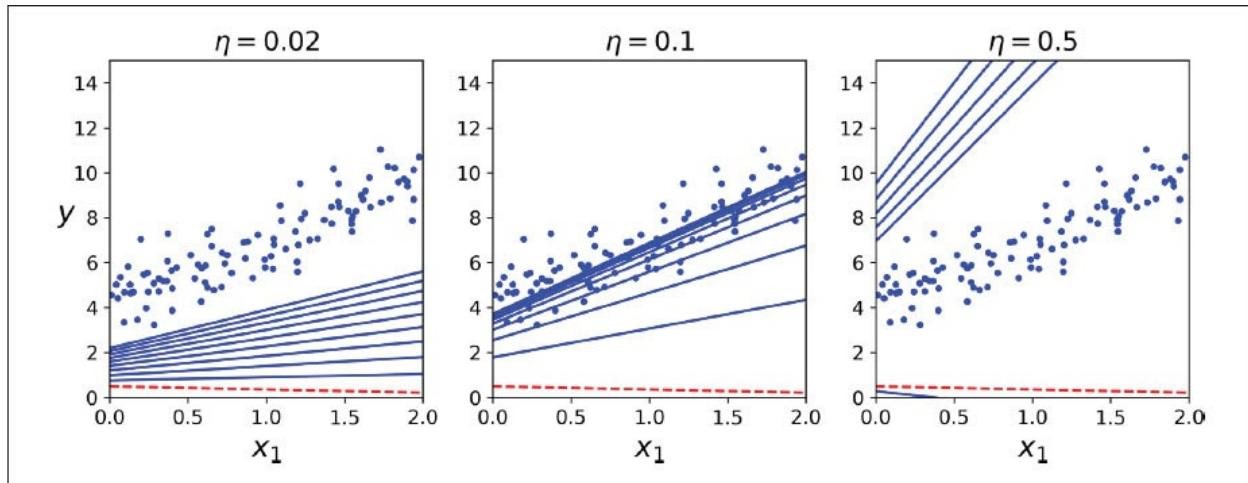


Abbildung 4-8: Gradientenverfahren mit unterschiedlichen Lernraten

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 ϵ 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), konvergiert das Batch-Gradientenverfahren mit einer gegebenen Lernrate immer zur optimalen Lösung, aber Sie müssen eventuell ein Weilchen warten – es kann $O(1/\epsilon)$ Iterationen erfordern, um das Optimum in einem Bereich ϵ zu erreichen (abhängig von der Form der Kostenfunktion). Wenn Sie die Toleranz ϵ durch 10 teilen (um eine genauere Lösung erhalten), muss der Algorithmus etwa 10 Mal länger laufen.

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, siehe [Kapitel 1](#)).

Andererseits ist dieser Algorithmus wegen seiner stochastischen (d.h. zufälligen) Arbeitsweise wesentlich unregelmäßiger als das Batch-Gradientenverfahren: Anstatt sanft zum Minimum hinabzugleiten, 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.

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*, inspiriert vom Ausglühen in der Metallurgie, 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.

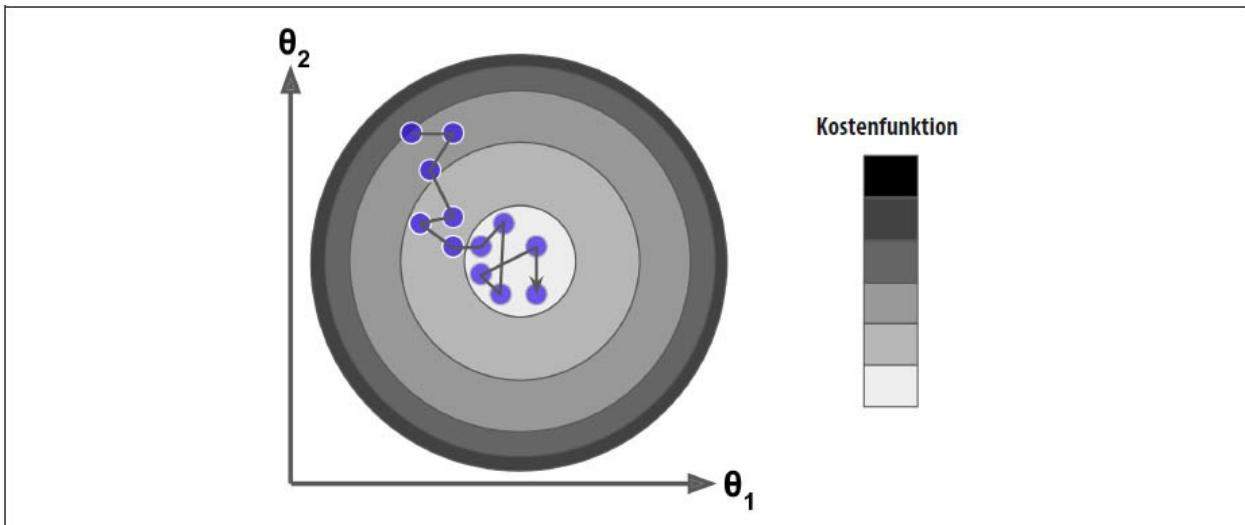


Abbildung 4-9: Mit dem stochastischen Gradientenverfahren ist jeder Trainingsschritt viel schneller, aber auch viel zufälliger als beim Einsatz des Batch-Gradientenverfahrens.

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 1.000 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 20 Schritte beim Trainieren (achten Sie darauf, wie unregelmäßig die Schritte sind).

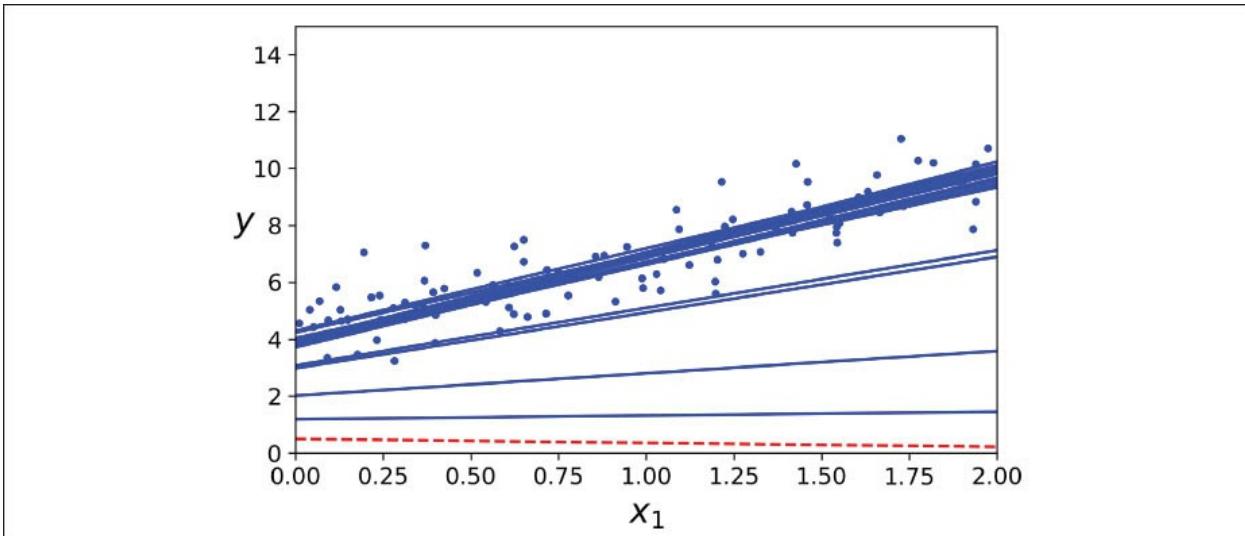
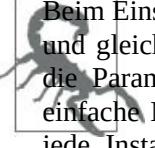


Abbildung 4-10: Die ersten 20 Schritte des stochastischen Gradientenverfahrens

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 sicherstellen, dass die Eingabemerkmale und die Labels zusammenbleiben) und dann Eintrag für Eintrag durchgehen. Anschließend mischen Sie die Daten erneut und so weiter. Allerdings konvergiert dieses Verfahren im Allgemeinen langsamer.



Beim Einsatz des stochastischen Gradientenverfahren müssen die Trainingsinstanzen unabhängig und gleichverteilt sein (Independent and Identically Distributed, IID), um sicherzustellen, dass die Parameter im Durchschnitt in Richtung des globalen Optimums gedrängt werden. Eine einfache Möglichkeit ist, die Instanzen während des Trainings zu durchmischen (zum Beispiel jede Instanz zufällig auszuwählen oder zu Beginn jeder Epoche den Trainingsdatensatz zu mischen). Vermischen Sie die Instanzen nicht – beispielsweise wenn sie anhand ihres Labels geordnet sind –, wird das stochastische Gradientenverfahren damit beginnen, erst für ein Label zu optimieren, dann für das nächste und so weiter. Dabei wird es aber nicht nahe an das globale Minimum gelangen.

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 1.000 Epochen aus, oder es läuft, bis der Verlust während einer Epoche um weniger als 0,001 sinkt (`max_iter=1000, tol=1e-3`). Der Code 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(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)  
  
sgd_reg.fit(X, y.ravel())
```

Die erzielte Lösung liegt erneut 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 Fall 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 herumbewegen.

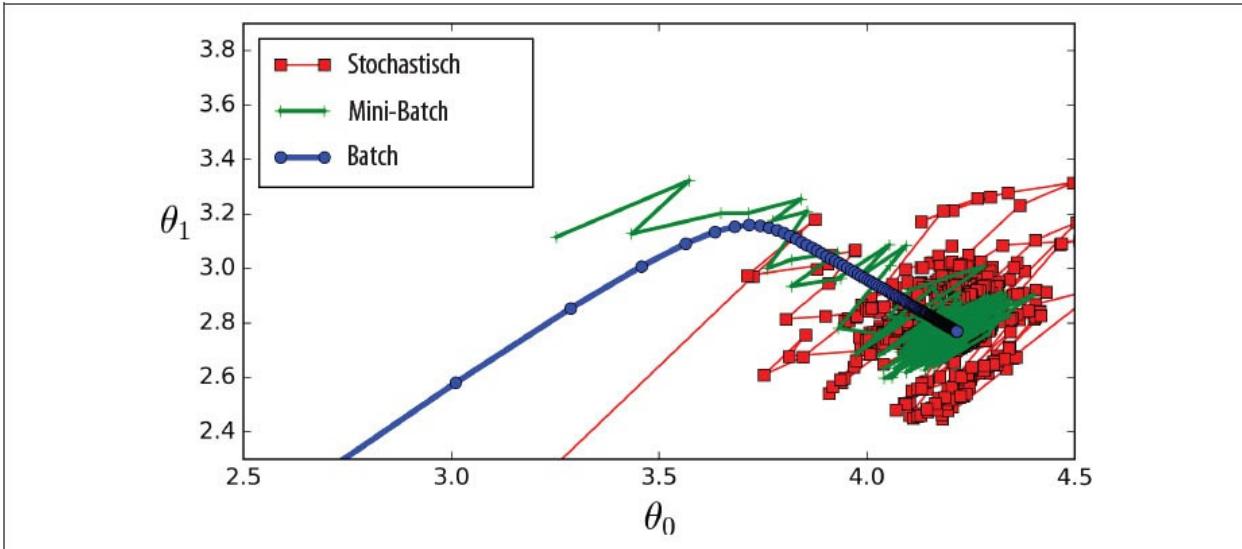


Abbildung 4-11: Pfade von Gradientenverfahren im Parameterraum

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.

Vergleichen wir die bisher besprochenen Algorithmen zur linearen Regression⁶ (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	N/A
SVD	schnell	nein	langsam	0	nein	LinearRegression
Batch-GD	langsam	nein	schnell	2	ja	SGDRegressor
stochastisches GD	schnell	ja	schnell	≥ 2	ja	SGDRegressor
Mini-Batch-GD	schnell	ja	schnell	≥ 2	ja	SGDRegressor



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.

Polynomielle Regression

Wie sieht es aus, wenn Ihre Daten komplexer als eine gerade Linie sind? Überraschenderweise können wir auch nichtlineare Daten mit einem linearen Modell fitten. Dazu können wir einfach Potenzen jedes Merkmals als neue Merkmale hinzufügen und dann ein lineares Modell auf diesem erweiterten Merkmalssatz trainieren. Diese Technik nennt man *polynomielle Regression*.

Betrachten wir ein Beispiel. Zunächst generieren wir einige nichtlineare Daten anhand einer einfachen *quadratischen Gleichung*⁷ (zuzüglich von 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)
```

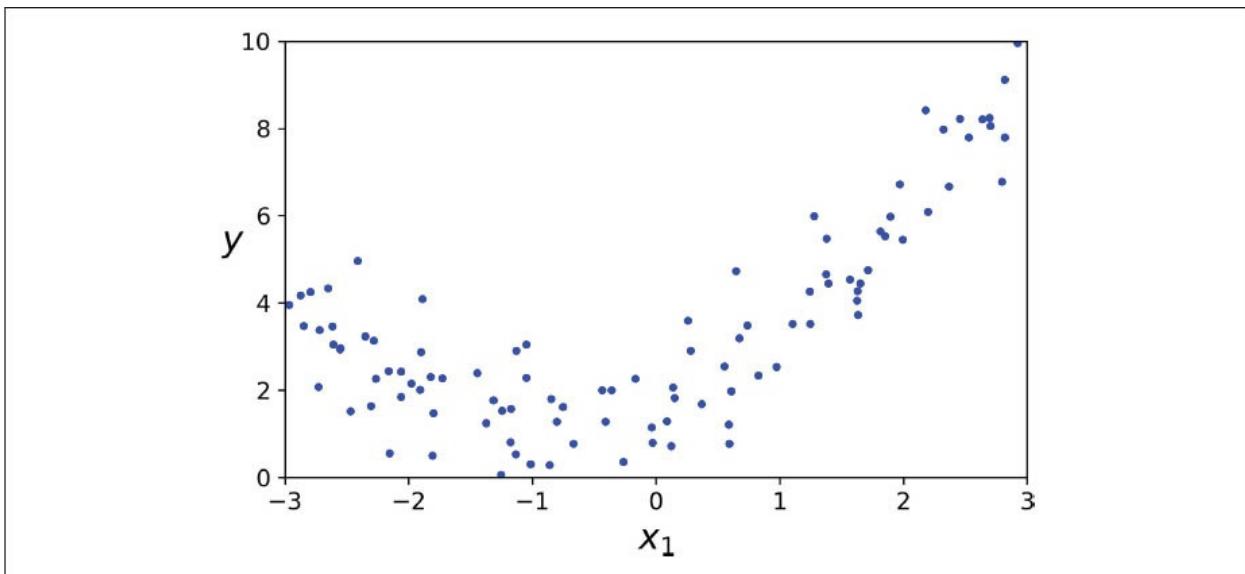


Abbildung 4-12: Generierte nichtlineare verrauschte Daten

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](#)):

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 1.78134581]), array([[ 0.93366893,  0.56456263]]))
```

Nicht schlecht: Das Modell schätzt $\hat{y} = 0,56^2_1 + 0,93x_1 + 1,78$, 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 .

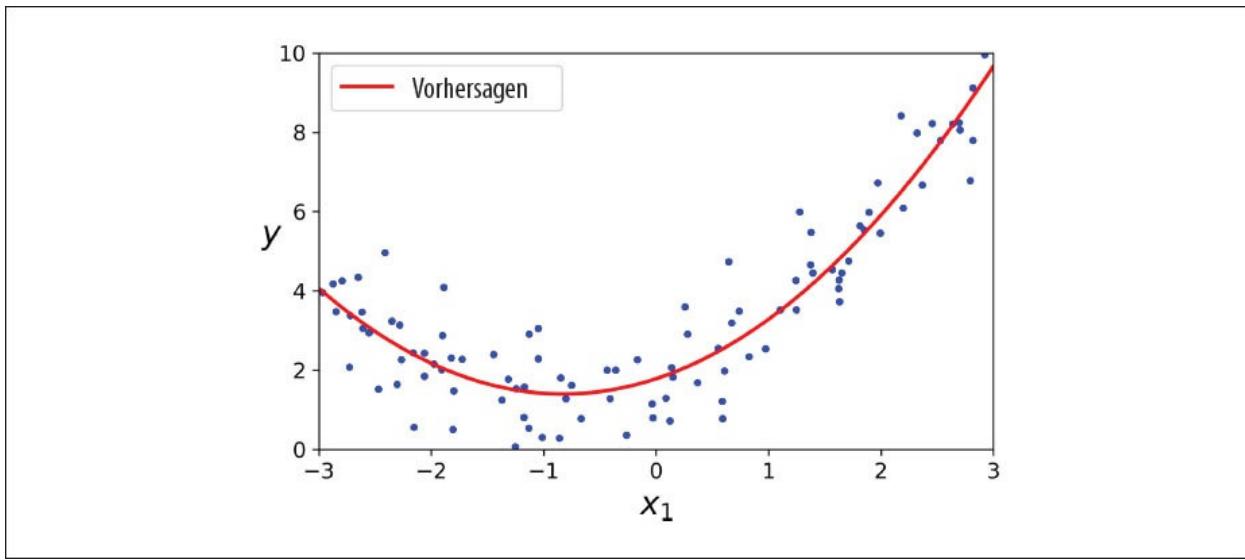


Abbildung 4-13: Vorhersagen eines polynomiellen Regressionsmodells



`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.

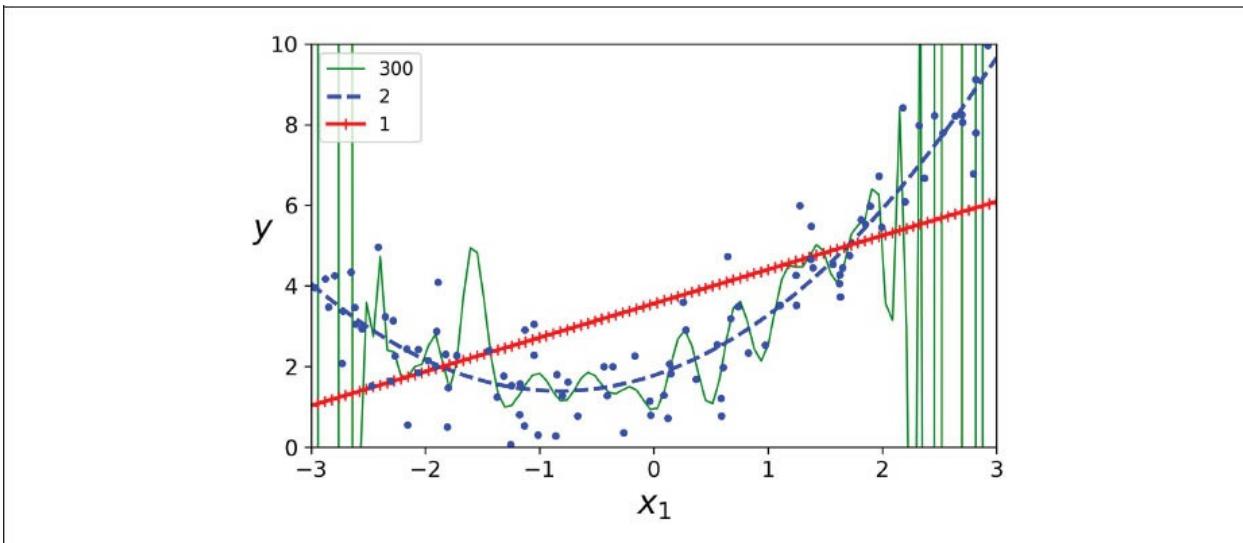


Abbildung 4-14: Regression mit einem höhergradigen Polynom

Natürlich ist beim Modell mit der höhergradigen Regression sehr starkes Overfitting zu finden, während beim linearen Modell Underfitting vorliegt. In diesem Fall verallgemeinert das quadratische Modell am besten. Das 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 muss? 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, liegt in Ihrem Modell Overfitting vor. Erbringt es bei beiden eine schlechte Leistung, 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 (oder die Trainingsiterationen). 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[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    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)

```

An dieser Stelle sind einige Erklärungen zum Underfitting-Modell angebracht. Zuerst betrachten wir die Vorhersageleistung 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 nichtlinear 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.

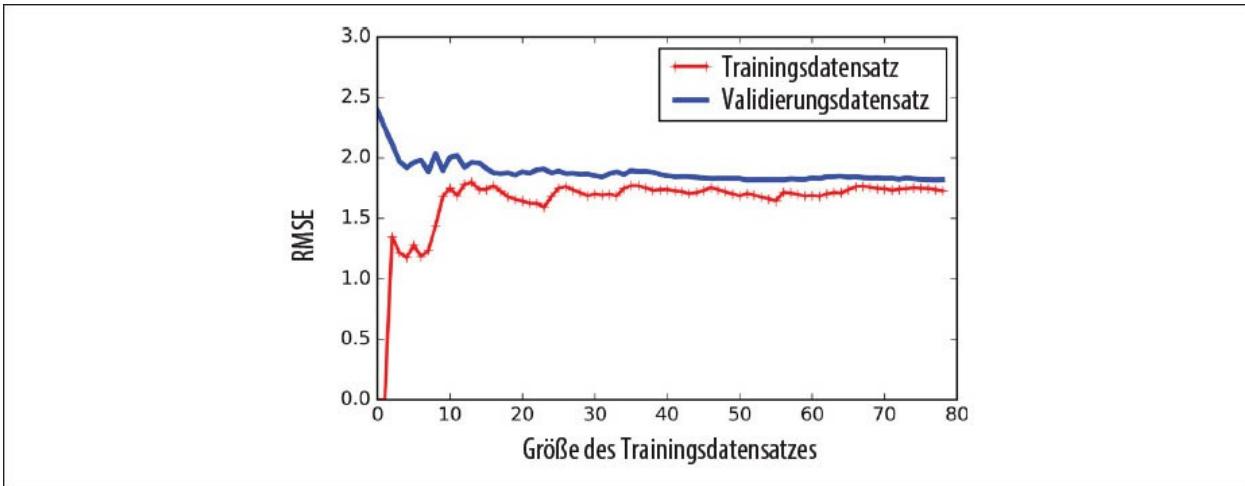


Abbildung 4-15: Lernkurven

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 vorherigen, 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.

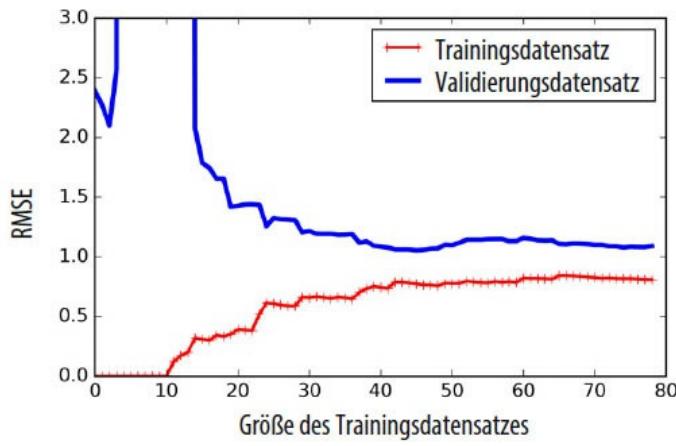


Abbildung 4-16: Lernkurven für ein polynomielles Modell 10. Grades



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.⁸

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 1](#) und [2](#) gesehen haben, ist die Regularisierung des Modells (also 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.

Ridge-Regression

Die *Ridge-Regression* (auch *Regularisierung nach Tikhonov* genannt) ist eine regularisierte 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 tritt recht häufig auf, dass sich die Kostenfunktion beim Trainieren vom Qualitätsmaß beim Testen unterscheidet. Neben der Regularisierung besteht ein weiterer Grund darin, 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.⁹

Formel 4-8: Kostenfunktion bei der Ridge-Regression

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

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), ist der Regularisierungsterm einfach gleich $\frac{1}{2}(\|\mathbf{w}\|_2)^2$, wobei $\|\mathbf{w}\|_2$ für die ℓ_2 -Norm des Gewichtsvektors steht.¹⁰ Beim Gradientenverfahren addieren Sie einfach $\alpha\mathbf{w}$ zum MSE-Gradientenvektor hinzu ([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.

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 wurde die Ridge-Regression auf die fertigen Merkmale angewendet: Dies ist eine polynomiale Regression mit Ridge-Regularisierung.

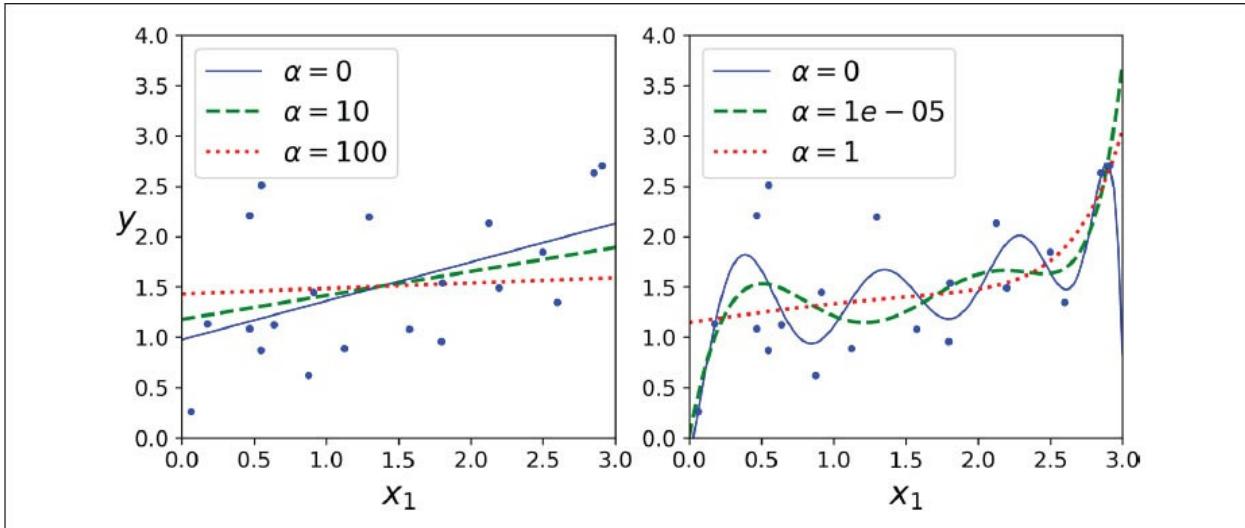


Abbildung 4-17: Ein lineares Modell (links) und ein polynomielles Modell (rechts), beide mit unterschiedlich starker Ridge-Regression

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 $(n + 1) \times (n + 1)$ -*Identitätsmatrix*¹¹ ist, nur dass die linke obere Ecke eine 0 für den Bias-Term enthält.

Formel 4-9: Lösung der geschlossenen Form bei der Ridge-Regression

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{A})^{-1} \mathbf{X}^T \mathbf{y}$$

Die Ridge-Regression lässt sich mit Scikit-Learn in der geschlossenen Form folgendermaßen lösen (mit einer Variante von [Formel 4-9](#) nach einer Technik zur Matrizenfaktorisierung von André-Louis Cholesky):

```
>>> 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]])
```

Über das stochastische Gradientenverfahren:¹²

```
>>> sgd_reg = SGDRegressor(penalty="l2")  
>>> sgd_reg.fit(X, y.ravel())  
>>> sgd_reg.predict([[1.5]])  
array([1.47012588])
```

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 Regression: Wie die Ridge-Regression fügt sie zur Kostenfunktion einen Regularisierungsterm hinzu, dieser verwendet aber die ℓ_1 -Norm des Gewichtsvektors anstatt des halbierten Quadrats der ℓ_2 -Norm (siehe [Formel 4-10](#)).

Formel 4-10: Kostenfunktion bei der Lasso-Regression

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^n |\theta_i|$$

[Abbildung 4-18](#) zeigt das Gleiche wie [Abbildung 4-17](#), aber ersetzt die Ridge-Modelle durch Lasso-Modelle und verwendet kleinere Werte für α .

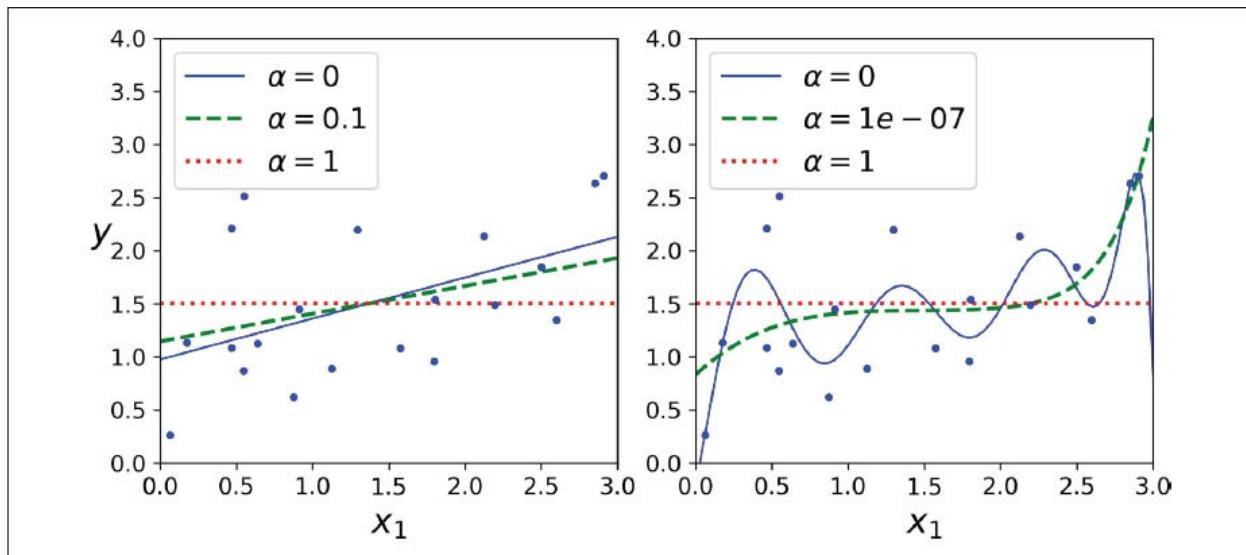


Abbildung 4-18: Ein lineares Modell (links) und ein polynomielles Modell (rechts), beide mit unterschiedlichen Lasso-Regressionen

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 polynomischen Merkmale sind auf null gesetzt. Anders ausgedrückt, die Lasso-Regression führt 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: Die Achsen repräsentieren die beiden Modellparameter, und die Hintergrundkonturen repräsentieren verschiedene Verlustfunktionen. Im oberen linken Plot stehen die Konturen für den ℓ_1 -Verlust ($|\theta_1| + |\theta_2|$), der linear abnimmt, wenn Sie sich einer Achse nähern. Initialisieren Sie beispielsweise die Modellparameter mit $\theta_1 = 2$ und $\theta_2 = 0,5$, wird das Gradientenverfahren beide Parameter gleich verringern (das ist die gestrichelte Linie), und θ_2 wird damit als Erstes null erreichen (da es zu Beginn näher an null war). Danach wird das Gradientenverfahren die Rinne herabrollen, bis es $\theta_1 = 0$ erreicht (mit ein bisschen hin und her springen, da die Gradienten von ℓ_1 niemals nahe an null kommen – sie sind für jeden Parameter entweder -1 oder 1). Im oberen rechten Plot repräsentieren die Konturen die Lasso-Kostenfunktion (also eine MSE-Kostenfunktion plus einen ℓ_1 -Verlust). Die kleinen weißen Punkte zeigen den Weg, dem das Gradientenverfahren folgt, um Modellparameter zu optimieren, die ungefähr mit $\theta_1 = 0,25$ und $\theta_2 = -1$ initialisiert wurden: Beachten Sie, wie der Weg wieder schnell $\theta_2 = 0$ erreicht, es dann die Rinne heruntergeht und schließlich um das globale Minimum herumspringt (dargestellt durch das Quadrat). Würden wir α erhöhen, würde das globale Optimum entlang der gestrichelten Linie nach links verschoben werden; würden wir α verringern, würde das globale Optimum nach rechts verschoben werden (in diesem Beispiel sind die optimalen Parameter für den unregulierten MSE $\theta_1 = 2$ und $\theta_2 = 0,5$).

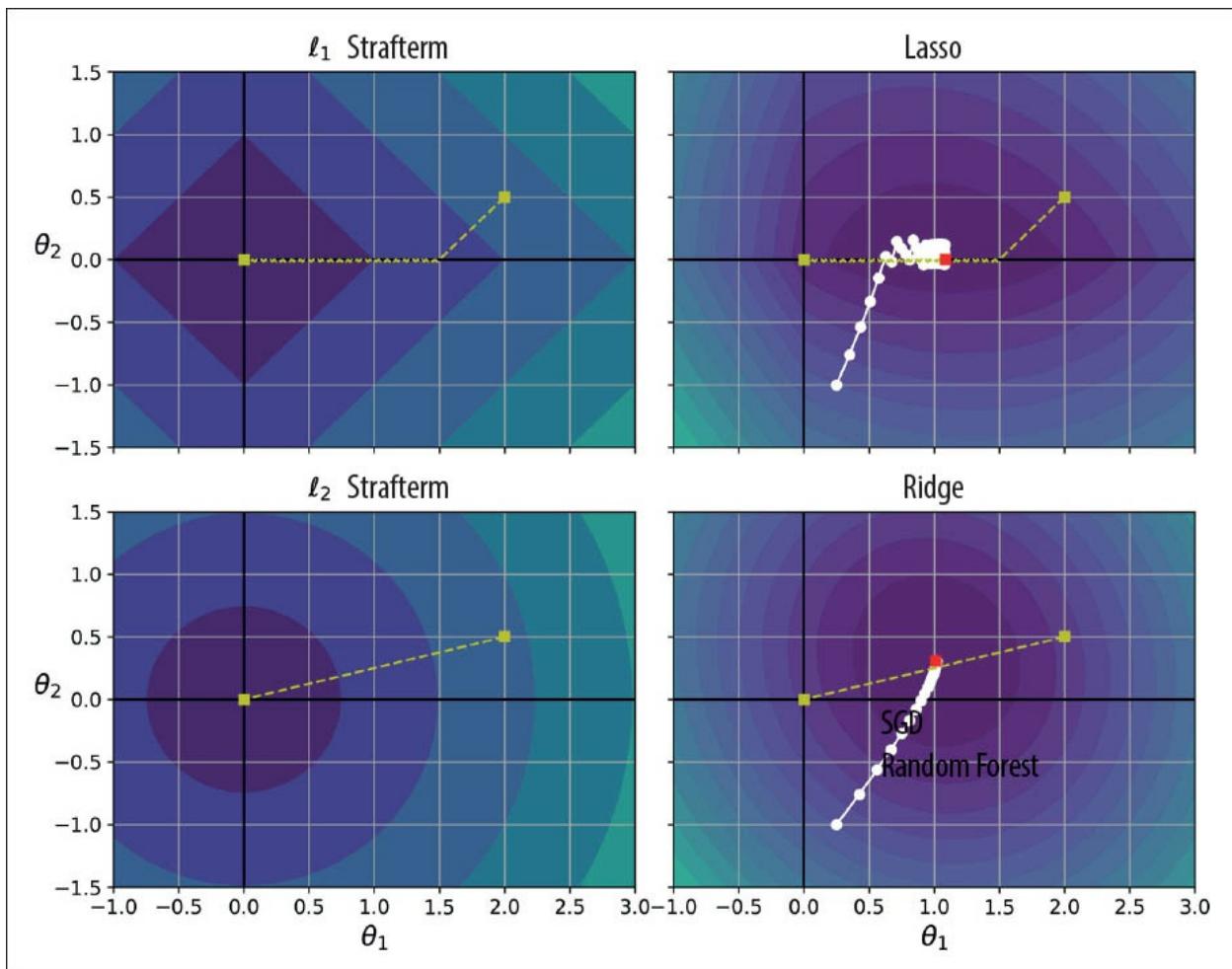


Abbildung 4-19: Lasso im Vergleich zur Ridge-Regularisierung

Die beiden unteren Plots zeigen das Gleiche, aber stattdessen mit einem ℓ_2 -Strafterm. Im linken unteren Plot sehen Sie, dass der ℓ_2 -Verlust mit dem Abstand zum Ursprung abnimmt, sodass das Gradientenverfahren einfach einen geraden Weg zu diesem Punkt wählt. Im unteren rechten Plot zeigen die Konturen die Ridge-Regressions-Kostenfunktion (also eine MSE-Kostenfunktion plus einen ℓ_2 -Verlust). Es gibt zwei wichtige Unterschiede zum Lasso. Erstens werden die Gradienten kleiner, wenn sich die Parameter dem globalen Optimum nähern, sodass das Gradientenverfahren von allein langsamer wird, was dabei hilft, zu konvergieren (da es kein Herumspringen gibt). Und zweitens nähern sich die optimalen Parameter (dargestellt durch das rote Quadrat) immer mehr dem Ursprung, wenn Sie α erhöhen, aber sie werden niemals ganz eliminiert.

- ❖ Um beim Gradientenverfahren zu vermeiden, dass es beim Einsatz des Lassos am Ende um das Optimum herumspringt, müssen Sie die Lernrate während des Trainings nach und nach reduzieren (es wird immer noch um das Optimum herumspringen, aber die Schritte werden immer kleiner werden, und das Ganze wird damit 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.¹³ 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

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix} \quad \text{mit } \text{sign}(\theta_i) = \begin{cases} -1 & \text{wenn } \theta_i < 0 \\ 0 & \text{wenn } \theta_i = 0 \\ +1 & \text{wenn } \theta_i > 0 \end{cases}$$

Hier folgt ein kleines Scikit-Learn-Beispiel für das Verwenden der Klasse Lasso.

```
>>> 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])
```

Sie könnten stattdessen auch die Klasse SGDRegressor(penalty="l1") verwenden.

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

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^n \theta_i^2$$

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. 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 es, 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.

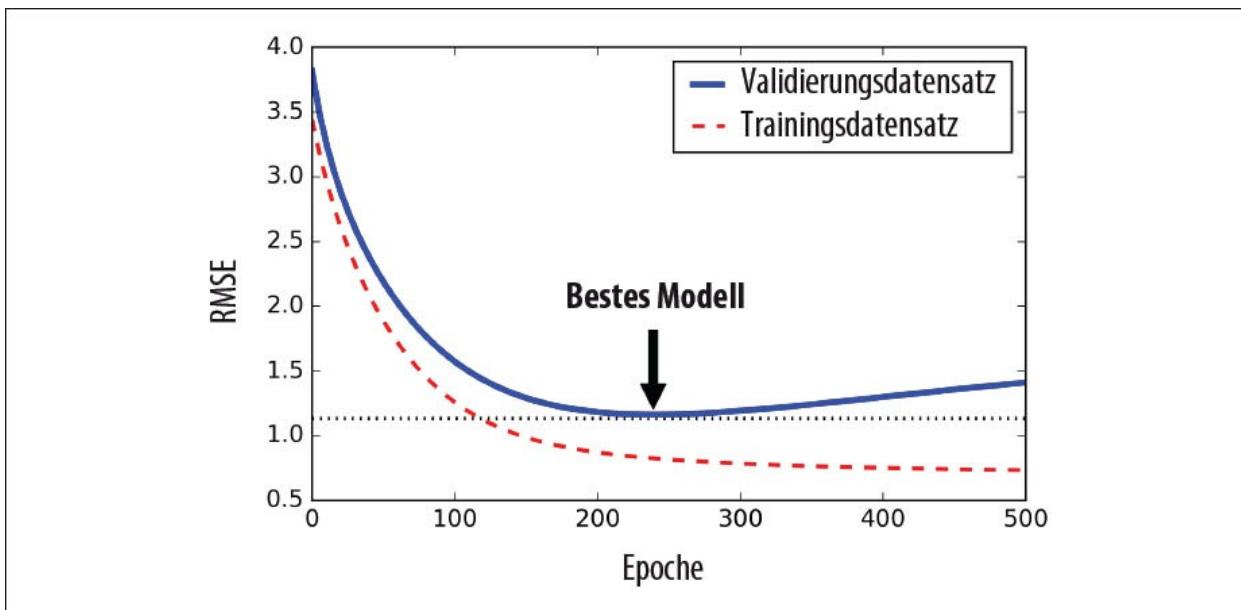


Abbildung 4-20: Regularisierung mit Early Stopping

Tipp: Beim stochastischen und beim 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

# Vorbereiten der Daten

poly_scaler = Pipeline([
    ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
    ("std_scaler", StandardScaler())
])

X_train_poly_scaled = poly_scaler.fit_transform(X_train)

X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, 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, y_val_predict)
    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)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\mathbf{x}^T \boldsymbol{\theta})$$

Die logistische Funktion – 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

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

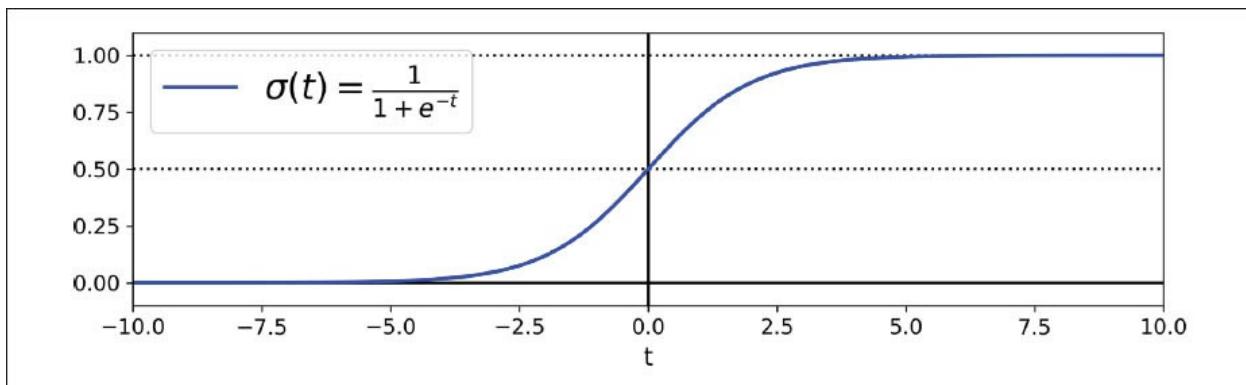


Abbildung 4-21: Logistische Funktion

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 $\mathbf{x}^T \boldsymbol{\theta}$ positiv ist, und 0, falls es negativ ist.



Der Score t wird häufig als *Logit* bezeichnet. Der Name kommt daher, dass es sich bei der Logit-Funktion, definiert als $\text{logit}(p) = \log(p / (1 - p))$, um die Inverse der logistischen Funktion handelt. Wenn Sie den Logit der geschätzten Wahrscheinlichkeit p berechnen, werden Sie feststellen, dass das Ergebnis t ist. Der Logit wird auch als *Log-Odds* bezeichnet, da es sich um den Logarithmus des Verhältnisses zwischen der geschätzten Wahrscheinlichkeit für die positive Kategorie und der für die negative Kategorie handelt.

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

$$c(\boldsymbol{\theta}) = \begin{cases} -\log(\hat{p}) & \text{bei } y = 1, \\ -\log(1 - \hat{p}) & \text{bei } y = 0. \end{cases}$$

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, und 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, die man als *Log Loss* bezeichnet.

Formel 4-17: Kostenfunktion bei der logistischen Regression (Log Loss)

$$J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

Die schlechte Nachricht dabei ist, dass es keine bekannte Gleichung mit geschlossener Form gibt, mit der sich ein Wert für $\boldsymbol{\theta}$ 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 . Modellparameter θ_j sind in [Formel 4-18](#) angegeben.

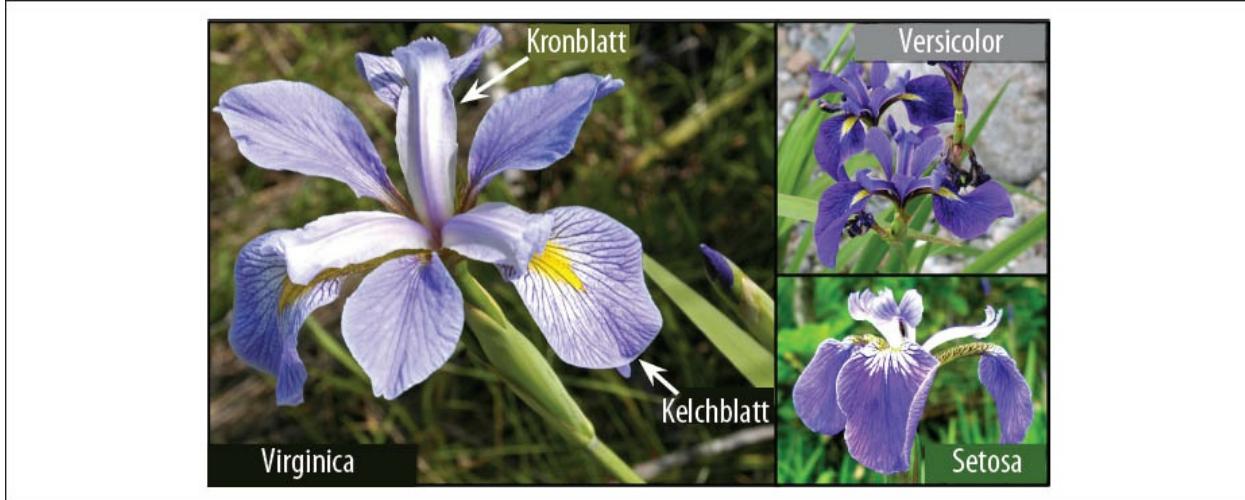
Formel 4-18: Partielle Ableitungen der logistischen Kostenfunktion

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m (\sigma(\theta^\top \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

Diese Gleichung ist [Formel 4-5](#) sehr ähnlich: Für jeden Datenpunkt wird der Vorhersagefehler berechnet und mit dem j . 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-Pflanzen*¹⁴

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', 'target_names', 'DESCR', 'feature_names', 'filename' ]  
>>> 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 (siehe Abbildung 4-23).¹⁵

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)  
  
y_proba = log_reg.predict_proba(X_new)  
  
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
```

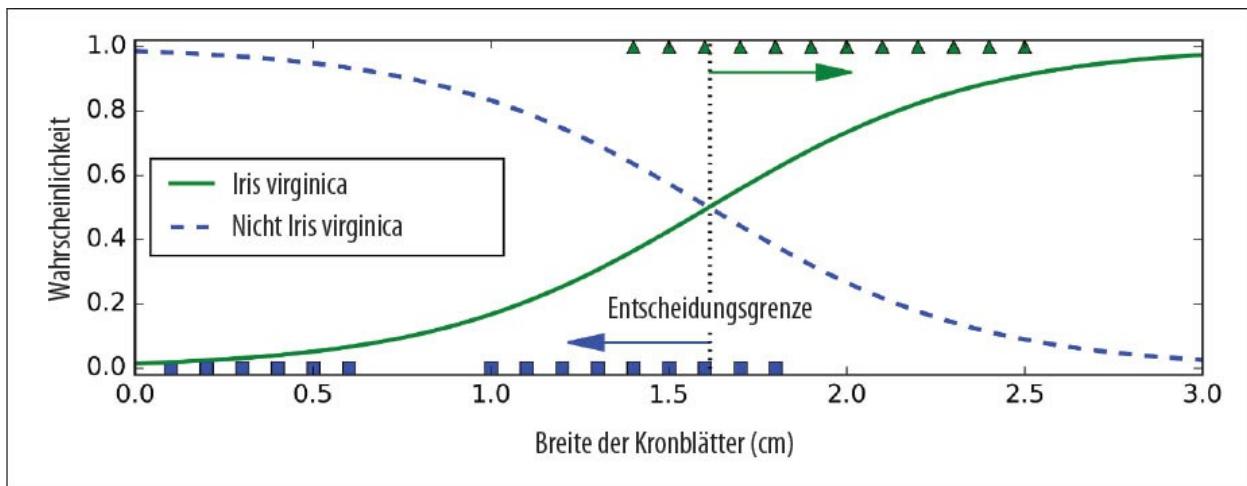


Abbildung 4-23: Geschätzte Wahrscheinlichkeiten und Entscheidungsgrenze

Die Breite der Kronblätter der *Iris-virginica*-Blüten (als Dreiecke dargestellt) liegt zwischen 1,4 und 2,5 cm, während die anderen Iris-Blüten (die Quadrate) schmalere Kronblätter zwischen 0,1 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.¹⁶ Jede der parallelen Linien steht für die Punkte, an denen das Modell eine bestimmte Wahrscheinlichkeit liefert, von 15% (unten links) bis 90% (oben rechts). Sämtliche Blüten jenseits der Linie oben rechts sind laut dem Modell mit mehr als 90%iger Wahrscheinlichkeit *Iris virginica*.

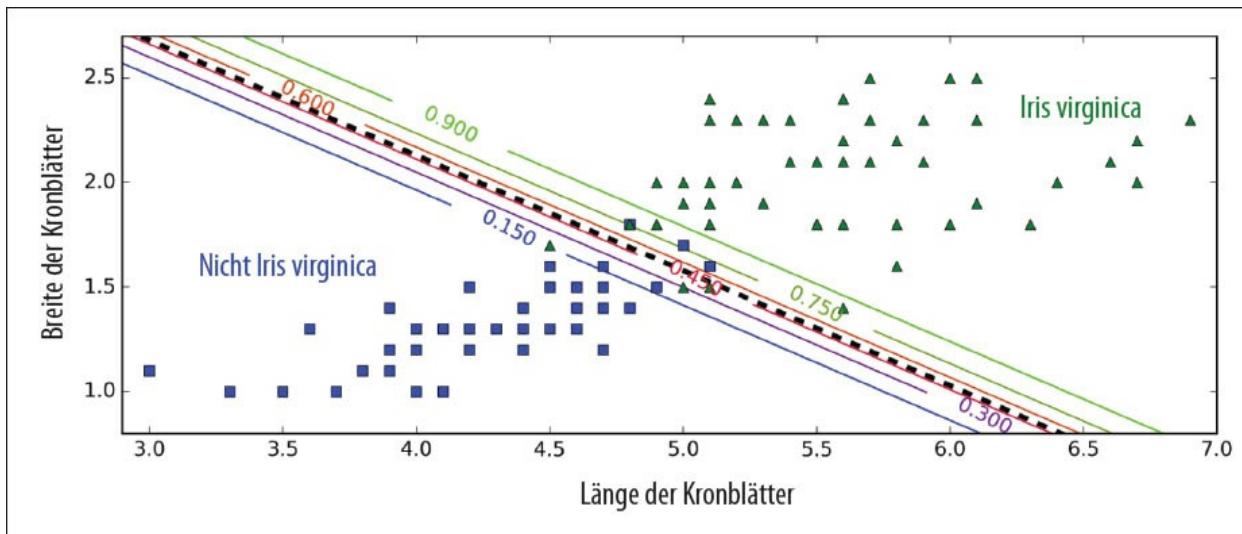


Abbildung 4-24: Lineare Entscheidungsgrenze

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 \mathbf{x} zunächst den Score $s_k(\mathbf{x})$ 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 $s_k(\mathbf{x})$ 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

$$s_k(\mathbf{X}) = \mathbf{X}^T \boldsymbol{\theta}^{(k)}$$

Dabei hat jede Kategorie ihren eigenen Parametervektor $\boldsymbol{\theta}^{(k)}$. Diese Vektoren werden üblicherweise als Zeilen einer *Parametermatrix* Θ abgelegt.

Haben Sie erst einmal den Score jeder Kategorie für den Datenpunkt \mathbf{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). Die Scores werden im Allgemeinen als Logits oder Log-Odds bezeichnet (auch wenn es sich eigentlich um nicht normalisierte Log-Odds handelt).

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

$$\hat{y} = \operatorname{argmax}_k \sigma(\mathbf{s}(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k \left((\boldsymbol{\theta}^{(k)})^T \mathbf{x} \right)$$

- Der Operator *argmax* liefert den Wert einer Variablen, 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 abstrahrt. Mit der Kreuzentropie wird häufig gemessen, wie gut die Wahrscheinlichkeiten der Kategorien mit den Zielkategorien übereinstimmen.

Formel 4-22: Die Kreuzentropie als Kostenfunktion

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(p_k^{(i)})$$

Hier gilt:

- $y_k^{(i)}$ ist die Zielwahrscheinlichkeit dafür, dass die i . Instanz zur Kategorie k gehört. Im Allgemeinen ist sie entweder 1 oder 0 – abhängig davon, ob der Datenpunkt zur Kategorie gehört oder nicht.

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 wir 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). Sind Ihre Annahmen aber falsch (z.B. weil es häufig regnet), erhöht sich die Kreuzentropie um einen Betrag, den man als *Kullback-Leibler-Divergenz* (KL-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). Mehr Informationen dazu finden Sie in meinem Video zum Thema (<https://homl.info/xentropy>).

Der Gradientenvektor dieser Kostenfunktion nach $\Theta^{(k)}$ ist in [Formel 4-23](#) geschrieben.

Formel 4-23: Gradientenvektor der Kreuzentropie für Kategorie k

$$\nabla_{\Theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

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` nutzt standardmäßig die One-versus-the-Rest-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([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

Abbildung 4-25 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. steht die mit 0,450 beschriftete Linie für eine Wahrscheinlichkeit von 45%). Das Modell kann eine Kategorie vorhersagen, die eine geschätzte Wahrscheinlichkeit unter 50% hat. Beispielsweise ist an dem Punkt, an dem sich alle drei Linien treffen, die Wahrscheinlichkeit für alle Kategorien die gleiche, nämlich 33%.

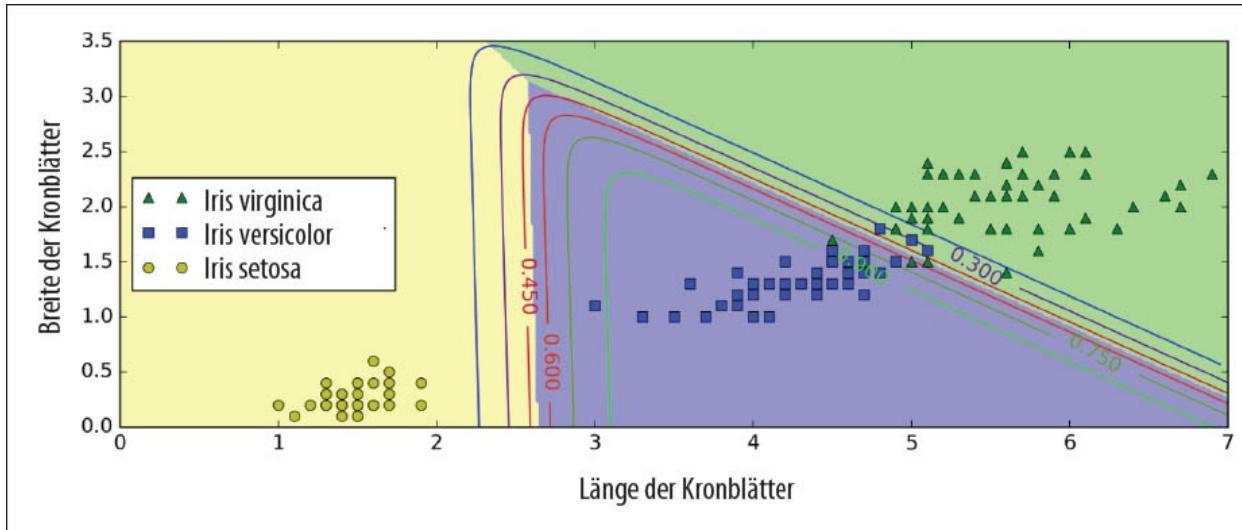


Abbildung 4-25: Entscheidungsgrenzen bei der Softmax-Regression

Übungen

1. Welchen Trainingsalgorithmus für die lineare Regression können Sie verwenden, wenn Sie einen Trainingsdatensatz mit Millionen Merkmalen haben?
2. Nehmen wir 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 wir 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 polynomiale 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?
 - a. Ridge-Regression anstelle einer einfachen linearen Regression (d.h. ohne Regularisierung)?
 - b. Lasso anstelle einer Ridge-Regression?
 - c. 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](#).

KAPITEL 5

Support Vector Machines

Eine *Support Vector Machine* (SVM) ist ein 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 breitestmögliche Straße zwischen den Kategorien vorstellen (hier als parallele gestrichelte Linien dargestellt). Dies bezeichnet man auch als *Large-Margin-Klassifikation*.

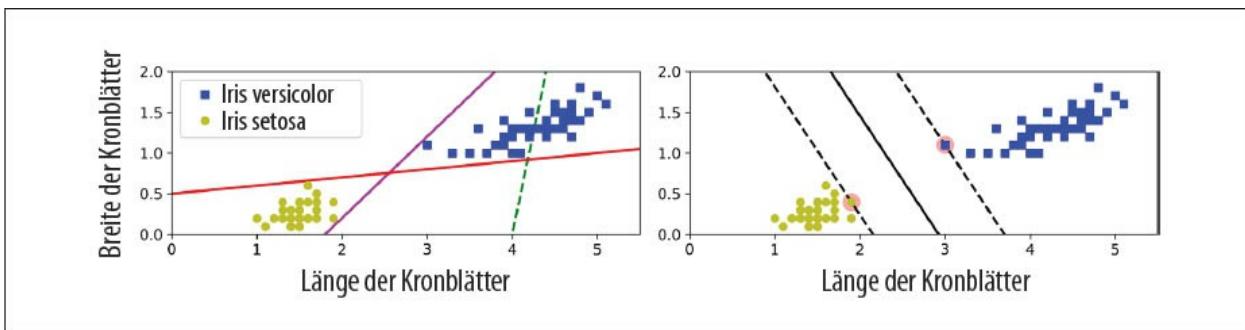


Abbildung 5-1: Large-Margin-Klassifikation

Beachten Sie, dass das Hinzufügen neuer Trainingsdaten »abseits der Straße« die Entscheidungsgrenze überhaupt nicht beeinflusst: Sie wird ausschließlich durch die Datenpunkte am Rand 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).

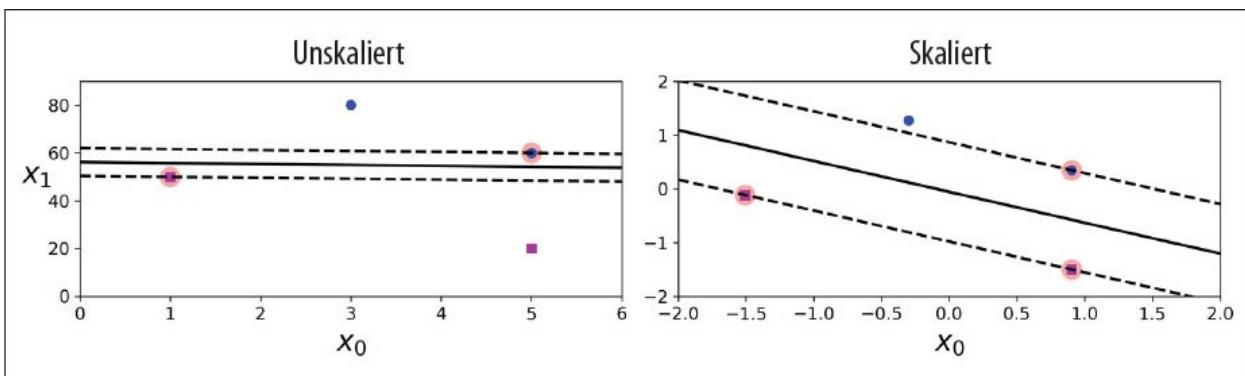


Abbildung 5-2: Empfindlichkeit für Skalierung der Merkmale

- 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 breitestmögliche 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).

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 Margins 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.

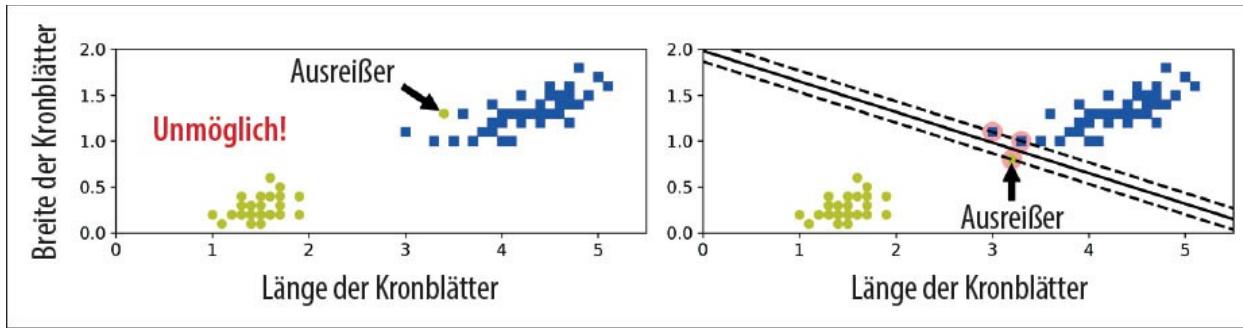


Abbildung 5-3: Anfälligkeit der Hard-Margin-Klassifikation für Ausreißer

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*.

Beim Erstellen eines SVM-Modells mit Scikit-Learn können wir eine Reihe von Hyperparametern angeben. C ist einer davon. Setzen wir ihn auf einen niedrigen Wert, landen wir bei dem Modell, das in Abbildung 5-4 links zu sehen ist. Mit einem hohen Wert erhalten wir das Modell auf der rechten Seite. Verletzungen des Margins sind schlecht. Es ist im Allgemeinen besser, nur wenige von ihnen zu haben. In diesem Fall besitzt das Modell auf der linken Seite viele Verletzungen des Margins, wird aber vermutlich besser verallgemeinern.

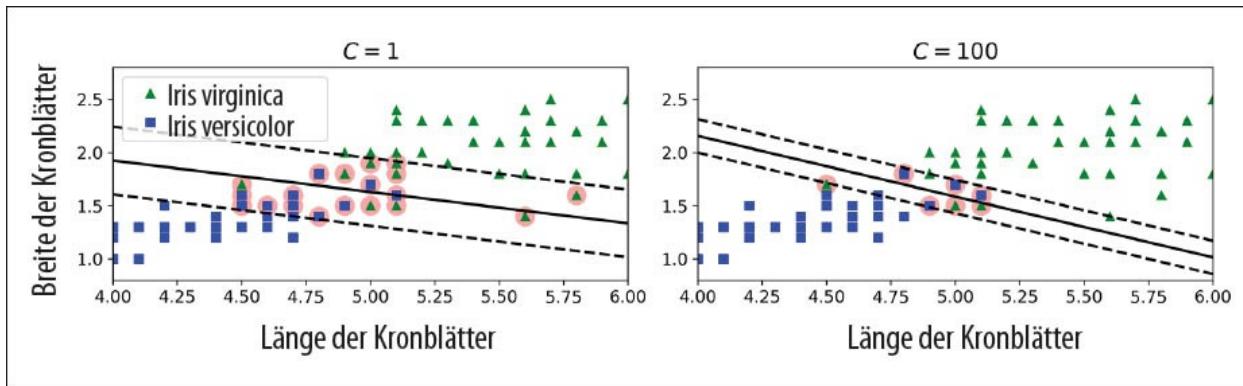


Abbildung 5-4: Breiterer Margin (links) gegenüber weniger Margin-Verletzungen (rechts)



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:

```

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)

Das dabei erhaltene Modell sehen Sie auf der linken Seite von [Abbildung 5-4](#).

Anschließend können Sie mit dem Modell wie bisher Vorhersagen vornehmen:

```

>>> svm_clf.predict([[5.5, 1.7]])
array([1.])

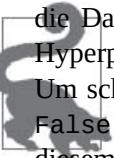
```



Im Gegensatz zu Klassifikatoren, die mit der logistischen Regression arbeiten, geben SVM-Klassifikatoren keine Wahrscheinlichkeiten für die einzelnen Kategorien aus.

Statt der Klasse `LinearSVC` könnten Sie auch die Klasse `SVC` mit einem linearen Kernel verwenden. Beim Erstellen des `SVC`-Modells würden wir `SVC(kernel="linear", C=1)` schreiben. 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.

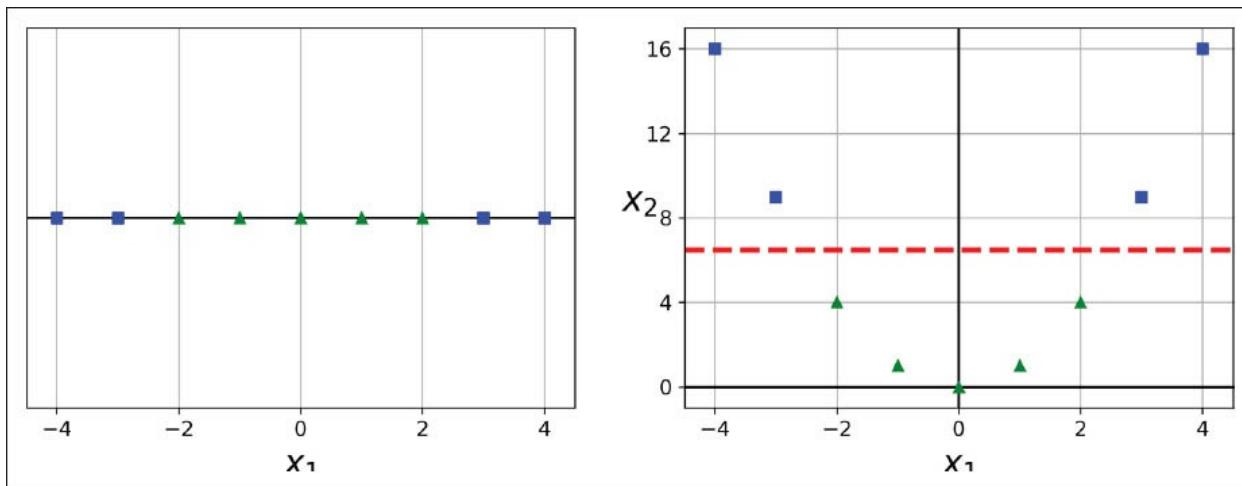


Abbildung 5-5: Hinzufügen von Merkmalen, um einen linear separierbaren Datensatz zu erhalten

Dieser Ansatz lässt sich mit Scikit-Learn umsetzen, indem Sie eine Pipeline erstellen, die aus einem `PolynomialFeatures`-Transformer (wie in »Polynomielle Regression« besprochen) und anschließend einem `StandardScaler` sowie einem `LinearSVC` besteht. Probieren wir dies anhand des Datensatzes `moons` aus. Dabei handelt es sich um einen Spieldatensatz, bei dem die Datenpunkte als zwei ineinander verschränkte Halbkreise geformt sind (siehe [Abbildung 5-6](#)). Sie können diesen Datensatz mit der Funktion `make_moons()` erzeugen:

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
```

```

X, y = make_moons(n_samples=100, noise=0.15)

polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])

polynomial_svm_clf.fit(X, y)

```

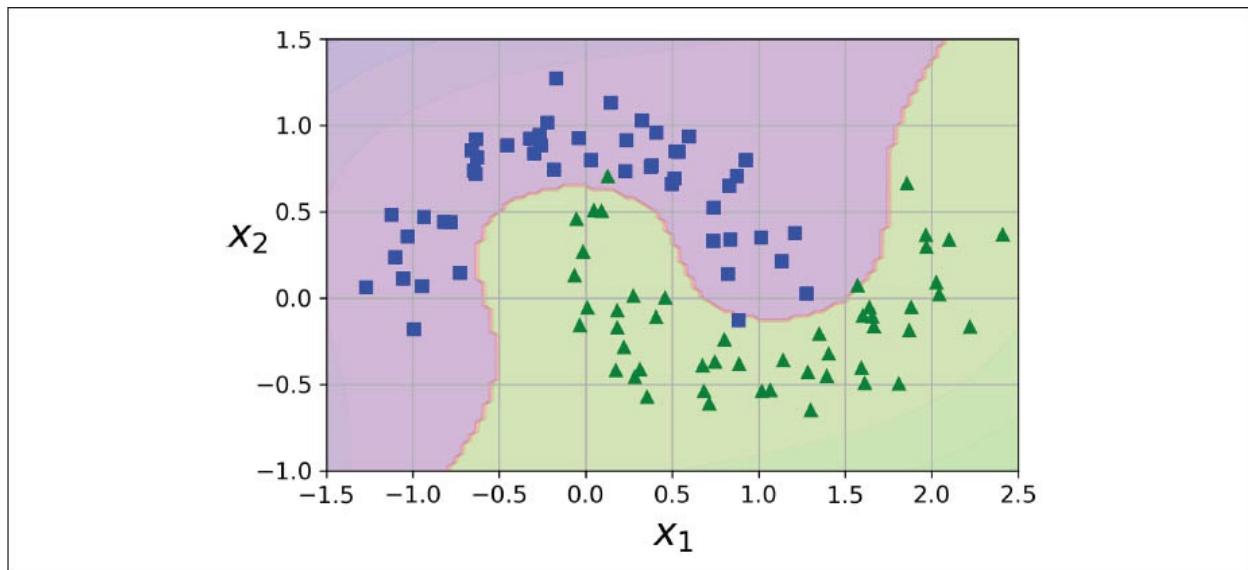


Abbildung 5-6: Linearer SVM-Klassifikator mit polynomiellen Merkmalen

Polynomieller Kernel

Das Hinzufügen polynomieller Merkmale lässt sich einfach umsetzen und funktioniert bei vielen Machine-Learning-Algorithmen sehr gut (nicht nur bei 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 *Kerneltrick* (ihn 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. Sie sehen ihn auf der linken Seite von [Abbildung 5-7](#). Auf der rechten Seite wird 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, wenn Underfitting vorliegt. Der Hyperparameter `coef0` steuert, wie stark das Modell von den höhergradigen im Gegensatz zu den niedriggradigen Polynomialtermen beeinflusst wird.

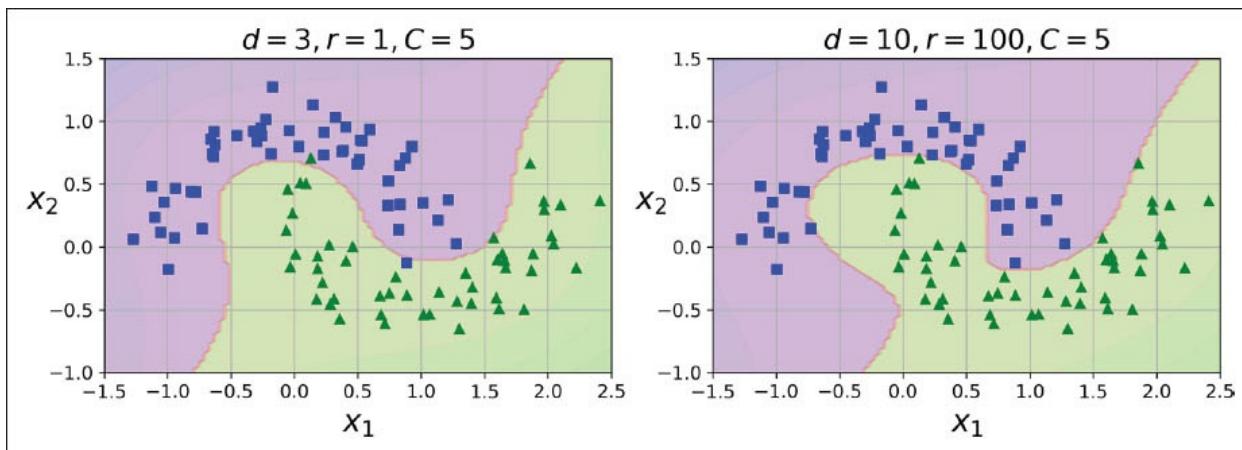


Abbildung 5-7: SVM-Klassifikatoren mit polynomiellem Kernel

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 Hyperparameterraums zu durchkämmen.

Ähnlichkeitsbasierte Merkmale

Eine weitere Technik zum Umgang mit nichtlinearen Aufgaben besteht darin, 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.

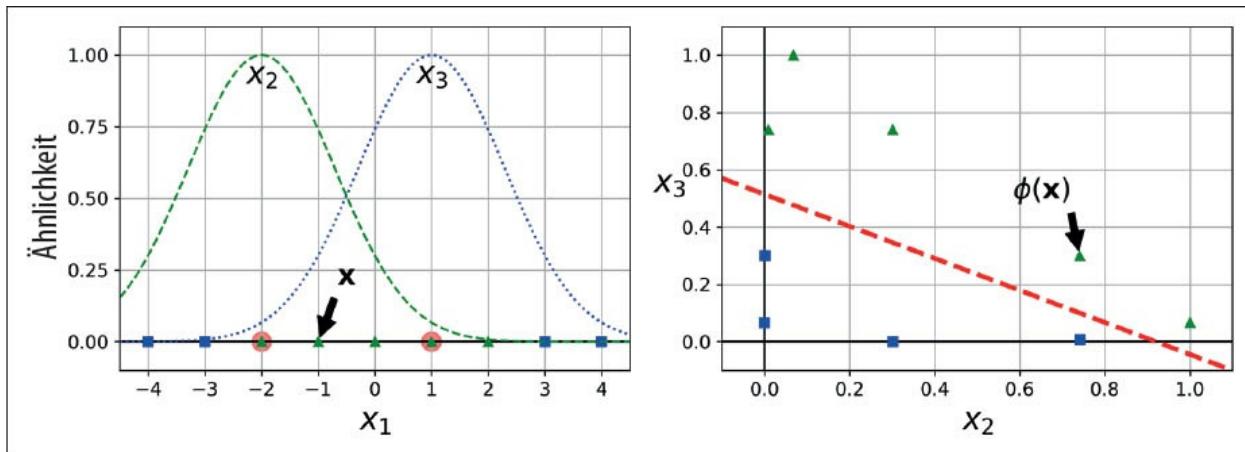


Abbildung 5-8: Ähnlichkeit von Merkmalen, berechnet mit der gaußschen RBF

Sie fragen sich vielleicht, wie die Landmarken ausgesucht werden. Die einfachste Möglichkeit ist, lediglich 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 Kerneltrick der SVMs zu Hilfe: Er ermöglicht es, ein zu vielen ähnlichkeitsbasierten Merkmale äquivalentes Ergebnis zu erhalten. 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 (rechte Diagramme 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).

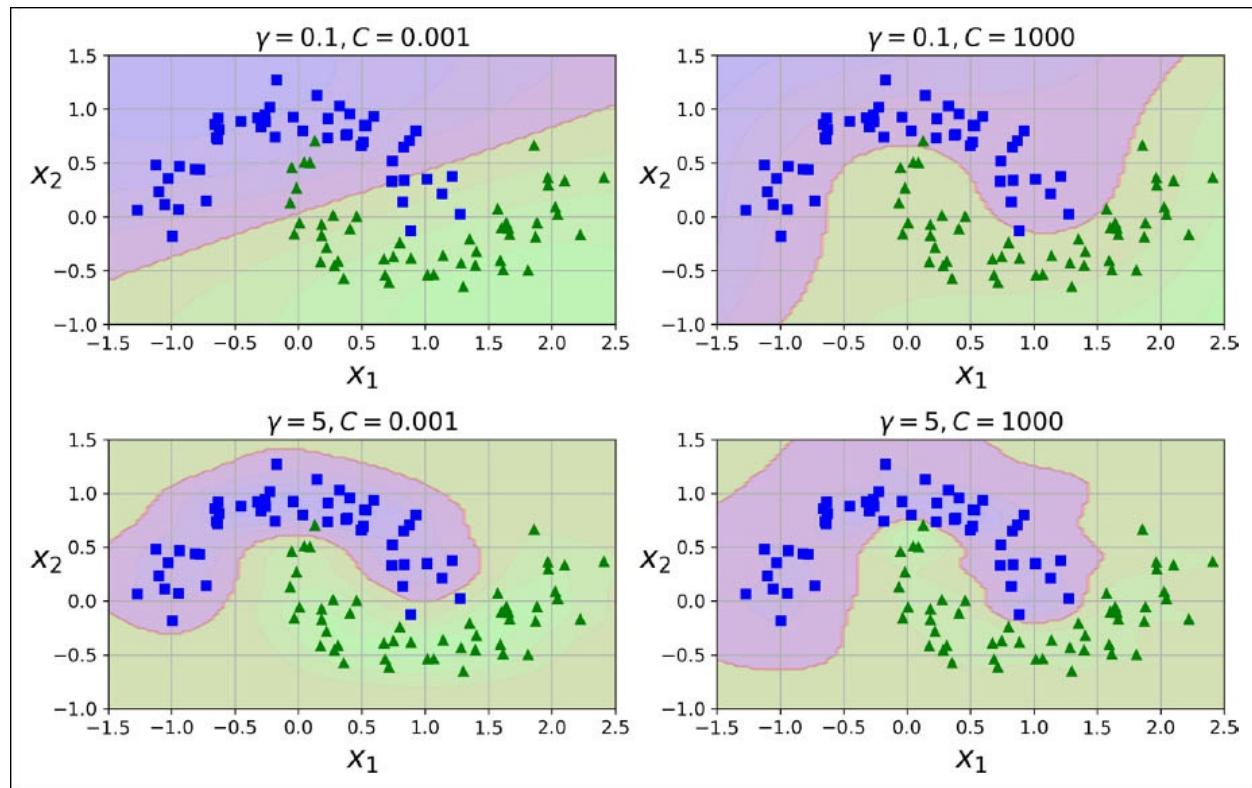


Abbildung 5-9: SVM-Klassifikatoren mit RBF-Kernels

Es gibt noch weitere Kernels, aber diese kommen sehr viel 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. der *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 es auch mit dem gaußschen RBF-Kernel versuchen; 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 (<https://homl.info/13>) für lineare SVMs enthält.¹ Sie unterstützt den Kerneltrick 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 (<https://homl.info/14>) den Kerneltrick 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 von 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 von Nicht-null-Merkmalen 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	Zeitliche Komplexität	Out-of-Core möglich	Scaling nötig	Kerneltrick
<code>LinearSVC</code>	$O(m \times n)$	nein	ja	nein
<code>SGDClassifier</code>	$O(m \times n)$	ja	ja	nein
<code>SVC</code>	$O(m^2 \times n)$ 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 die lineare und nichtlineare Regression. Um SVMs zur Regression statt zur Klassifikation zu verwenden, 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 ($\varepsilon = 1,5$) und das andere mit schmalem Margin ($\varepsilon = 0,5$).

Das Hinzufügen von weiteren Trainingsdatenpunkten innerhalb des Margins beeinflusst die Vorhersagen des Modells nicht; man bezeichnet dieses Modell daher als ε -insensitiv.

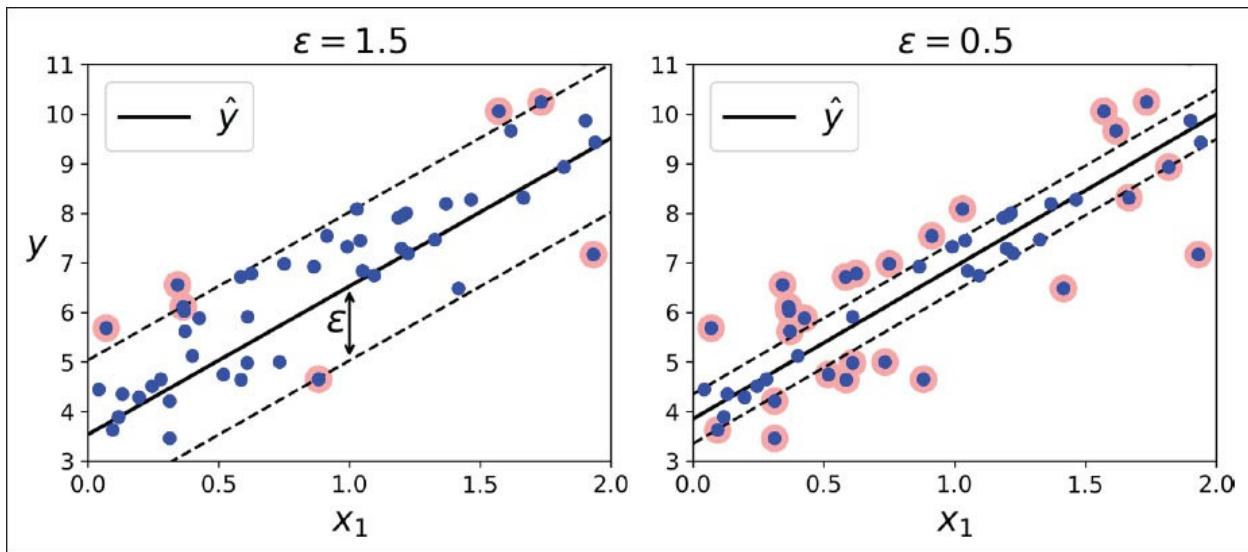


Abbildung 5-10: SVM-Regression

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).

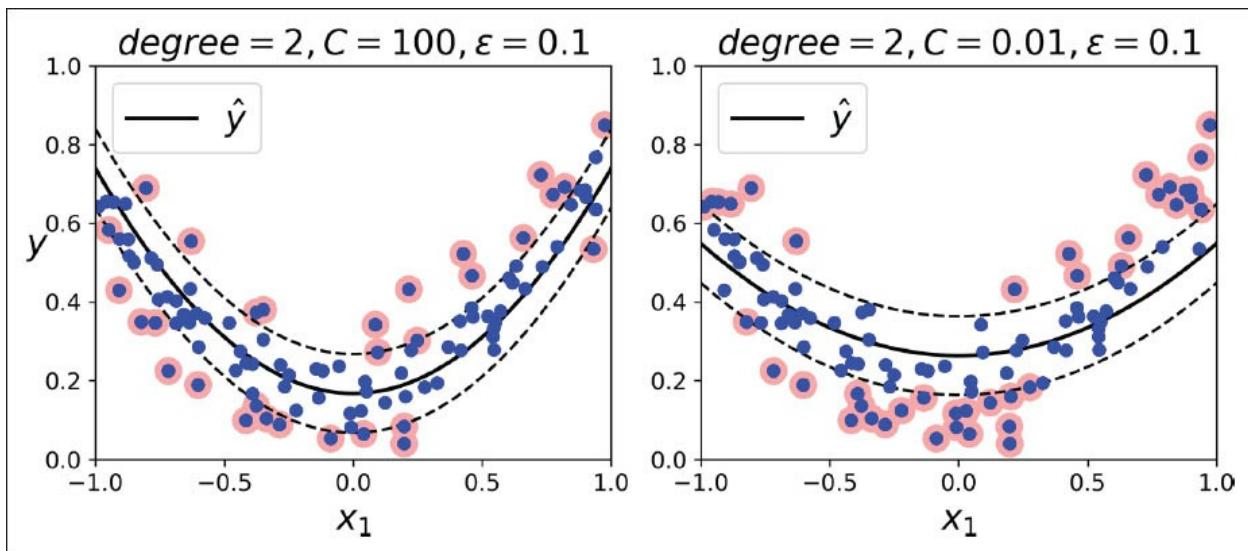


Abbildung 5-11: SVM-Regression mit einem polynomiellen Kernel 2. Grades

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 Kerneltrick unterstützt).

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

Die Klasse SVR ist das Pendant zur 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).



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 üblichere) Notation verwenden: Den Bias-Term bezeichnen wir als b und den Vektor mit den Gewichten der Merkmale als \mathbf{w} . Wir müssen diesmal zum Merkmalsvektor kein Bias-Merkmal hinzufügen.

Entscheidungsfunktion und Vorhersagen

Der lineare SVM-Klassifikator sagt die Kategorie eines neuen Datenpunkts \mathbf{x} vorher, indem er einfach die Entscheidungsfunktion $\mathbf{w}^T \mathbf{x} + b = w_1 x_1 + \dots + w_n x_n + b$ 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

$$\hat{y} = \begin{cases} 0 & \text{wenn } \mathbf{w}^T \mathbf{x} + b < 0, \\ 1 & \text{wenn } \mathbf{w}^T \mathbf{x} + b \geq 0 \end{cases}$$

[Abbildung 5-12](#) zeigt die Entscheidungsfunktion, die dem Modell auf der linken 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

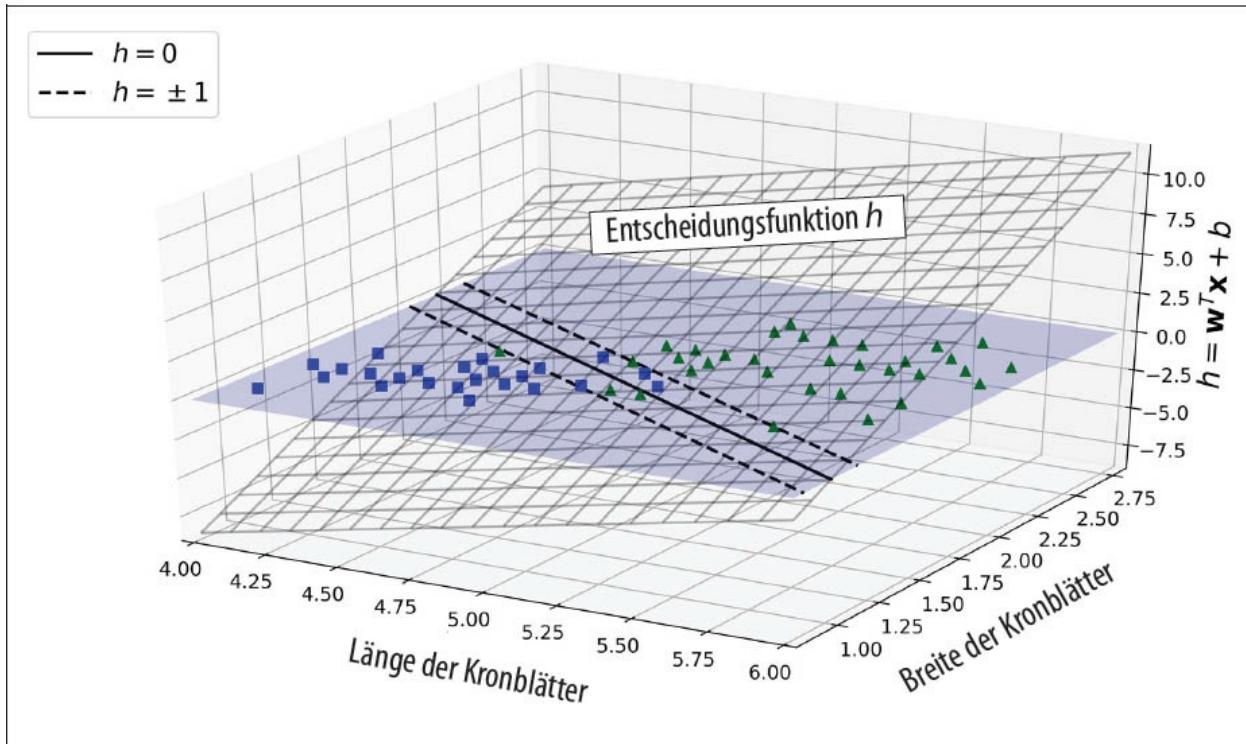


Abbildung 5-12: Entscheidungsfunktion für den Iris-Datensatz

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 \mathbf{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 $\|\mathbf{w}\|$. Wenn wir diese Steigung durch 2 teilen, 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 Margins mit dem Faktor 2. Dies lässt sich etwas besser in 2-D veranschaulichen, wie in Abbildung 5-13 gezeigt. Je kleiner der Gewichtsvektor \mathbf{w} , umso größer wird der Margin.

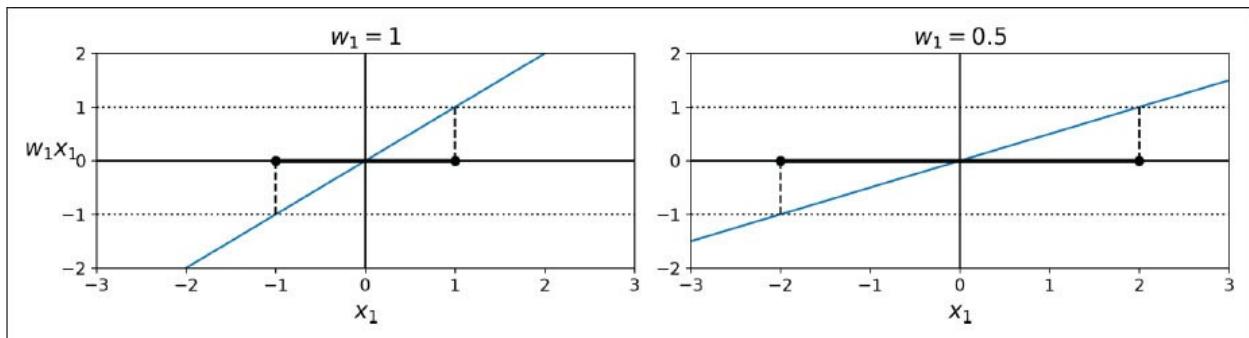


Abbildung 5-13: Ein kleinerer Gewichtsvektor führt zu einem größeren Margin.

Wir möchten also $\|\mathbf{w}\|$ minimieren, um einen großen Margin zu erhalten. Wenn wir allerdings jegliche Verletzungen des Margins 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), können wir diese Bedingung für alle Datenpunkte durch $t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1$ 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

$$\underset{\mathbf{w}, b}{\text{minimiere}} \quad \frac{1}{2} \mathbf{w}^\top \mathbf{w}$$

$$\text{minimiere } 12 \text{ unter der Bedingung } t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 \text{ für } i = 1, 2, \dots, m$$

- ☞ Wir minimieren $1/2 \mathbf{w}^\top \mathbf{w}$, was gleich $1/2 \|\mathbf{w}\|^2$ ist, anstatt $\|\mathbf{w}\|$ zu minimieren. Denn $1/2 \|\mathbf{w}\|^2$ besitzt eine einfache Ableitung (sie beträgt einfach nur \mathbf{w}), während $\|\mathbf{w}\|$ bei $\mathbf{w} = \mathbf{0}$ 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 . Datenpunkt den Margin verletzen darf. Wir haben nun zwei gegenläufige Ziele: die Slack-Variablen so klein wie möglich zu machen, um Verletzungen des Margins zu verringern, und $1/2 \mathbf{w}^\top \mathbf{w}$ 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](#).

Formel 5-4: Zielfunktion eines linearen SVM-Klassifikators mit Soft-Margin

$$\underset{\mathbf{w}, b, \zeta}{\text{minimiere}} \quad \frac{1}{2} \mathbf{w}^\top \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)}$$

unter den Bedingungen $t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)}$ und $\zeta^{(i)} \geq 0$ für $i = 1, 2, \dots, m$

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

$$\begin{aligned} & \underset{\mathbf{p}}{\text{Minimiere}} \quad \frac{1}{2} \mathbf{p}^\top \mathbf{H} \mathbf{p} + \mathbf{f}^\top \mathbf{p} \\ & \text{unter der Bedingung} \quad \mathbf{A} \mathbf{p} \leq \mathbf{b} \\ & \text{wobei} \quad \left\{ \begin{array}{ll} \mathbf{p} & \text{ein } n_p\text{-dimensionaler Vektor } (n_p = \text{Anzahl Parameter}), \\ \mathbf{H} & \text{eine } n_p \times n_p\text{-Matrix,} \\ \mathbf{f} & \text{ein } n_p\text{-dimensionaler Vektor,} \\ \mathbf{A} & \text{eine } n_c \times n_p\text{-Matrix } (n_c = \text{Anzahl Nebenbedingungen}), \\ \mathbf{b} & \text{ein } n_c\text{-dimensionaler Vektor ist.} \end{array} \right. \end{aligned}$$

Beachten Sie, dass der Ausdruck $\mathbf{A} \mathbf{p} \leq \mathbf{b}$ genau n_c Nebenbedingungen definiert: $\mathbf{p}^\top \mathbf{a}^{(i)} \leq b^{(i)}$ mit $i = 1, 2, \dots, n_c$, wobei $\mathbf{a}^{(i)}$ ein Vektor mit den Elementen der i . Zeile von \mathbf{A} und $b^{(i)}$ das i . 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).
- $\mathbf{f} = 0$, ein n_p -dimensionaler Vektor voller Nullen.
- $\mathbf{b} = -1$, ein n_c -dimensionaler Vektor voll mit -1 .
- $\mathbf{a}^{(i)} = -t^{(i)} \hat{\mathbf{x}}^{(i)}$, wobei $\hat{\mathbf{x}}^{(i)}$ gleich $\mathbf{x}^{(i)}$ mit dem zusätzlichen Bias-Merkmal $\hat{x}_0 = 1$ ist.

Demnach können Sie einen linearen Hard-Margin-SVM-Klassifikator trainieren, indem Sie einen QP-Solver von der Stange verwenden und ihm die oben angegebenen 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 $i = 1, 2, \dots, n$. 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 Kerneltrick 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 einer linearen SVM

$$\underset{\alpha}{\text{minimiere}} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)\top} \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

unter der Bedingung $\alpha^{(i)} \geq 0$ 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

$$\begin{aligned} \hat{\mathbf{w}} &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)} \\ \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m (t^{(i)} - \hat{\mathbf{w}}^\top \mathbf{x}^{(i)}) \end{aligned}$$

Das duale Problem lässt sich schneller als das primale lösen, wenn die Anzahl Trainingsdatenpunkte kleiner als die Anzahl der Merkmale ist. Bedeutender ist aber, dass es den Kerneltrick ermöglicht, was mit dem primalen nicht funktioniert. Was also ist dieser Kerneltrick überhaupt?

Kernel-SVM

Nehmen wir an, Sie möchten mit zweidimensionalen Trainingsdaten (wie dem Datensatz *moons*) eine polynomiale Transformation 2. Ordnung durchführen und anschließend einen linearen SVM-Klassifikator auf den transformierten Daten trainieren. [Formel 5-8](#) zeigt die zu verwendende polynomiale Zuordnungsfunktion 2. Grades ϕ .

Formel 5-8: Polynomiale Zuordnung 2. Grades

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{pmatrix}$$

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 polynomiale Zuordnung 2. Grades anwenden und anschließend das Skalarprodukt⁷ der transformierten Vektoren berechnen (siehe [Formel 5-9](#)).

Formel 5-9: Kerneltrick bei einer polynomiellen Zuordnung 2. Grades

$$\begin{aligned} \phi(\mathbf{a})^\top \phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2} a_1 a_2 \\ a_2^2 \end{pmatrix}^\top \begin{pmatrix} b_1^2 \\ \sqrt{2} b_1 b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 \\ &= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^\top \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^\top \mathbf{b})^2 \end{aligned}$$

Was halten Sie davon? Das Skalarprodukt der transformierten Vektoren entspricht dem Quadrat des Skalarprodukts der ursprünglichen Vektoren: $\phi(\mathbf{a})^\top \phi(\mathbf{b}) = (\mathbf{a}^\top \mathbf{b})^2$.

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 $\phi(\mathbf{x}^{(i)})^\top \phi(\mathbf{x}^{(j)})$. Wenn aber ϕ die in [Formel 5-8](#) definierte polynomielle Transformation 2. Ordnung ist, dann können Sie dieses Skalarprodukt der transformierten Vektoren einfach durch $(\mathbf{x}^{(i)\top} \mathbf{x}^{(j)})^2$ 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.

Die Funktion $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^\top \mathbf{b})^2$ nennt man einen *polynomiellen Kernel* 2. Grades. Im Machine Learning versteht man unter einem *Kernel* eine Funktion, mit der sich das Skalarprodukt $\phi(\mathbf{a})^\top \phi(\mathbf{b})$ 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

Linear:	$K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\top \mathbf{b}$
Polynomiell:	$K(\mathbf{a}, \mathbf{b}) = (y\mathbf{a}^\top \mathbf{b} + r)^d$
Gaußsche RBF:	$K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \ \mathbf{a} - \mathbf{b}\ ^2)$
Sigmoid:	$K(\mathbf{a}, \mathbf{b}) = \tanh(y\mathbf{a}^\top \mathbf{b} + r)$

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: $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^\top \phi(\mathbf{b})$. Damit können Sie K als Kernel einsetzen, da Sie ja wissen, dass ϕ existiert, selbst wenn Sie ϕ nicht genau kennen. Im Fall 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 Fall eines linearen SVM-Klassifikators von einer dualen Lösung zur primalen Lösung gelangen. Wenn Sie aber den Kerneltrick 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 Kerneltrick einsetzbar ([Formel 5-11](#)).

Formel 5-11: Vorhersagen mit einer Kernel-SVM treffen

$$\begin{aligned}
h_{\widehat{\mathbf{w}}, \widehat{b}}(\phi(\mathbf{x}^{(n)})) &= \widehat{\mathbf{w}}^\top \phi(\mathbf{x}^{(n)}) + \widehat{b} = \left(\sum_{i=1}^m \widehat{\alpha}^{(i)} t^{(i)} \phi(\mathbf{x}^{(i)}) \right)^\top \phi(\mathbf{x}^{(n)}) + \widehat{b} \\
&= \sum_{i=1}^m \widehat{\alpha}^{(i)} t^{(i)} \left(\phi(\mathbf{x}^{(i)})^\top \phi(\mathbf{x}^{(n)}) \right) + \widehat{b} \\
&= \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \widehat{\alpha}^{(i)} t^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \widehat{b}
\end{aligned}$$

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 \widehat{b} mit dem gleichen Trick berechnen ([Formel 5-12](#)).

Formel 5-12: Berechnen des Bias-Terms mithilfe des Kerneltricks

$$\begin{aligned}
\widehat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m (t^{(i)} - \widehat{\mathbf{w}}^\top \phi(\mathbf{x}^{(i)})) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \left(\sum_{j=1}^m \widehat{\alpha}^{(j)} t^{(j)} \phi(\mathbf{x}^{(j)}) \right)^\top \phi(\mathbf{x}^{(i)}) \right) \\
&= \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \sum_{\substack{j=1 \\ \widehat{\alpha}^{(j)} > 0}}^m \widehat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \right)
\end{aligned}$$

Sollten Sie hiervon jetzt Kopfschmerzen bekommen haben, ist das völlig normal: Dies ist eine Nebenwirkung des Kerneltricks.

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

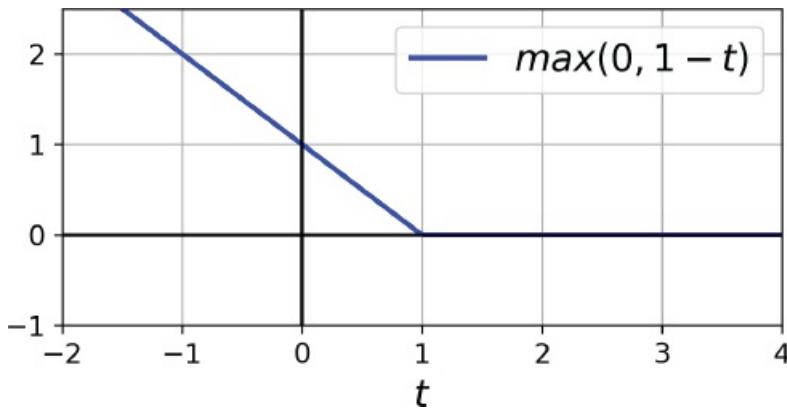
$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^\top \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b))$$

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 Margins. Die Verletzung des Margins ist für einen Datenpunkt dann gleich 0, wenn er abseits der Straße und auf der richtigen Seite liegt, andernfalls 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 »[Lasso-Regression](#)« auf [Seite 139](#)) 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« (<https://homl.info/17>)⁸ oder »Fast Kernel Classifiers with Online and Active Learning« (<https://homl.info/18>)⁹. Diese sind allerdings in Matlab und C++ implementiert. Für größere nichtlineare Aufgaben sollten Sie stattdessen neuronale Netze in Betracht ziehen (siehe [Teil II](#)).

Übungen

1. Was ist die den 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 (**H**, **f**, **A** und **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-the-Rest-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](#).

Entscheidungsbäume

Wie SVMs sind auch *Entscheidungsbäume* sehr flexible Machine-Learning-Algorithmen, die sich sowohl für Klassifikations- als auch für 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 Sie einige Einschränkungen von Entscheidungsbäumen kennenlernen.

Trainieren und Visualisieren eines Entscheidungsbaums

Um Entscheidungsbäume zu verstehen, wollen 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.

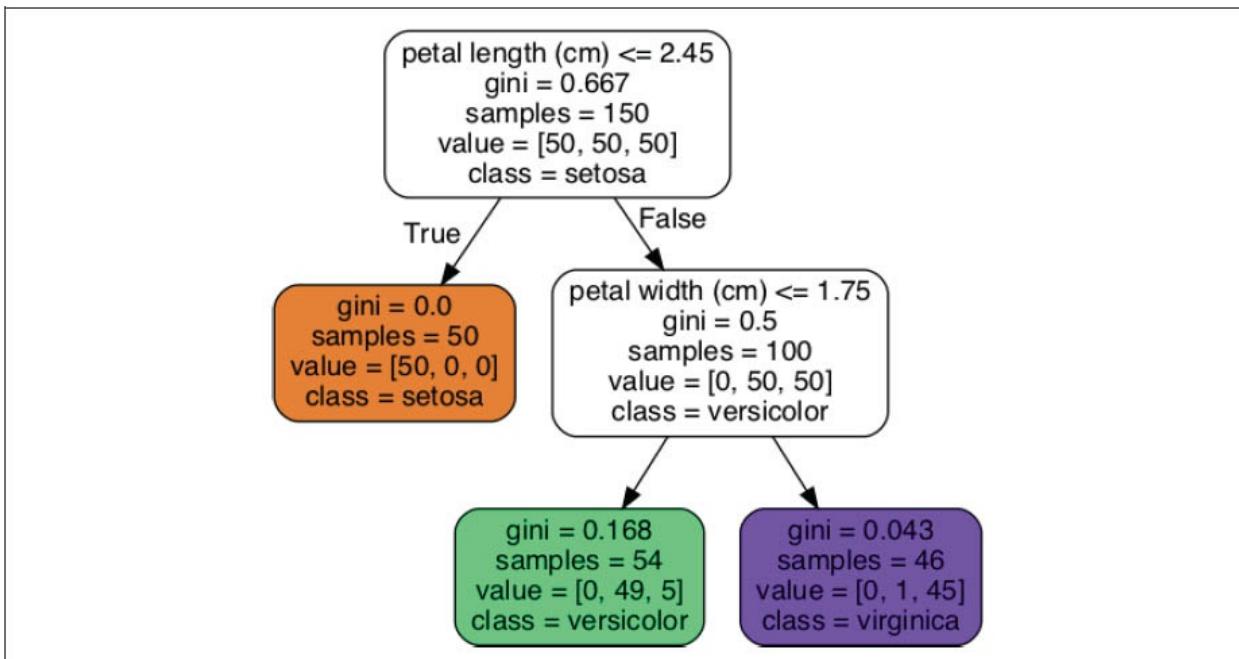


Abbildung 6-1: Iris-Entscheidungsbaum

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 die Frage, ob das Kronblatt der Blüte kürzer als 2,45 cm ist. Ist das 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 von *Iris versicolor* und 45 von *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 . Knoten berechnet. Beispielsweise ist der `gini`-Score für den linken Knoten bei depth 2 gleich $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0,168$.

Formel 6-1: Gini-Unreinheit

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

Hier gilt:

- $p_{i,k}$ ist 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).

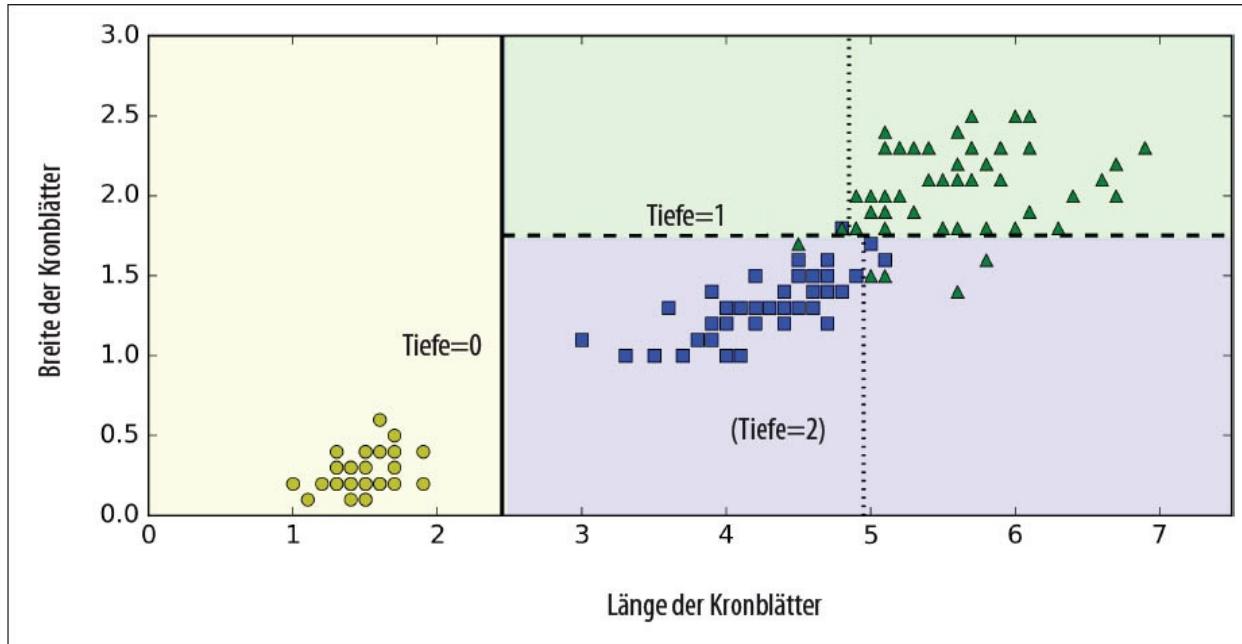


Abbildung 6-2: Entscheidungsgrenzen in einem Entscheidungsbaum

Interpretation von Modellen: White Box im Vergleich zu Black Box

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 und neuronale Netze im Allgemeinen zu den *Black-Box-Modellen*. Sie treffen zwar ausgezeichnete Vorhersagen, und Sie können die dazu durchgeführten Berechnungen leicht nachprüfen, dennoch 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-Trainingsalgorithmus

Scikit-Learn verwendet den Algorithmus *Classification and Regression Tree* (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 $\leq 2,45$ 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 ist 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 verlangsamt dies das Trainieren aber deutlich.

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 . Knotens. Beispielsweise beträgt die Entropie für den linken Knoten bei depth 2 in [Abbildung 6-1](#) genau $-\frac{49}{54} \log_2\left(\frac{49}{54}\right) - \frac{5}{54} \log_2\left(\frac{5}{54}\right) \approx 0,445$.

Formel 6-3: Entropie

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^n p_{i,k} \log_2(p_{i,k})$$

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 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 überlassen, 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 Entscheidungsbäums 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 Entscheidungsbäums 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 Entscheidungsbäums 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 werden gelöscht. Dieses Pruning wird fortgesetzt, bis alle überflüssigen Knoten entfernt worden sind.

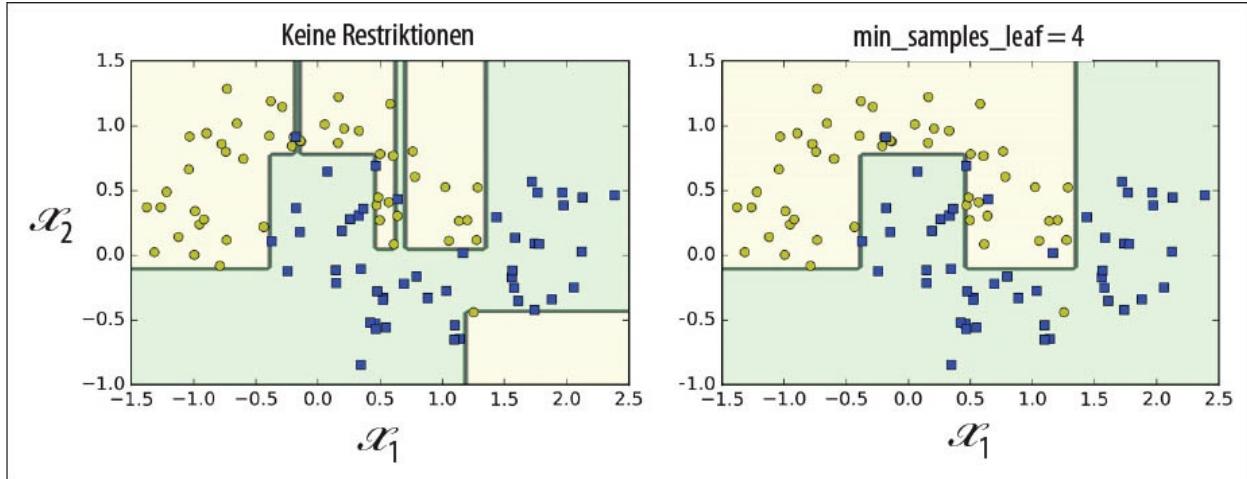


Abbildung 6-3: Regularisierung mit `min_samples_leaf`

Abbildung 6-3 zeigt zwei auf dem Datensatz `moons` (aus Kapitel 5) trainierten Entscheidungsbäume. Der Entscheidungsbau 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.

Regression

Entscheidungsbäume können auch Regressionsaufgaben bewältigen. Erstellen wir mit der Klasse `DecisionTreeRegressor` aus Scikit-Learn einen Regressionsbaum und trainieren wir 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.

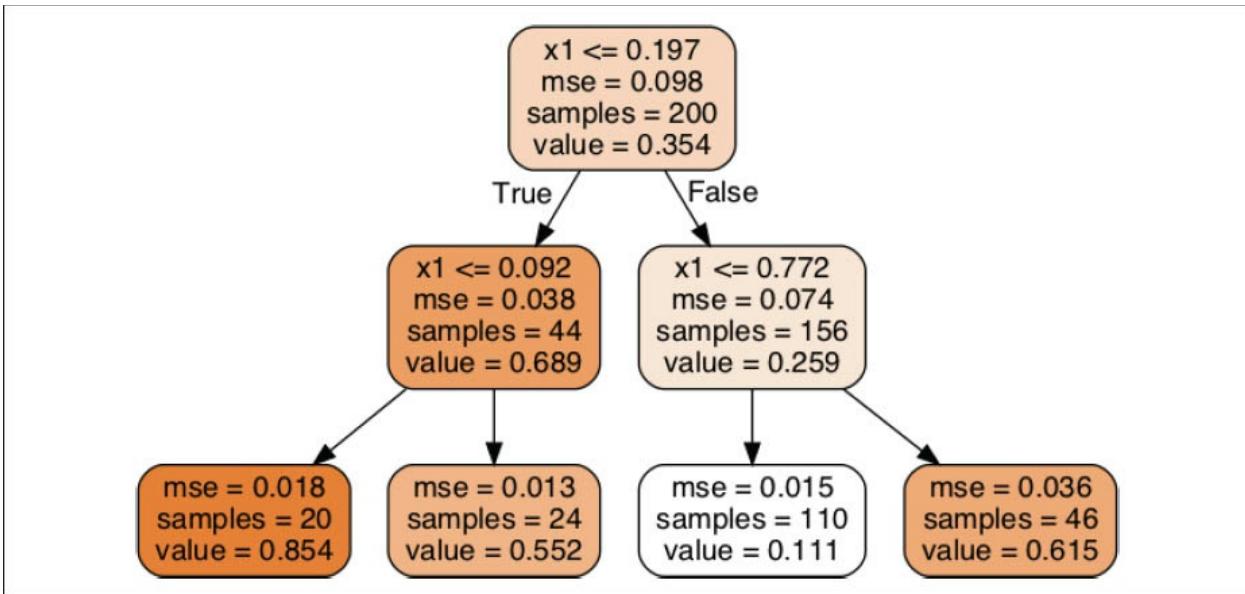


Abbildung 6-4: Ein Entscheidungsbaum zur Regression

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 $\text{value}=0.111$. Diese Vorhersage ist der durchschnittliche Zielwert der 110 Trainingsdatenpunkte in diesem Blatt. Sie führt zu einem mittleren quadratischen Fehler (MSE) von 0,015 in den 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.

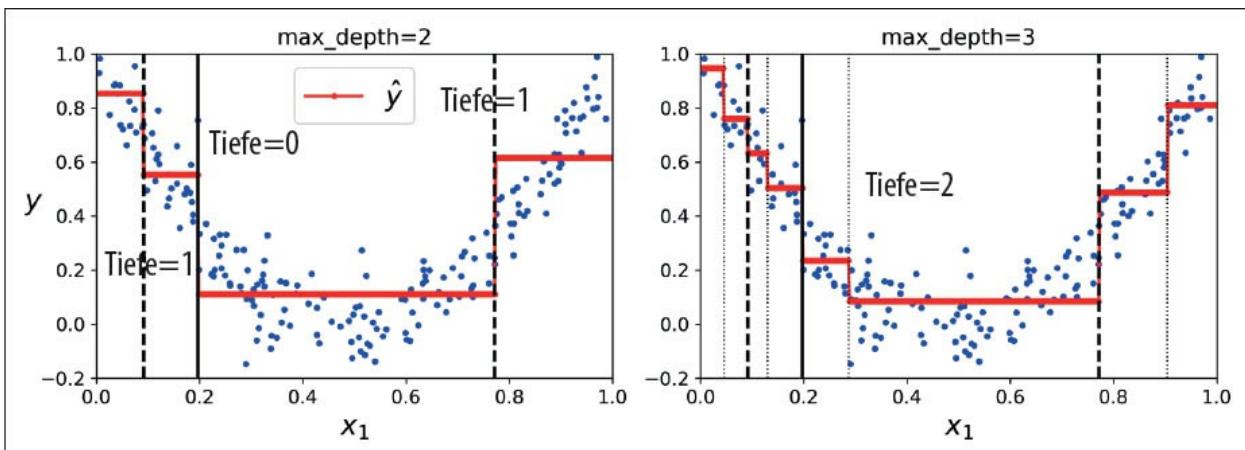


Abbildung 6-5: Vorhersagen zweier Entscheidungsbäume zur Regression

Der CART-Algorithmus funktioniert weitgehend wie oben vorgestellt. Der einzige Unterschied besteht darin, dass der Trainingsdatensatz so aufgeteilt wird, dass der MSE statt der Unreinheit minimiert wird. [Formel 6-4](#) zeigt die vom Algorithmus minimierte Kostenfunktion.

Formel 6-4: CART-Kostenfunktion für die Regression

$$J(k, t_k) = \frac{m_{\text{links}}}{m} \text{MSE}_{\text{links}} + \frac{m_{\text{rechts}}}{m} \text{MSE}_{\text{rechts}} \quad \text{wobei} \quad \begin{cases} \text{MSE}_{\text{Knoten}} = \sum_{i \in \text{Knoten}} (\hat{y}_{\text{Knoten}} - y^{(i)})^2 \\ \hat{y}_{\text{Knoten}} = \frac{1}{m_{\text{Knoten}}} \sum_{i \in \text{Knoten}} y^{(i)} \end{cases}$$

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 sinnvolleres Modell, das Sie auf der rechten Seite von [Abbildung 6-6](#) sehen.

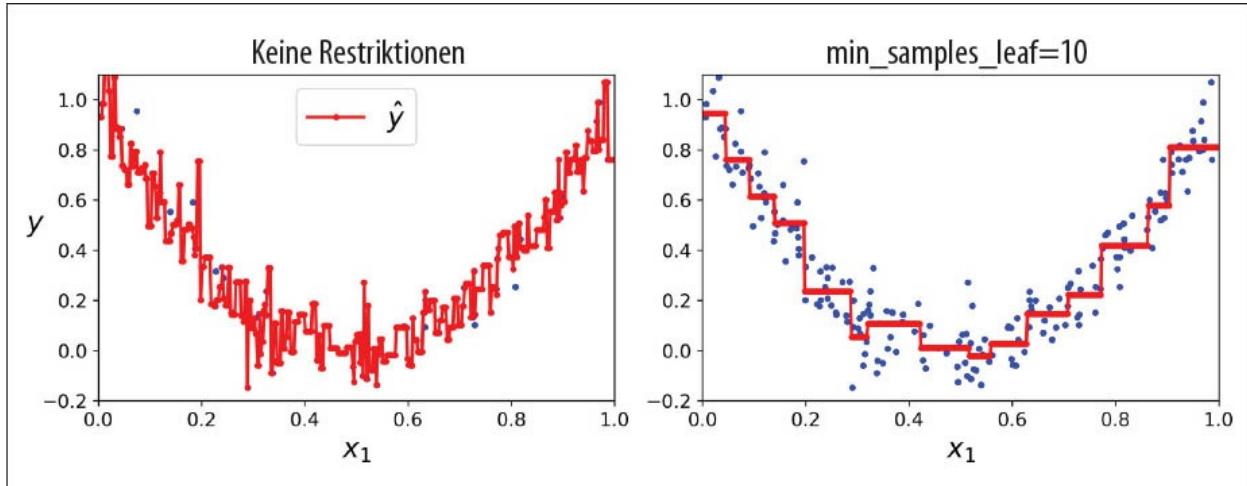


Abbildung 6-6: Regularisierung eines Regressionsbaums

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 diesen einfach 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.

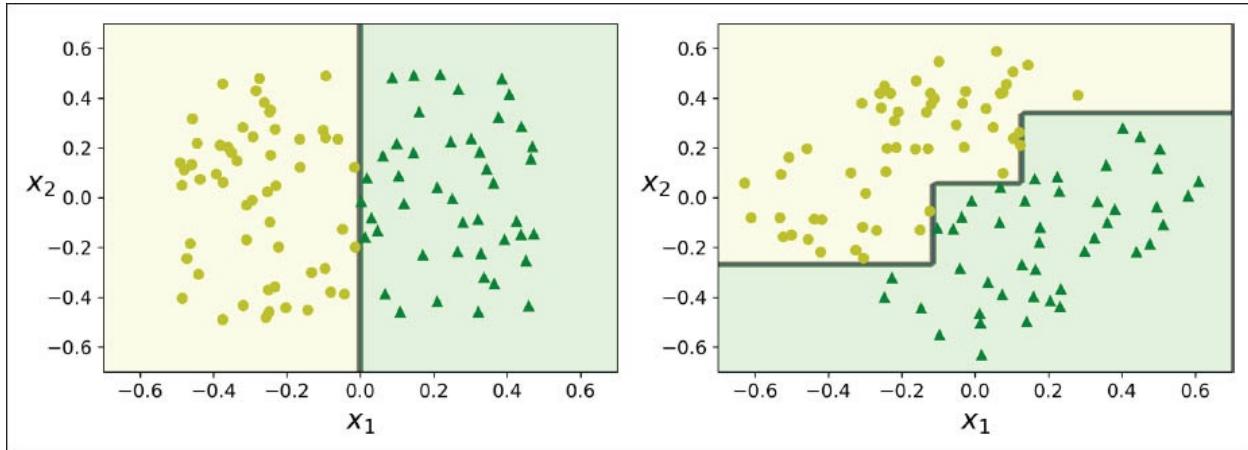


Abbildung 6-7: Empfindlichkeit für Rotation der Trainingsdaten

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 Entscheidungsbäum trainieren, erhalten Sie das in Abbildung 6-8 gezeigte Modell. Wie Sie sehen, unterscheidet es sich sehr stark vom vorherigen Entscheidungsbäum (siehe 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`).

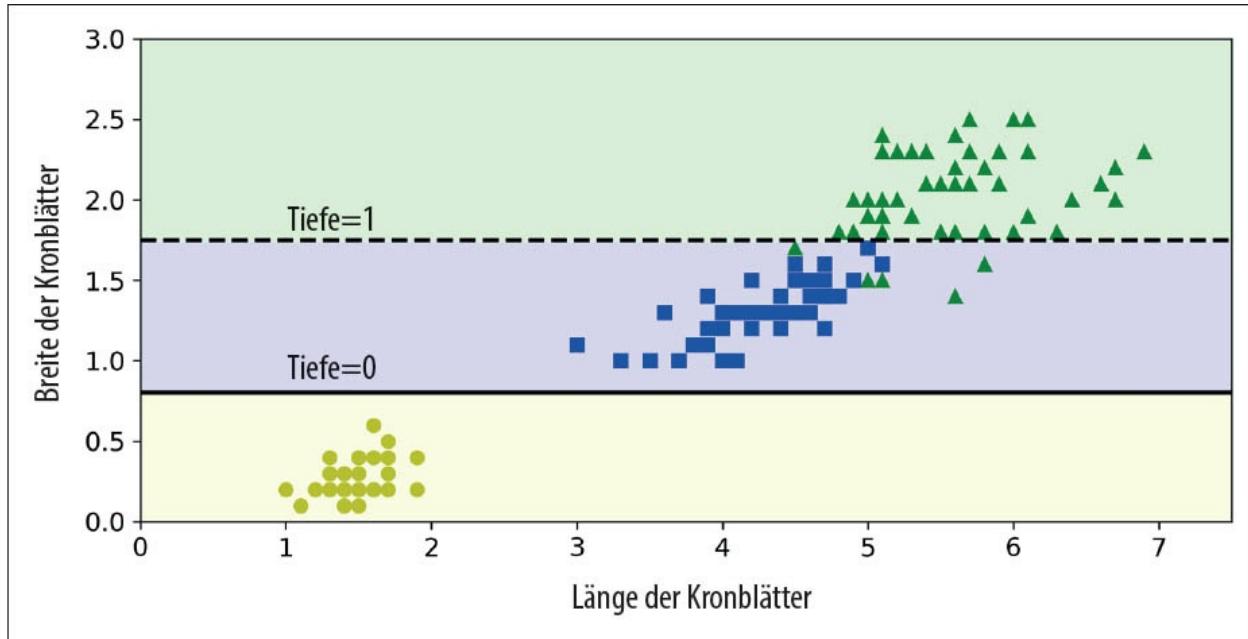


Abbildung 6-8: Anfälligkeit für Einzelheiten der Trainingsdaten

Random Forests wirken dieser Instabilität entgegen, indem sie die Vorhersagen vieler Bäume

mitteln, wie wir im nächsten Kapitel sehen werden.

Übungen

1. Was ist die ungefähre Tiefe eines mit 1 Million Datenpunkten (ohne Restriktionen) trainierten Entscheidungsbaums?
2. 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?
3. Sollte man versuchen, `max_depth` zu senken, wenn ein Entscheidungsbaum einen Trainingsdatensatz overfittet?
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 100.000 Datenpunkten besteht, beschleunigt das Setzen von `presort=True` das Trainieren?
7. Trainieren und optimieren Sie einen Entscheidungsbaum für den Datensatz `moons` anhand dieser Schritte.
 - 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 mit diesen Schritten.
 - a. Erzeugen Sie im Anschluss an die vorherige Aufgabe 1.000 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 1.000 Entscheidungsbäume auf den Testdaten aus. Da sie mit kleineren Datensätzen trainiert wurden, schneiden diese Bäume mit nur ca. 80% voraussichtlich schlechter ab als der erste Entscheidungsbaum.
 - c. Nun kommt der Zaubertrick. Generieren Sie für jeden Testdatenpunkt die Vorhersagen der 1.000 Entscheidungsbäume und heben Sie ausschließlich die häufigste Vorhersage

auf (Sie können dazu die Funktion `mode()` aus Sci-Py 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 wir einmal an, Sie würden Tausenden zufällig ausgewählten Menschen 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*.

Als Beispiel für eine Ensemble-Methode können Sie 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* und *Stacking*. 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 knächste-Nachbarn-Klassifikator und vielleicht noch weitere (siehe [Abbildung 7-1](#)).

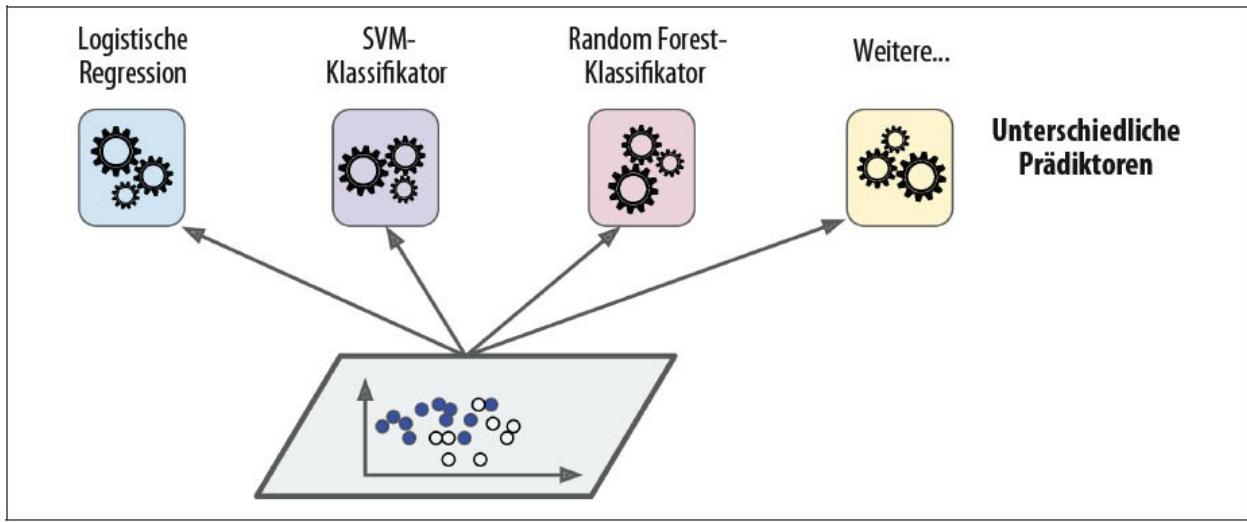


Abbildung 7-1: Trainieren unterschiedlicher Klassifikatoren

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).

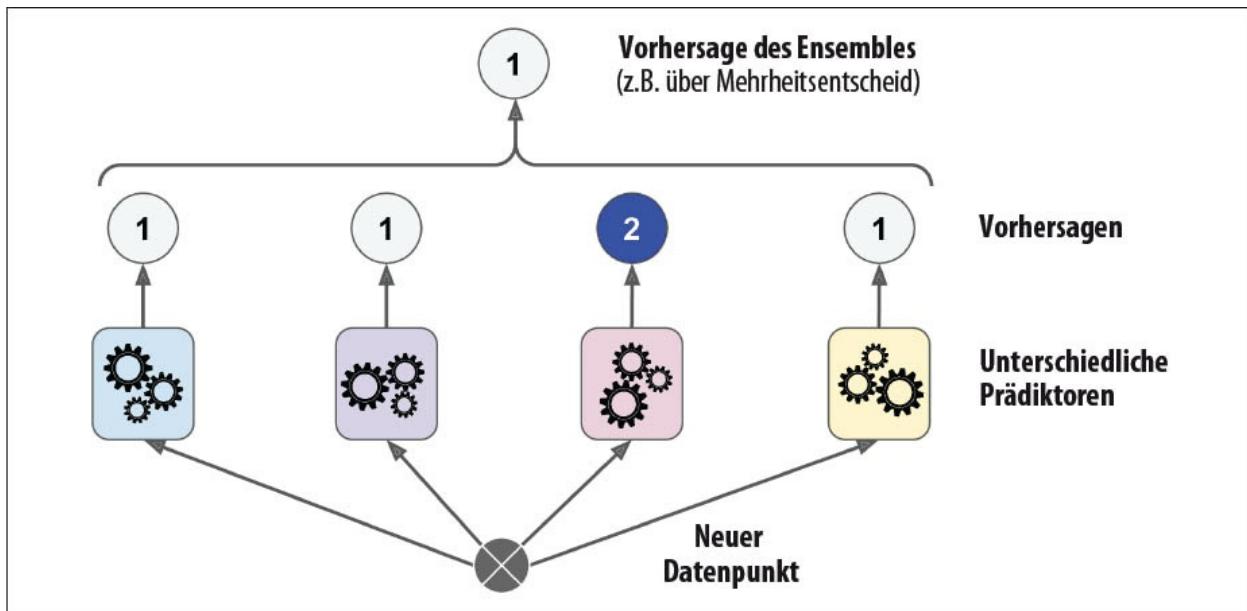


Abbildung 7-2: Vorhersagen eines Hard-Voting-Klassifikators

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 das möglich? Vielleicht kann die folgende Analogie dieses Rätsel ein wenig erhellen. 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 1.000 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. steigt sie mit 10.000 Würfen 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 unausgewogenen Münzwürfe. Sie können dort sehen, wie sich der Anteil von Kopf mit steigender Anzahl Würfe 51% annähert. Irgendwann nähern sich alle zehn Serien so nah an 51% an, dass sie kontinuierlich über 50% liegen.

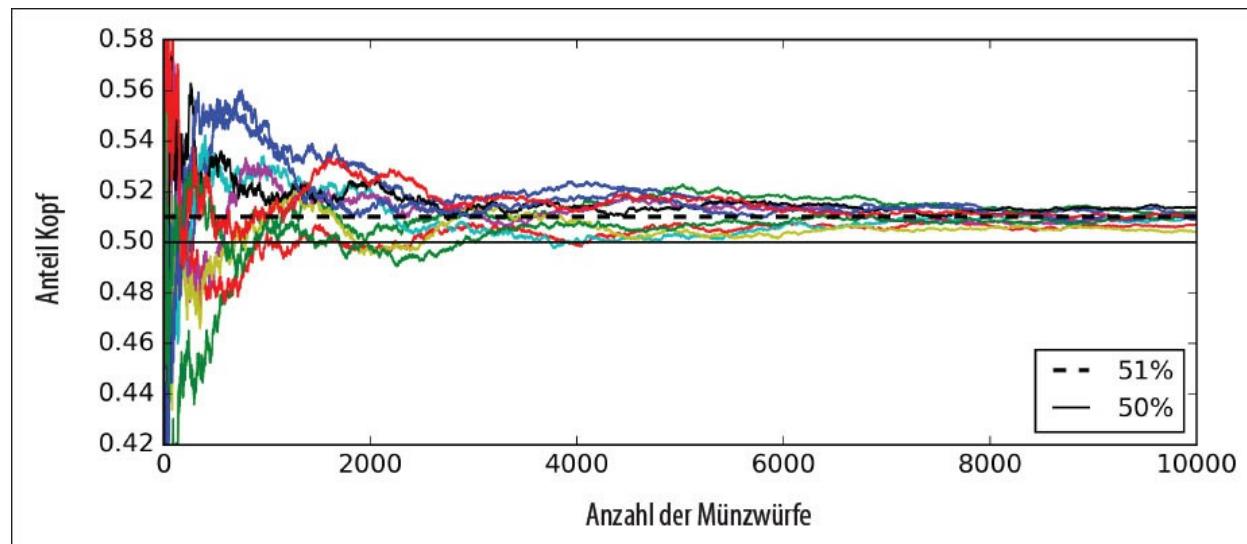


Abbildung 7-3: Das Gesetz der großen Zahl

In ähnlicher Weise können Sie ein Ensemble aus 1.000 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 das nicht gegeben, 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 0.896
SVC 0.888
VotingClassifier 0.904
```

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), 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 beigemessen wird. Sie müssen dazu lediglich `voting="hard"` durch `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 jedoch 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* (<https://homl.info/20>)¹ (eine Kurzform von *Bootstrap-Aggregation*²). Werden die Teilmengen *ohne* Zurücklegen gebildet, bezeichnet man dies als *Pasting* (<https://homl.info/21>).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 wird dieser Vorgang dargestellt.

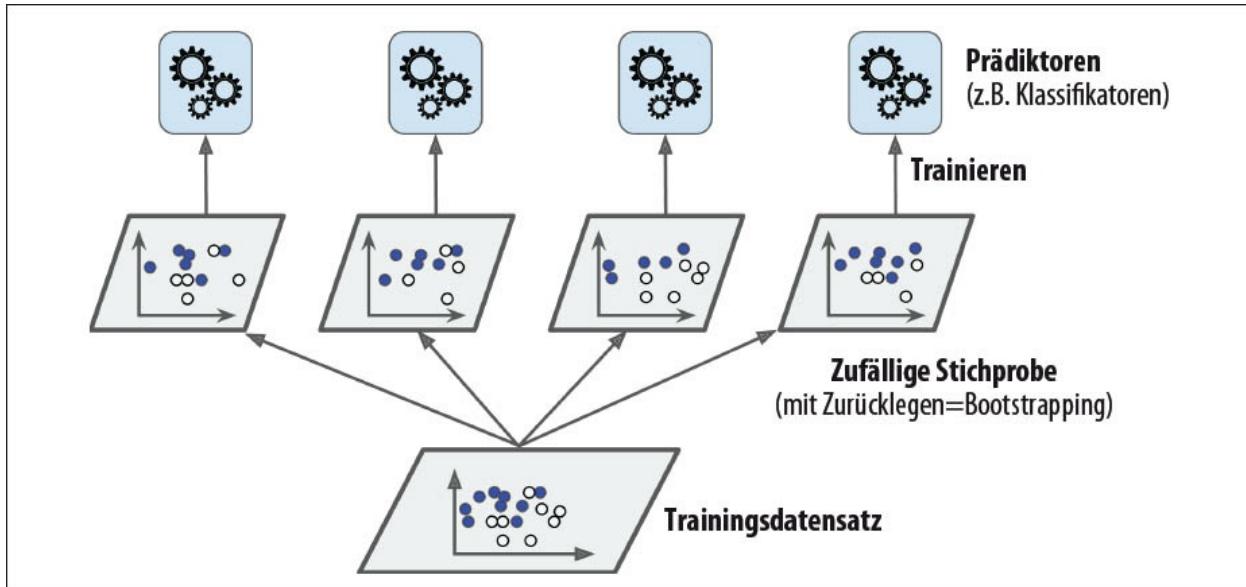


Abbildung 7-4: Zum Bagging und Pasting gehört das Trainieren diverser Prädiktoren mit verschiedenen Zufalls-Samples des Trainingsdatensatzes.

Sobald sämtliche Prädiktoren trainiert sind, 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 ähnliches 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 mit der 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 eines Hard Voting durch, falls die zugrunde liegenden Klassifikatoren Wahrscheinlichkeiten für Kategorien abschätzen können (also die Methode `predict_proba()` besitzen). Bei unseren Entscheidungsbäumen ist das der Fall.

[Abbildung 7-5](#) vergleicht die Entscheidungsgrenze eines einzelnen Entscheidungsbaums 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 Entscheidungsbaums: 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).

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.

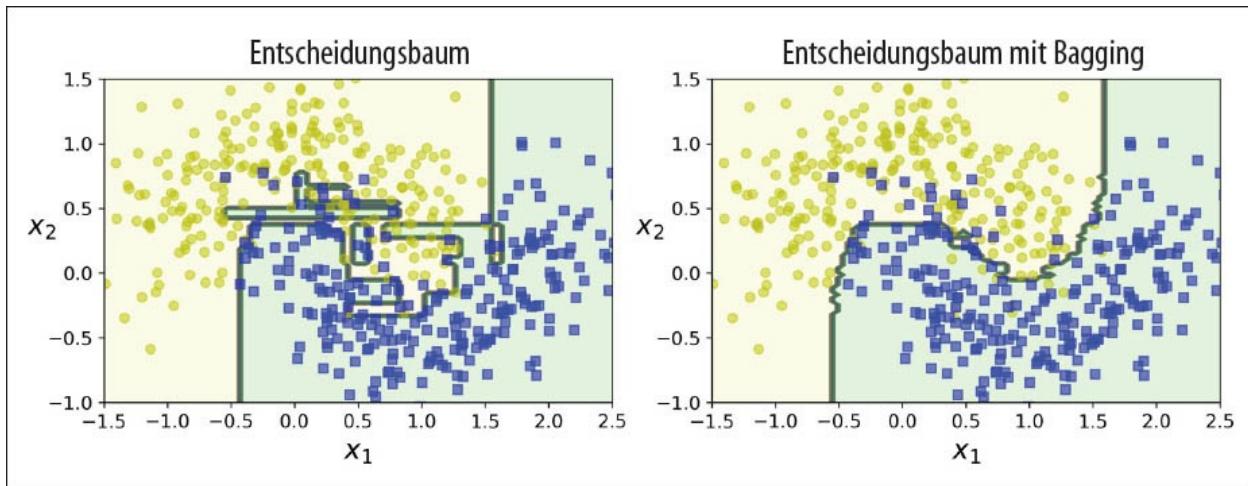


Abbildung 7-5: Ein einzelner Entscheidungsbaum (links) im Vergleich zu einem Ensemble von 500 Bäumen mit Bagging (rechts)

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 nötig wäre. 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 ziegt das folgende Codebeispiel. Der Score der dabei durchgeführten Auswertung ist

als Attribut `oob_score_` verfügbar:

```
>>> 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 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 dem 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 erste Trainingsdatenpunkt mit einer Wahrscheinlichkeit von 68,25% 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. Das 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* (<https://homl.info/22>).⁷ Werden sämtliche Trainingsdatenpunkte verwendet (z.B. mit `bootstrap=False` und `max_samples=1.0`), aber Merkmale ausgewählt (z.B. `bootstrap_features=True` und/oder `max_features` kleiner als 1,0), so nennt man dies die *Methode der zufälligen Subräume* (<https://homl.info/23>).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 (<https://homl.info/24>)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 nutzt 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:

```
bag_clf = BaggingClassifier(  
    DecisionTreeClassifier(max_features="auto", 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* (<https://homl.info/25>)¹² (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 `ExtraTrees Classifier`. 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 `RandomForestClassifier` 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 die 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](#).

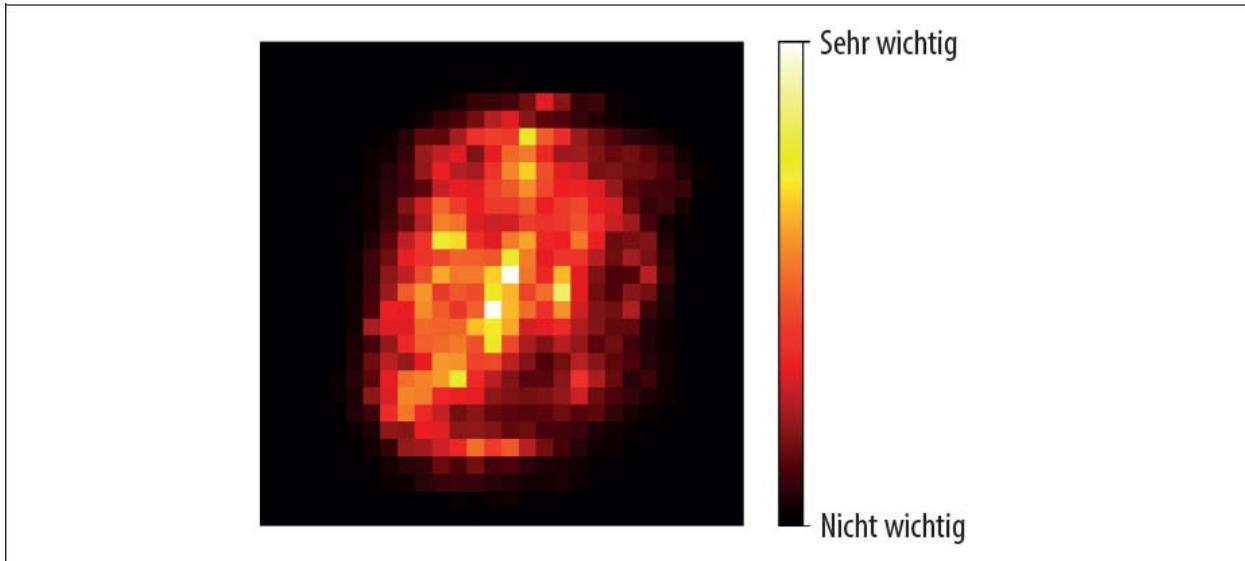


Abbildung 7-6: Wichtigkeit der Pixel in MNIST (laut einem Random-Forest-Klassifikator)

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 Prädiktoren 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* (<https://homl.info/26>)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).

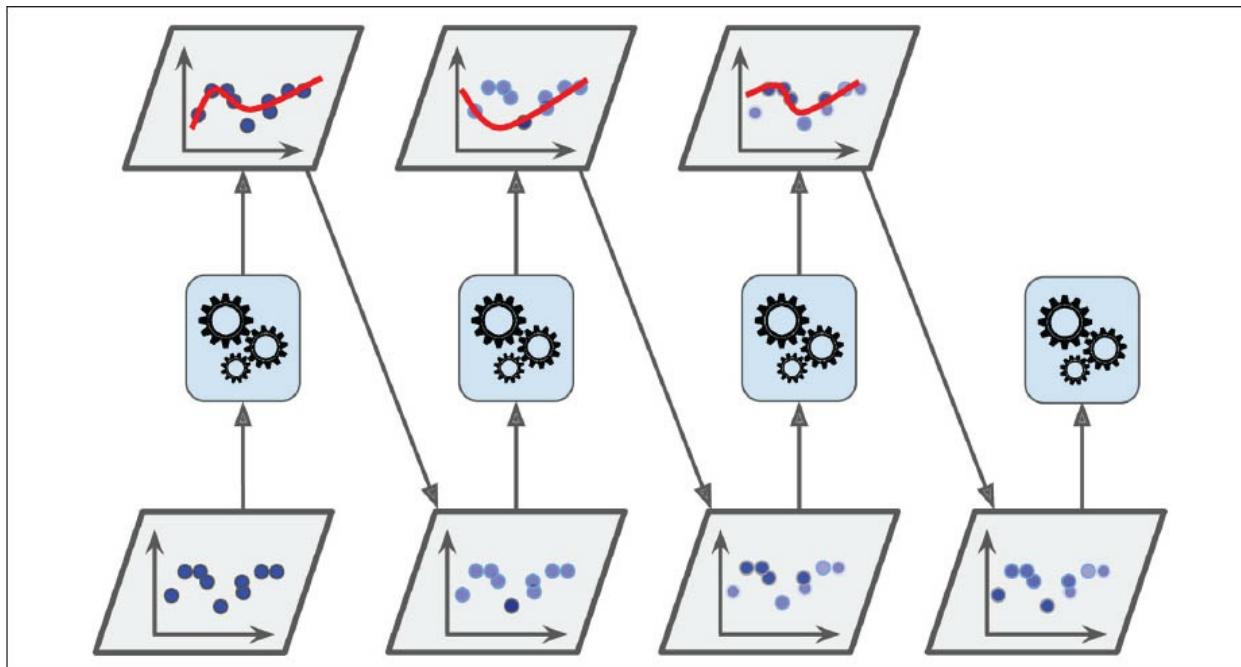


Abbildung 7-7: Sequenzielles Training mit aktualisierten Gewichten der Datenpunkte bei AdaBoost

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 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.

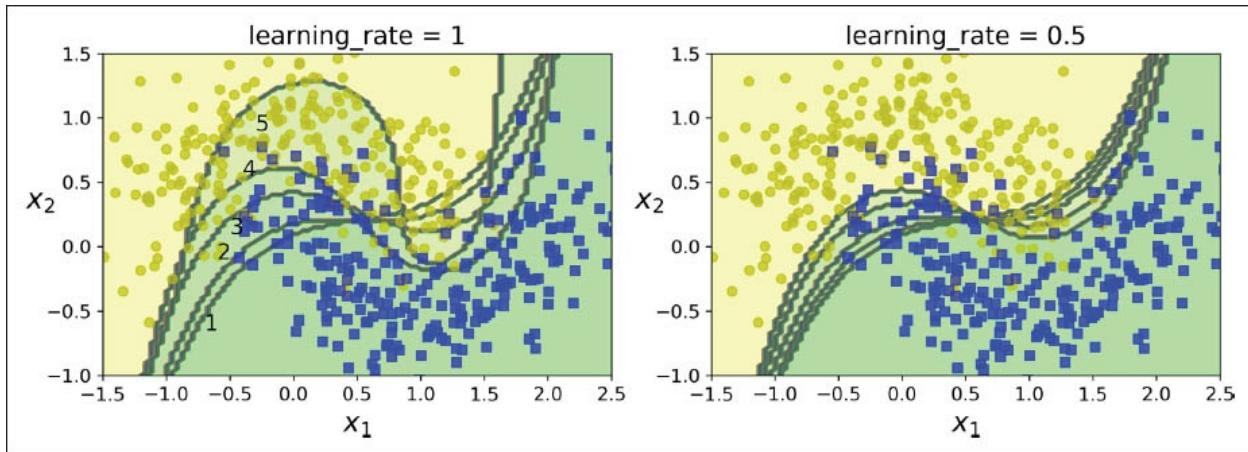


Abbildung 7-8: Entscheidungsgrenzen aufeinanderfolgender Prädiktoren

Sobald sämtliche Prädiktoren trainiert sind, 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 vorherige 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 $1/m$ 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. Prädiktor

$$r_j = \frac{\sum_{i=1}^m w^{(i)}}{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^m w^{(i)}}$$

wobei $\hat{y}_j^{(i)}$ die Vorhersage des j . Prädiktors für den i . 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 $i = 1, 2, \dots, m$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{wenn } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{wenn } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Danach 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

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{j=1}^N \alpha_j \text{ wobei } N \text{ die Anzahl Prädiktoren ist.}$$

$\hat{y}_j(\mathbf{x})=k$

Scikit-Learn verwendet eine für mehrere Kategorien geeignete Version von Ada-Boost namens *SAMME* (<https://homl.info/27>)¹⁶, was für *Stagewise Additive Modeling 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* (<https://homl.info/28>).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 vorherigen 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)
```

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.

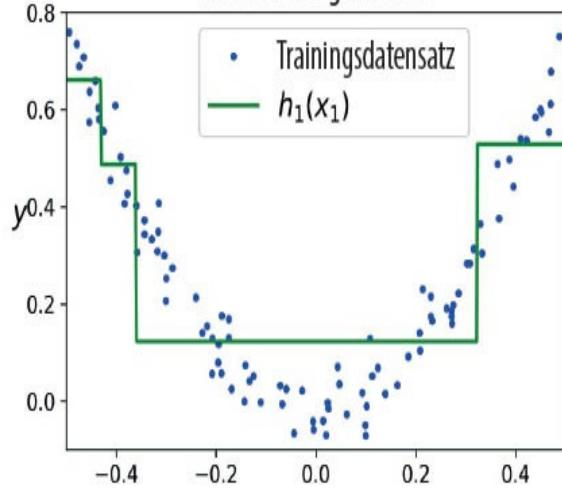
Ein GBRT-Ensemble lässt sich einfacher mit der Klasse `GradientBoostingRegressor` in Scikit-Learn trainieren. Wie die Klasse `RandomForestRegressor` enthält auch diese Hyperparameter, die das Wachstum der Entscheidungsbäume steuern (z.B. `max_depth` oder `min_samples_leaf`), und Hyperparameter, die das 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.

Restfehler und Vorhersagen der Entscheidungsbäume



Vorhersagen des Ensembles

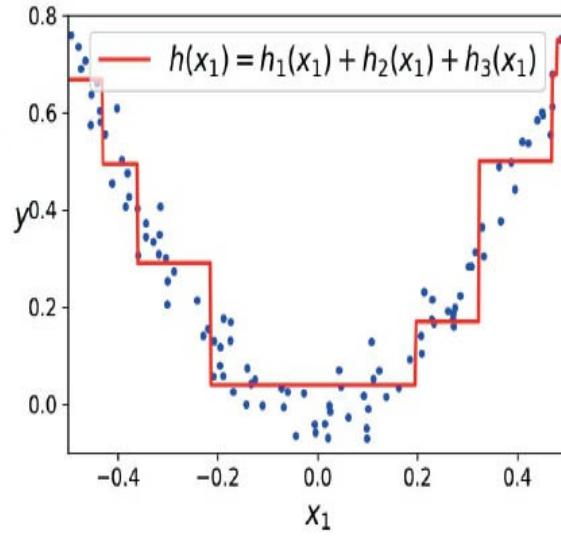
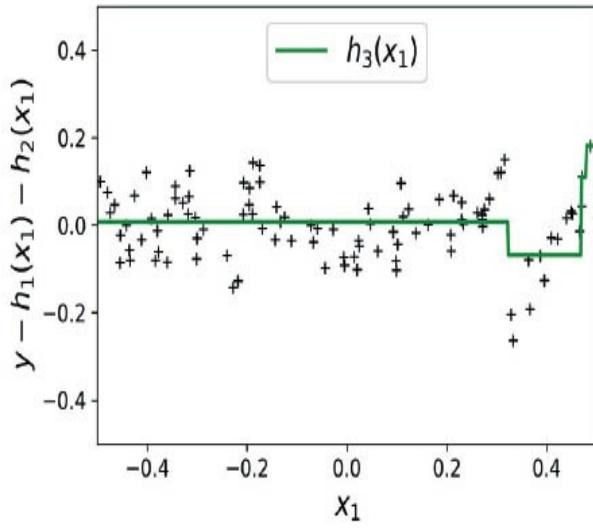
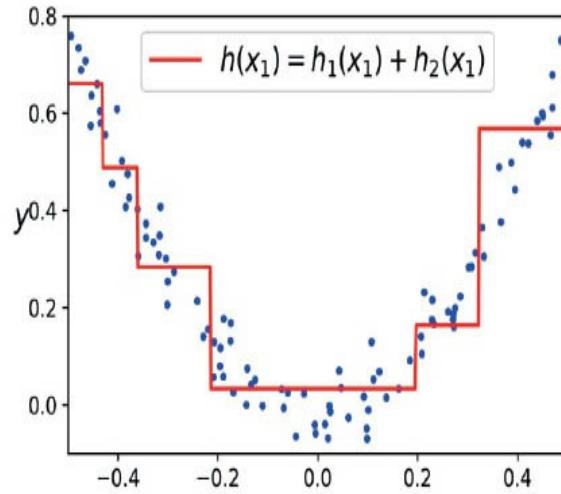
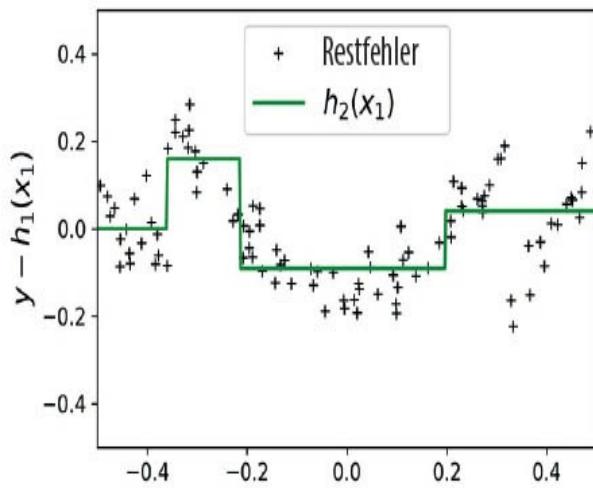
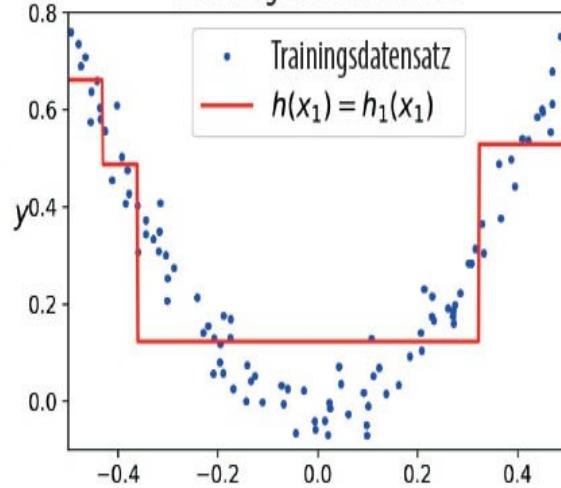


Abbildung 7-9: In dieser Darstellung des Gradient Boosting ist der erste Prädiktor (oben links) normal trainiert, jeder folgende Prädiktor (Mitte und unten links) wird dann mit dem Restfehler des vorherigen Prädiktors trainiert – die rechte Spalte zeigt die Vorhersagen der daraus entstandenen Ensembles.

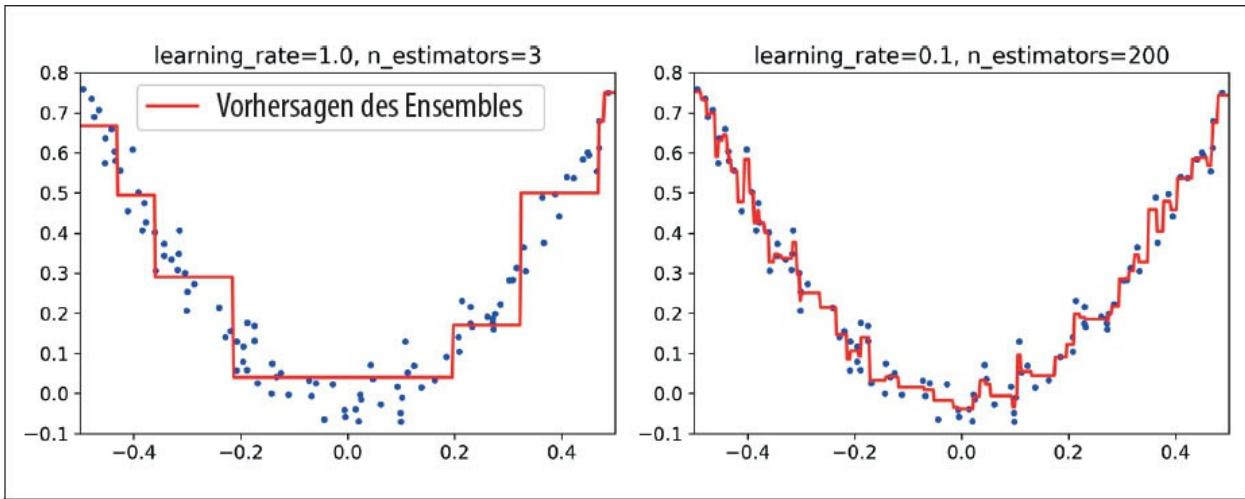


Abbildung 7-10: GBRT-Ensembles mit zu wenigen (links) und zu vielen Prädiktoren (rechts)

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) + 1

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 verfrüht 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()` wiederverwendet. Der folgende Code beendet das Training, sobald sich der Validierungsfehler in fünf aufeinanderfolgenden Iterationen nicht mehr 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
```

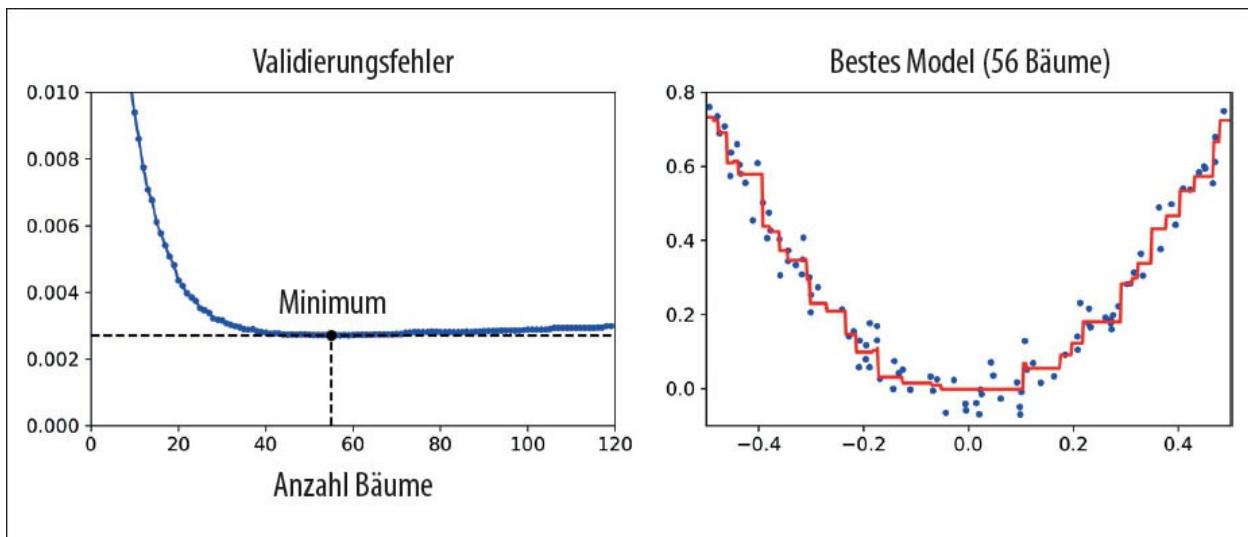


Abbildung 7-11: Optimieren der Anzahl Bäume durch Early Stopping

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 finden Sie in der Dokumentation von Scikit-Learn).

Es lohnt sich, darauf hinzuweisen, dass es eine optimierte Implementierung des Gradient Boosting in der beliebten Python-Bibliothek XGBoost (<https://github.com/dmlc/xgboost>) gibt (was für »Extreme Gradient Boosting« steht). Dieses Paket wurde ursprünglich von Tianqi Chen als Teil der Distributed (Deep) Machine Learning Community (DMLC) entwickelt und soll vor allem außerordentlich schnell, skalierbar und portabel sein. Tatsächlich dient XGBoost oft als wichtiger Bestandteil der Gewinner in ML-Wettbewerben. Die API von XGBoost ähnelt der von Scikit-Learn sehr:

```
import xgboost

xgb_reg = xgboost.XGBRegressor()

xgb_reg.fit(X_train, y_train)

y_pred = xgb_reg.predict(X_val)
```

XGBoost bietet zudem eine Reihe netter Features, wie zum Beispiel ein automatisches frühes Stoppen:

```
xgb_reg.fit(X_train, y_train,  
            eval_set=[(X_val, y_val)], early_stopping_rounds=2)  
  
y_pred = xgb_reg.predict(X_val)
```

Sie sollten sich die Bibliothek definitiv einmal anschauen!

Stacking

Die letzte in diesem Kapitel besprochene Ensemble-Methode nennt man *Stacking* (eine Kurzform von *Stacked Generalization* (<https://homl.info/29>)).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 verwendeter 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](#)).

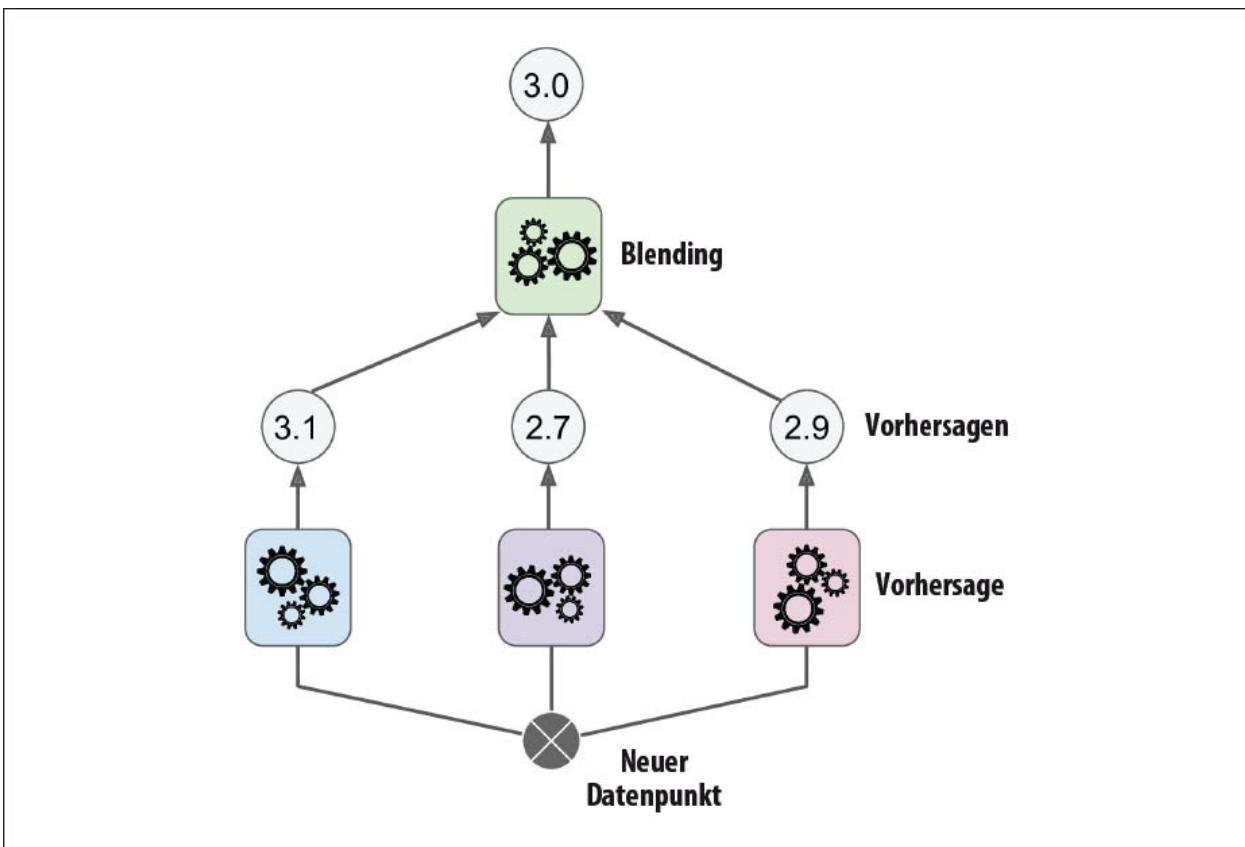


Abbildung 7-12: Aggregieren von Vorhersagen mit einem Blender-Prädiktor

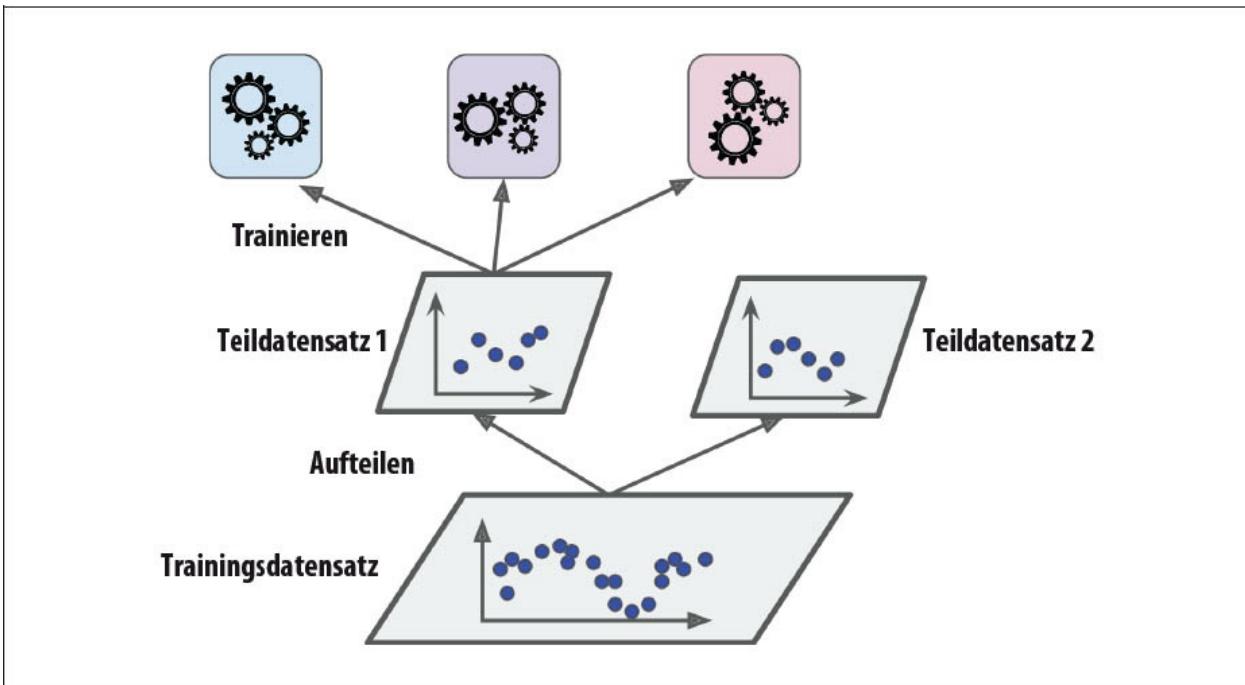


Abbildung 7-13: Trainieren der ersten Stufe

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.

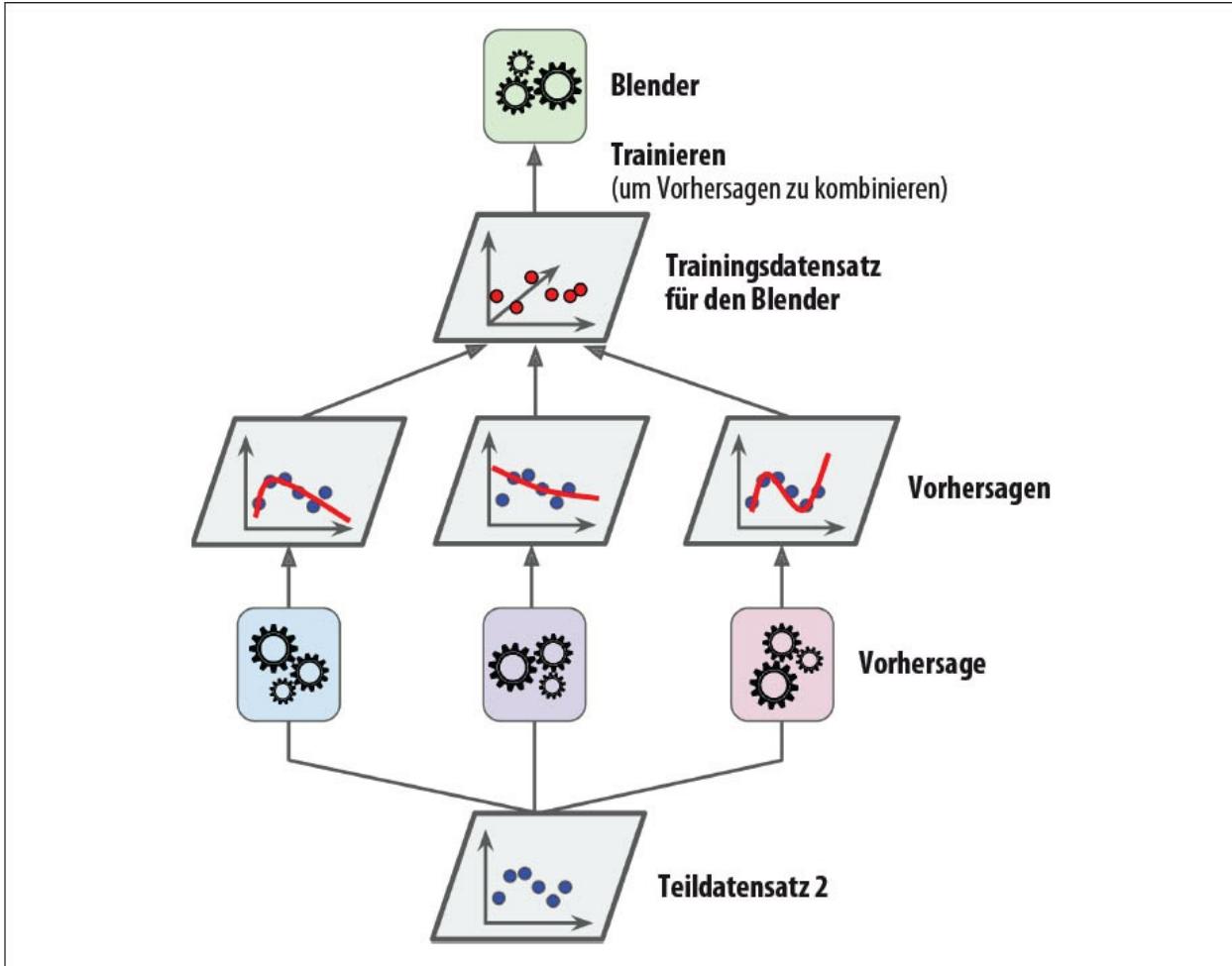


Abbildung 7-14: Trainieren eines Blenders

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 nutzen 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 oder einen zweiten als Random-Forest-Regressor): 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.

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 (<https://github.com/viisar/brew>) verwenden.

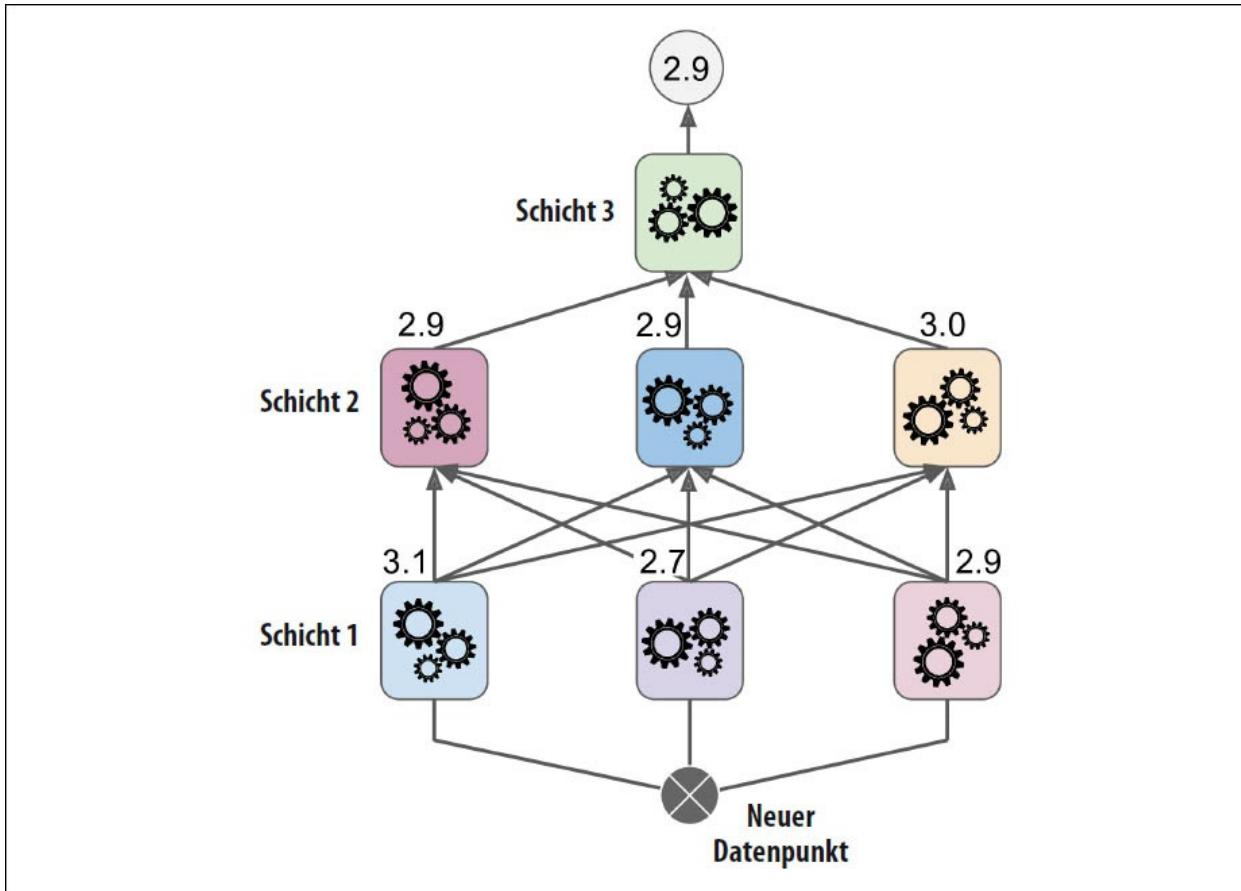


Abbildung 7-15: Vorhersagen in einem mehrschichtigen Stacking-Ensemble

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. 50.000 Datenpunkte zum Trainieren, 10.000 zur Validierung und 10.000 zum Testen). Trainieren Sie anschließend unterschiedliche Klassifikatoren, z.B. einen Random-Forest-Klassifikator, einen Extra-Trees-Klassifikator und eine 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. Trainieren Sie einen Klassifikator mit diesem neuen Trainingsdatensatz. 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?

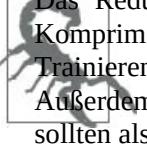
Lösungen zu diesen Aufgaben finden Sie in [Anhang A](#).

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 an 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 Fall 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.

Neben 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 einer eingedampften Darstellung eines hochdimensionalen Datensatzes als Diagramm, wodurch man häufig wichtige Einblicke in Form von visuell erkennbaren Mustern, z.B. Clustern, erhält. Zudem ist die Datenvisualisierung entscheidend für das Kommunizieren Ihrer Schlussfolgerungen an diejenigen, die keine Data Scientists sind – insbesondere Manager, die Ihre Ergebnisse nutzen.

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 gekrümmmt ist, einmal ganz zu schweigen.

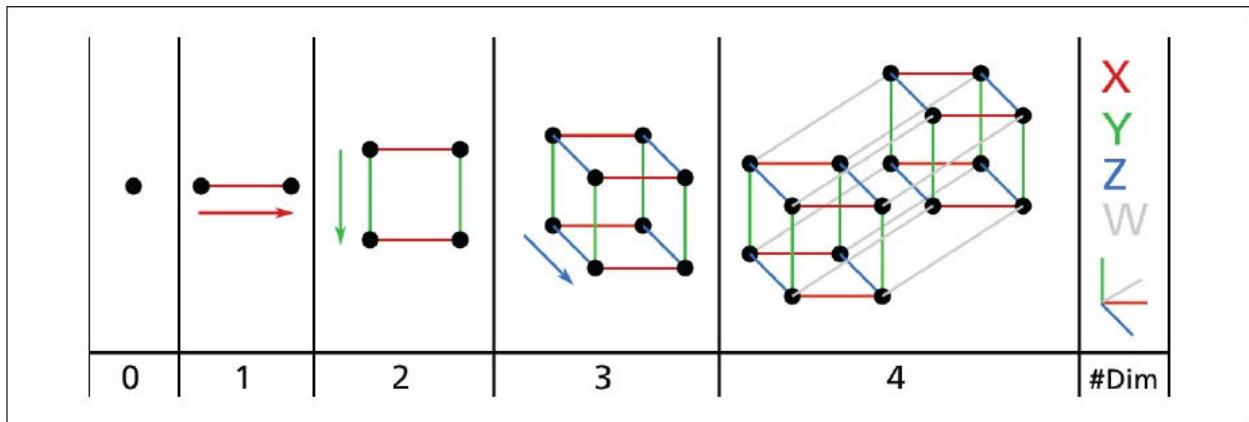


Abbildung 8-1: Punkt, Strecke, Quadrat, Würfel und Tesserakt (0-D bis 4-D- Hyperwürfel)²

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, beträgt die Wahrscheinlichkeit, näher als 0,001 an einer Kante zu sein, weniger als 0,4% (anders gesagt, es ist sehr unwahrscheinlich, dass ein zufälliger Punkt »extrem« in Bezug auf eine der Dimensionen ist). Aber in einem 10000-dimensionalen Einheitshyperwürfel steigt diese Wahrscheinlichkeit auf mehr als 99,999999% an. Die meisten Punkte in einem hochdimensionalen Hyperwürfel liegen sehr nah am Rand.³

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 $\sqrt{1000000/6}$)! Das ist wenig intuitiv: Wie können zwei Punkte so weit auseinanderliegen, wenn sie beide im gleichen Einheitshyperwürfel liegen? Nun, in höheren Dimensionen steht einfach viel Platz zur Verfügung, und damit sind hochdimensionale Datensätze tendenziell sehr dünn besetzt: 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 des Fluchs der Dimensionalität, den Trainingsdatensatz 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 auseinandersetzen, 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 eines viel niedriger dimensionalen *Subraums* des höher dimensionalen Raums (oder nahe bei). Da dies sehr abstrakt klingt, sehen wir uns ein Beispiel an. In [Abbildung 8-2](#) erkennen Sie einen durch Kreise dargestellten 3-D-Datensatz.

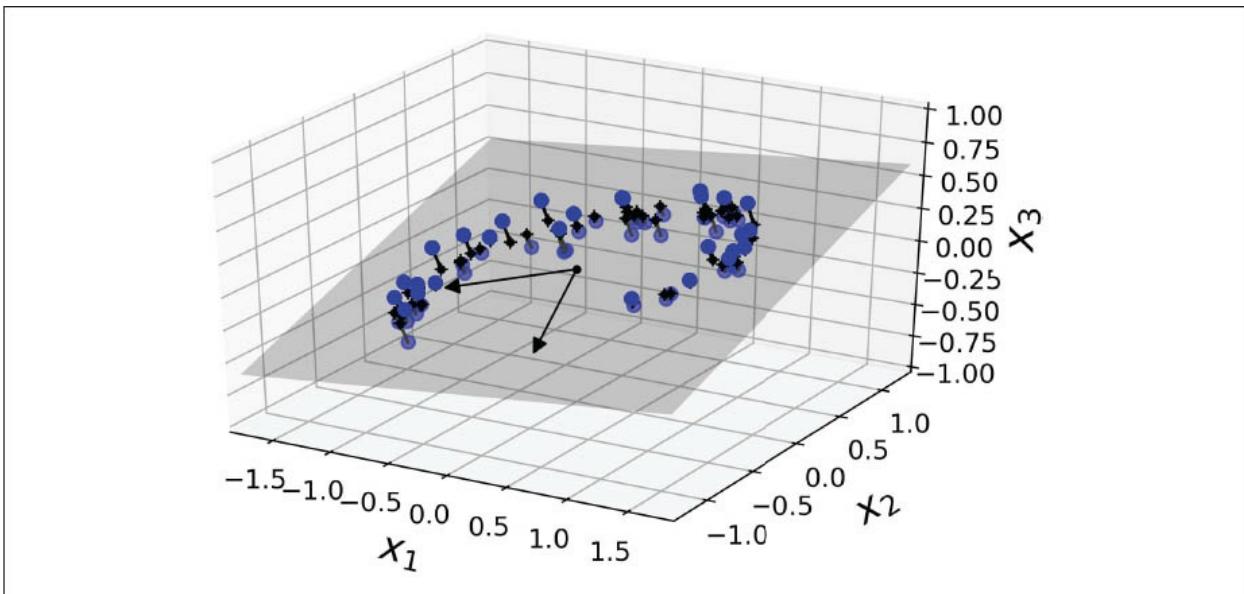


Abbildung 8-2: Ein 3-D-Datensatz in der Nähe eines 2-D-Subraums

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 den in [Abbildung 8-3](#) dargestellten neuen Datensatz. Tada! 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).

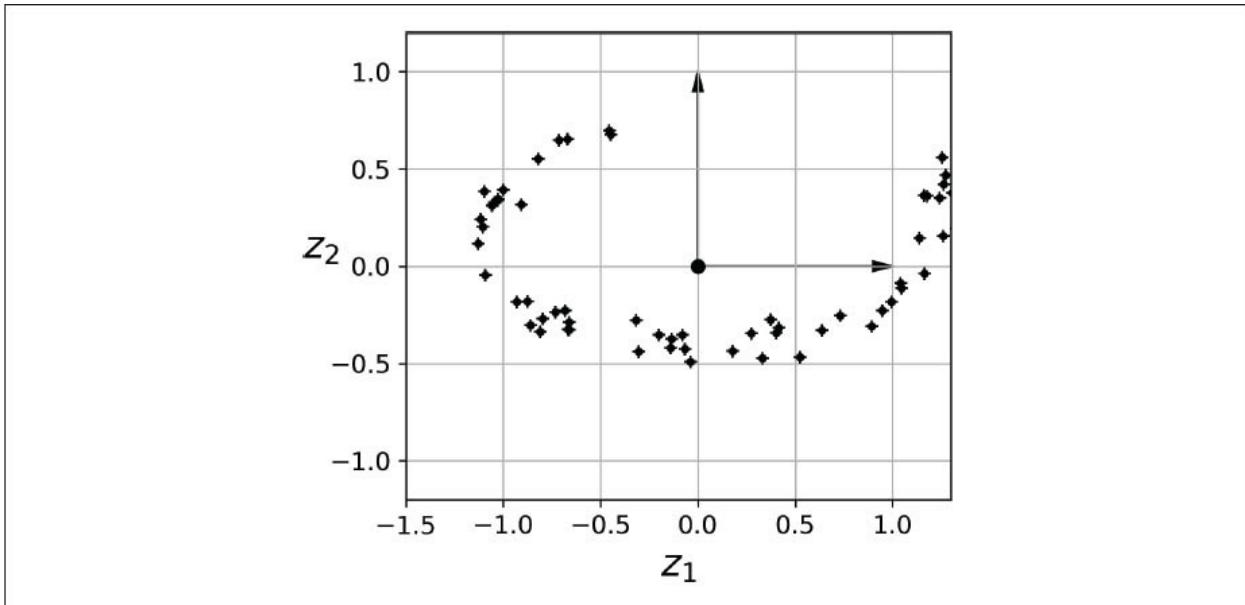


Abbildung 8-3: Der neue 2-D-Datensatz nach der Projektion

Eine Projektion ist jedoch nicht immer die beste Lösung zur Dimensionsreduktion. In vielen Fällen ist der Subraum gekrümmt und verdreht, wie im Fall des berühmten *Swiss-Roll*-Datensatzes in [Abbildung 8-4](#).

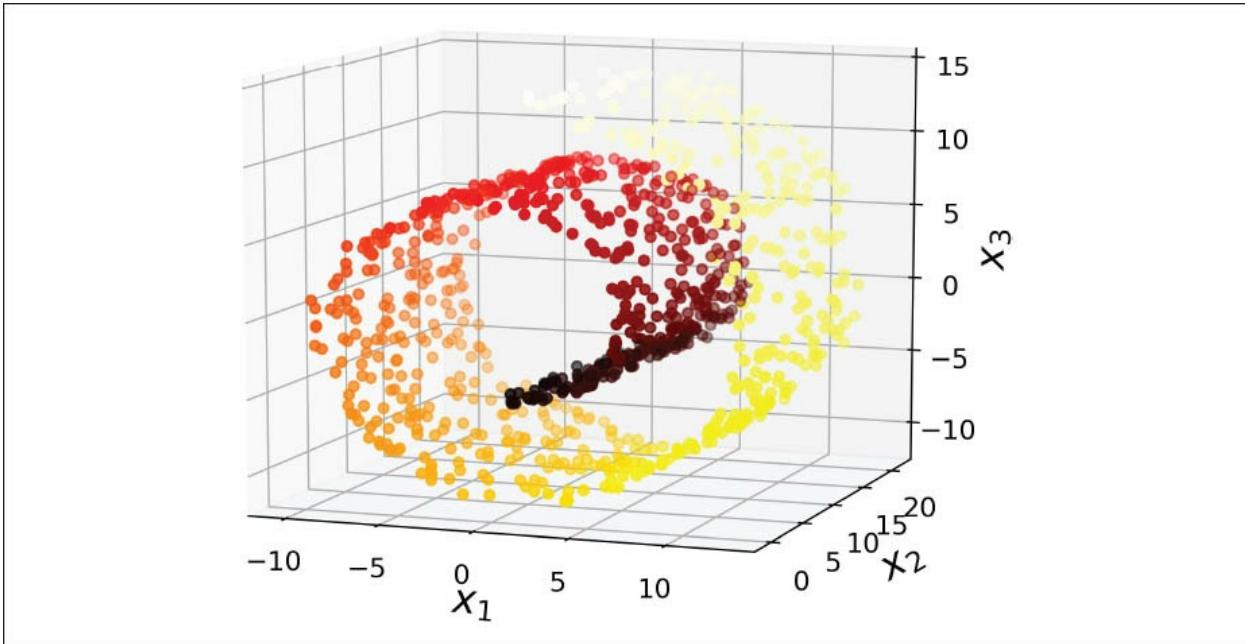


Abbildung 8-4: Swiss-Roll-Datensatz

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.

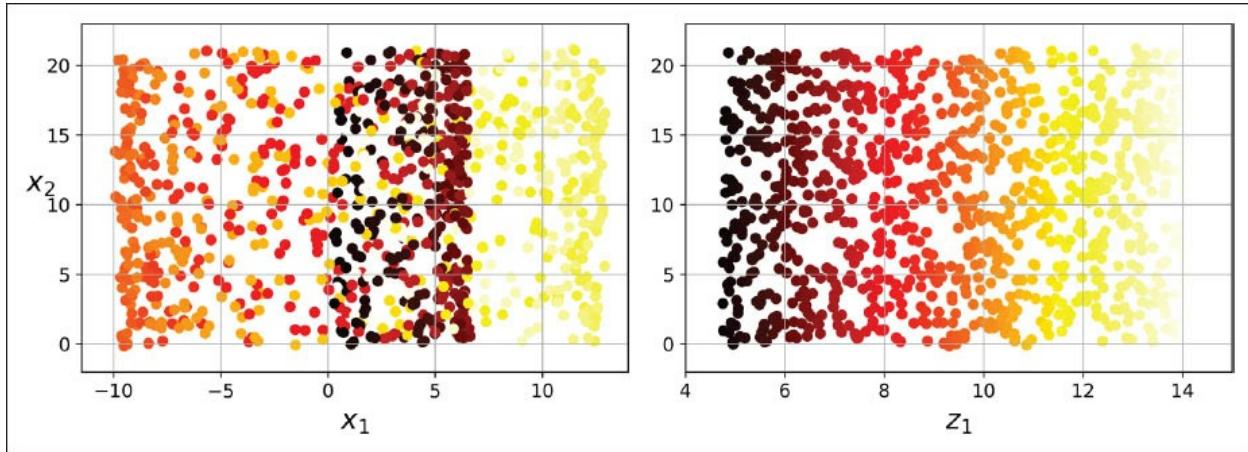


Abbildung 8-5: Eine auf eine Ebene gequetschte Projektion (links) im Vergleich zum Aufrollen der Swiss-Roll-Daten (rechts)

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 Fall 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, und sie sind mehr oder weniger zentriert. Wenn Sie zufällig Bilder erzeugten, würde nur ein lächerlich geringer Bruchteil nach handgeschriebenen Ziffern aussehen. Anders ausgedrückt, die Freiheitsgrade beim Erzeugen des Bilds einer Ziffer sind 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 implizite 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 im Allgemeinen 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 können, besonders wenn die Manifold-Annahme gilt. Im restlichen Kapitel werden wir uns einige der beliebtesten Algorithmen anschauen.

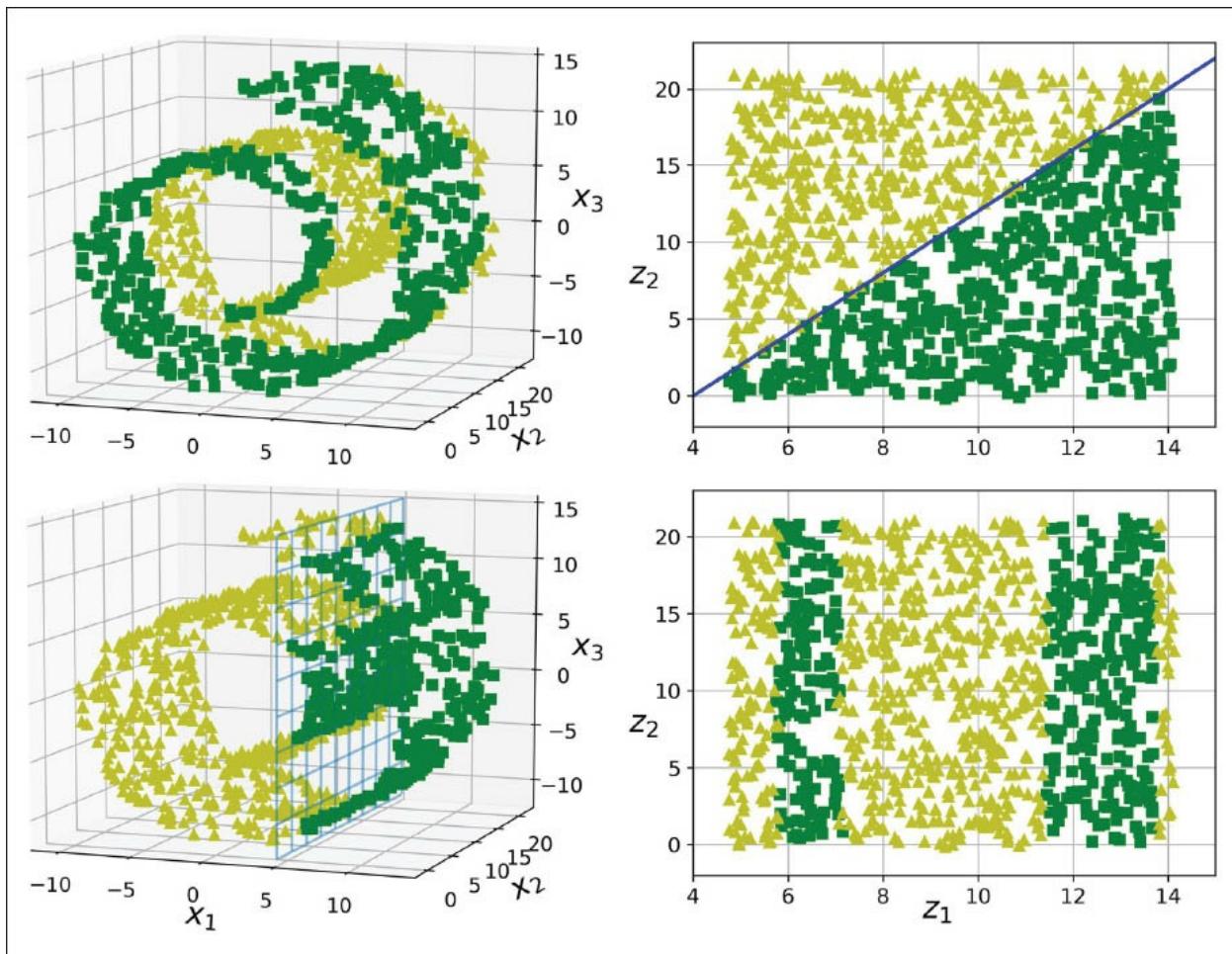


Abbildung 8-6: Die Entscheidungsgrenze wird bei weniger Dimensionen nicht immer einfacher.

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, so wie in [Abbildung 8-2](#).

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 gestrichelte Linie liegt irgendwo dazwischen.

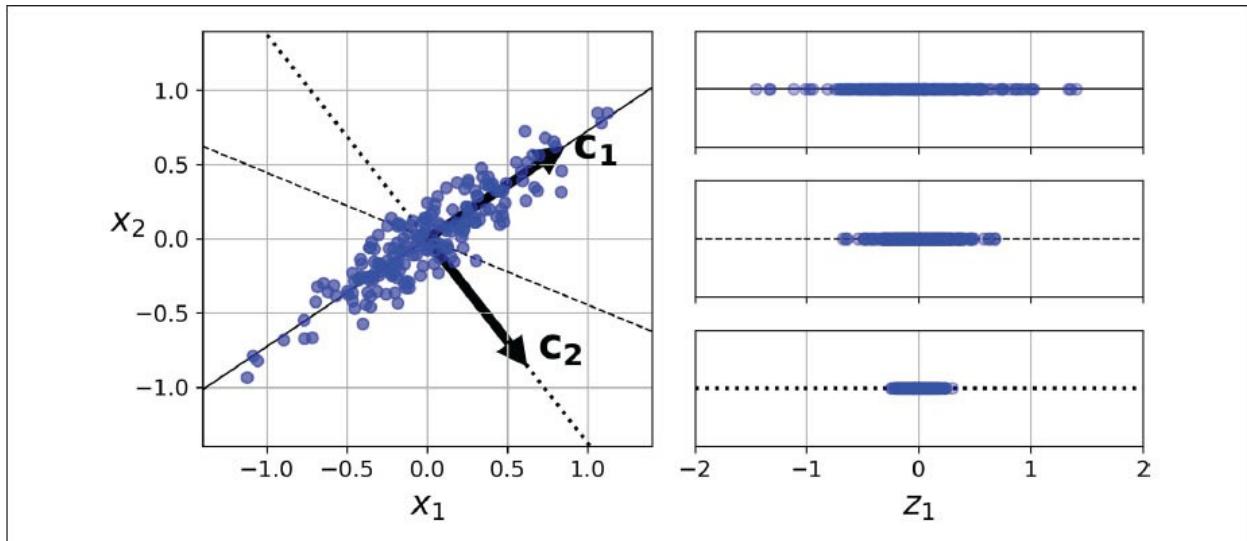


Abbildung 8-7: Auswählen eines Subraums für die Projektion

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 (<https://homl.info/pca>).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 vorherigen Achsen orthogonal, finden, dann eine vierte, eine fünfte und so weiter – so viele Achsen, wie der Datensatz Dimensionen besitzt.

Die i . Achse wird die i . *Hauptkomponente* (PC) der Daten genannt. In [Abbildung 8-7](#) ist die 1. Hauptkomponente die Achse, auf der Vektor \mathbf{c}_1 liegt, und die zweite Hauptkomponente

diejenige, auf der Vektor \mathbf{c}_2 liegt. In Abbildung 8-2 sind die ersten zwei Hauptkomponenten die orthogonalen Achsen, auf denen die beiden Pfeile in der Ebene liegen. Die dritte Hauptkomponente ist die Achse orthogonal zur Ebene.



Für jede Hauptkomponente findet die PCA einen um den Nullpunkt zentrierten Vektor, der in die Richtung der Hauptkomponente zeigt. Da zwei entgegengesetzte Vektoren auf der gleichen Achse liegen, ist die Richtung der Einheitsvektoren, die von der PCA zurückgegeben werden, nicht stabil: Wenn Sie den Trainingsdatensatz leicht durchmischen und die PCA erneut durchführen, zeigen einige Hauptkomponenten eventuell 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 (wenn die Abweichungen entlang dieser beiden Achsen klein sind), 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 Matrixprodukt der drei Matrizen $\mathbf{U} \Sigma \mathbf{V}^T$ zerlegen können, wobei \mathbf{V} die Einheitsvektoren enthält, die sämtliche von uns 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 beiden Einheitsvektoren, die die ersten zwei Hauptkomponenten definieren:

```
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 und einen reduzierten Datensatz $\mathbf{X}_{d\text{-proj}}$ mit der Dimension d zu erhalten, berechnen Sie die Matrixmultiplikation aus der Matrix \mathbf{X} mit den Trainingsdaten und der Matrix \mathbf{W}_d , bei der es sich um die Matrix aus den ersten d Spalten von \mathbf{V} handelt, wie [Formel 8-2](#) zeigt.

Formel 8-2: Projektion des Trainingsdatensatzes auf d Dimensionen

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X}\mathbf{W}_d$$

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 finden Sie im Attribut `components_` die Transponierte von \mathbf{W}_d (zum Beispiel entspricht der Einheitsvektor, der die erste Hauptkomponente definiert, `pca.components_.T[:, 0]`).

Der Anteil erklärter Varianz

Eine weitere sehr nützliche Information ist der *Anteil erklärter Varianz* jeder Hauptkomponente, der im Attribut `explained_variance_ratio_` verfügbar ist. Dieser zeigt an, welcher Teil der Varianz entlang 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 Hauptkomponente liegen und 14,6% entlang der zweiten Hauptkomponente. Damit bleiben weniger als 1,2% für die dritte Hauptkomponente übrig, man darf also annehmen, dass diese nur wenig Information enthält.

Auswählen der richtigen Anzahl Dimensionen

Anstatt die Anzahl verbleibender Dimensionen willkürlich auszuwählen, ist es einfacher, 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 Fall 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. In diesem Fall sehen Sie, dass bei einer Reduktion auf 100 Dimensionen nur wenig der erklärten Varianz verloren ginge.

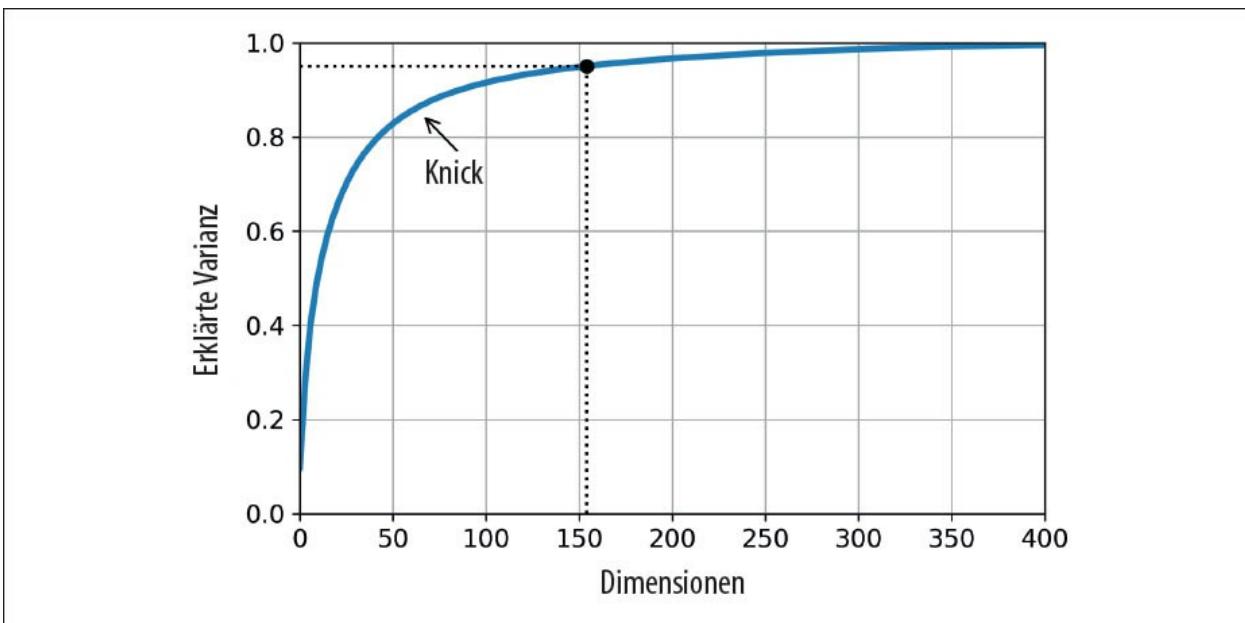


Abbildung 8-8: Erklärte Varianz als Funktion der Anzahl Dimensionen

PCA als Komprimierungsverfahren

Der Trainingsdatensatz nimmt 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 an 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:

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

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.

Original	Komprimiert
0 0 1 7 3	0 0 1 7 3
5 9 1 3 2	5 9 1 3 2
5 8 5 8 8	5 8 5 8 8
2 6 4 0 3	2 6 4 0 3
3 6 0 3 1	3 6 0 3 1

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

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d-\text{proj}} \mathbf{W}_d^T$$

Randomisierte PCA

Setzen Sie den Hyperparameter `svd_solver` auf "randomized", nutzt Scikit-Learn einen stochastischen Algorithmus namens *randomisierte PCA*, der schnell eine Näherung der ersten d Hauptkomponenten findet. Seine Rechenkomplexität ist $O(m \times d^2) + O(d^3)$ statt $O(m \times n^2) + O(n^3)$ für den vollständigen PCA-Ansatz, daher ist er massiv schneller, wenn d viel kleiner als n ist:

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

Standardmäßig hat `svd_solver` den Wert "auto": Scikit-Learn nutzt dann automatisch die randomisierte PCA, wenn m oder n größer als 500 und d kleiner als 80% von m oder n ist – ansonsten greift es auf den vollständigen SVD zurück. Wollen Sie Scikit-Learn dazu zwingen, den vollständigen SVD zu nutzen, können Sie den Hyperparameter `svd_solver` auf "full" setzen.

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 Onlineumgebungen 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` (<https://homl.info/32>) 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)
```

Kernel-PCA

In [Kapitel 5](#) haben wir den Kerneltrick 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) (<https://homl.info/33>).⁶ Sie ist dazu geeignet, bei der Projektion Cluster von Datenpunkten zusammenzuhalten oder bisweilen sogar Datensätze in der Nähe eines verdrehten Manifold aufzurollen.

Der folgende Code verwendet 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), einmal mit einem RBF-Kernel und einmal mit einem sigmoiden Kernel.

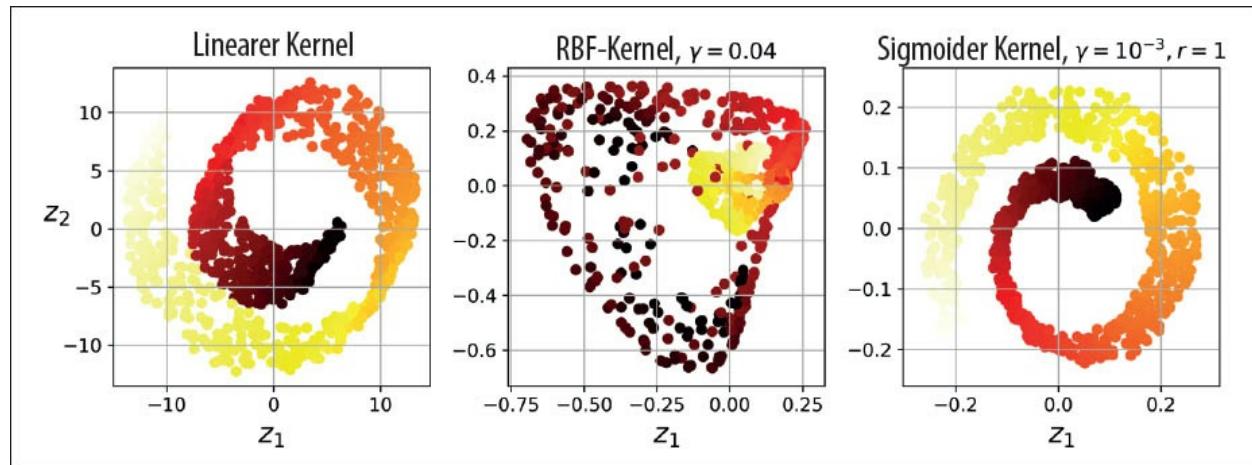


Abbildung 8-10: Mit kPCA auf 2-D reduzierte Swiss-Roll-Daten unter Verwendung unterschiedlicher Kernels

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 Kernels und Hyperparameter so zu wählen, dass die bestmögliche Leistung bei der Vorhersage erzielt wird. Der folgende Code erstellt 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 Kerneltrick 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.

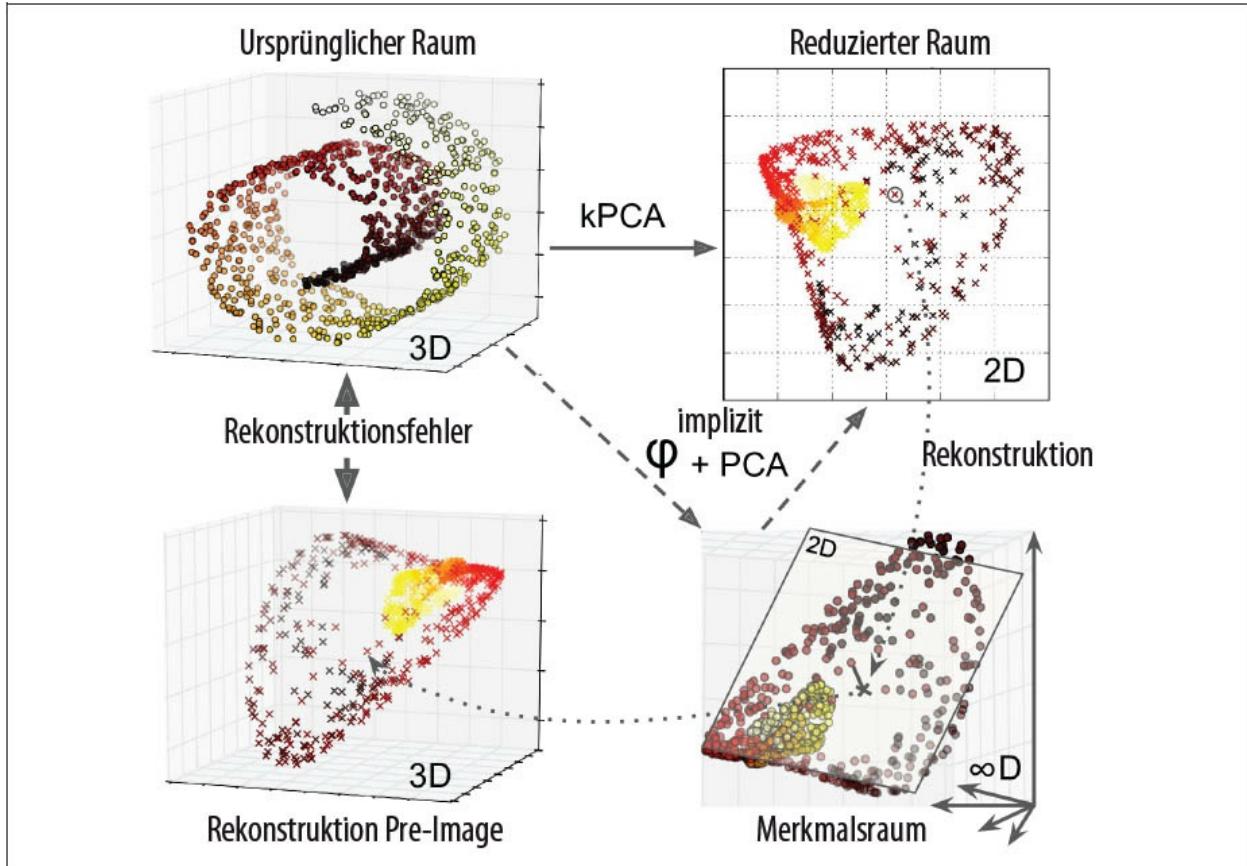


Abbildung 8-11: Kernel-PCA und die Abweichung vom Reconstruction Pre-Image

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.

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 zum Einsatz kommen.

LLE

Locally Linear Embedding (LLE) (<https://homl.info/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.

Der folgende Code verwendet die Scikit-Learn-Klasse `LocallyLinearEmbedding`, um die Swiss-Roll-Daten aufzurichten.

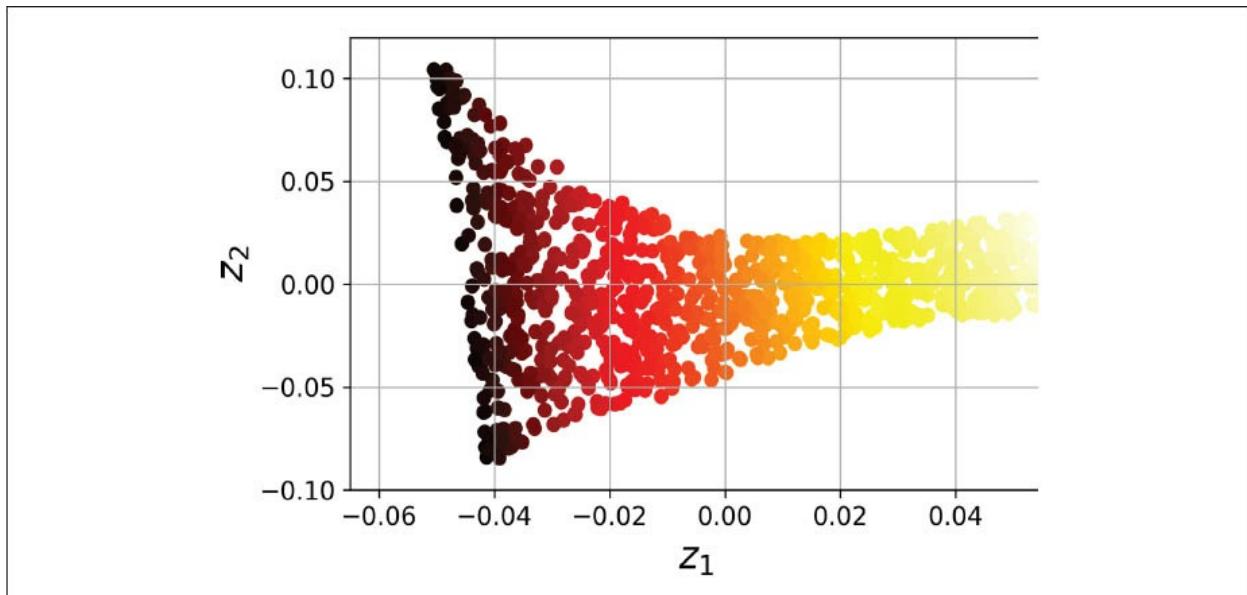


Abbildung 8-12: Mit LLE aufgerollte Swiss-Roll-Daten

Den erhaltenen 2-D-Datensatz zeigt [Abbildung 8-12](#). 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 auseinandergezogen, der rechte Teil gequetscht. 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

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^m \left(\mathbf{z}^{(i)} - \sum_{j=1}^m \widehat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

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:

Zufällige Projektionen

Wie der Name schon andeutet, werden die Daten mithilfe einer zufälligen linearen Projektion auf einen niedriger dimensionalen Raum projiziert. Das mag verrückt klingen, aber es zeigt sich, dass solch eine zufällige Projektion die Abstände sehr gut beibehält, was mathematisch von William B. Johnson und Joram Lindenstrauss in einem berühmten Lemma gezeigt wurde. Die Qualität der Dimensionsreduktion hängt von der Anzahl an Datenpunkten und der Zieldimensionalität ab, überraschenderweise aber nicht von der ursprünglichen Dimensionalität. Mehr dazu finden Sie in der Dokumentation zum Paket `sklearn.random_projection`.

Multidimensionale Skalierung (MDS)

reduziert die Dimensionen unter Beibehaltung der Abstände zwischen den Datenpunkten (siehe [Abbildung 8-13](#)).

Isomap

Erstellt einen Graphen, indem es jeden Datenpunkt mit seinen nächsten Nachbarn verbindet. Anschließend werden die Dimensionen unter Beibehaltung der *geodätischen Distanz*⁹ 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 so weit wie möglich voneinander entfernt hält. Damit ist LDA zur Dimensionsreduktion geeignet, bevor ein Klassifikationsalgorithmus wie ein SVM-Klassifikator ausgeführt wird.

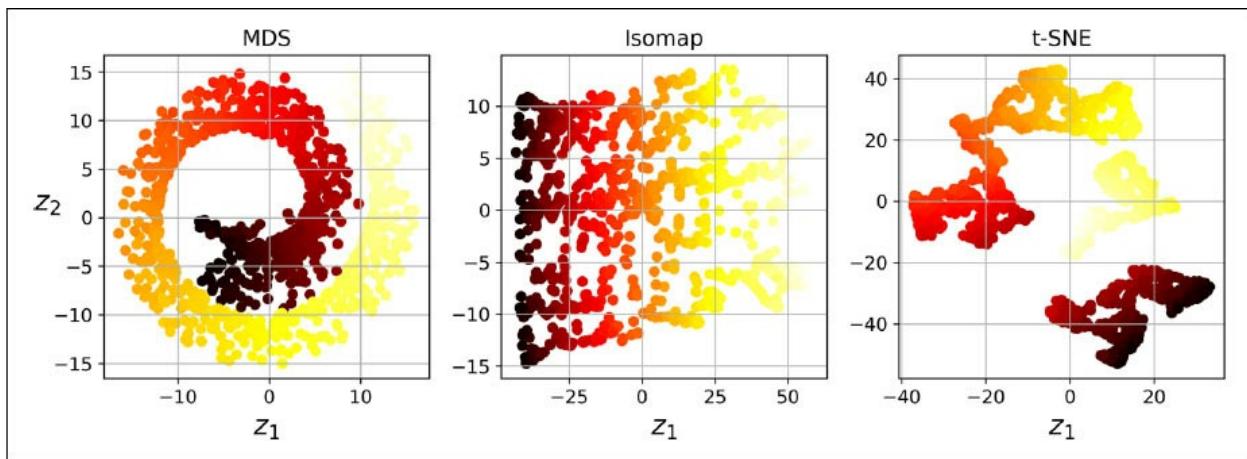


Abbildung 8-13: Reduzieren der Swiss-Roll-Daten zu einer 2-D-Darstellung mit unterschiedlichen Techniken

Übungen

1. Welche sind die wichtigsten Gründe, die Dimensionen eines Datensatzes zu verringern?
Was 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 60. 000 Datenpunkte zum Trainieren und die übrigen 10.000 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 jeden Punkt im Scatterplot durch die entsprechende Kategorie des Datenpunkts ersetzen (eine Ziffer von 0 bis 9) oder sogar verkleinerte Versionen der Ziffernbilder selbst nutzen. (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, wenn 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](#).

Techniken des unüberwachten Lernens

Auch wenn die meisten Anwendungen des maschinellen Lernens heutzutage auf überwachtem Lernen basieren (und daher dort auch am meisten investiert wird), ist der größte Teil der verfügbaren Daten nicht ausgezeichnet: Wir haben die Eingabemerkmale \mathbf{X} , aber keine Label y . Der Informatiker Yann LeCun hat einmal den berühmten Satz gesagt: »Wenn Intelligenz ein Kuchen wäre, wäre unüberwachtes Lernen der Kuchen, überwachtes Lernen der Zuckerguss und Reinforcement Learning die Kirsche darauf.« Mit anderen Worten – unüberwachtes Lernen bietet großes Potenzial, und wir haben gerade erst an der Oberfläche gekratzt.

Nehmen wir an, Sie wollten ein System erstellen, das ein paar Bilder von jedem Teil einer Fertigungslinie erfasst und dann ermittelt, welche davon defekt sind. Sie können ziemlich einfach ein System aufbauen, das automatisch Bilder aufnimmt, und damit bekommen Sie vielleicht jeden Tag Tausende Aufnahmen. So können Sie innerhalb von wenigen Wochen einen sinnvoll großen Datensatz erzeugen. Aber halt: Es gibt keine Labels! Wenn Sie einen normalen binären Klassifikator trainieren wollen, damit dieser vorhersagt, ob ein Teil defekt ist oder nicht, müssen Sie jedes einzelne Bild als »defekt« oder »normal« labeln. Dafür müssen sich im Allgemeinen Experten hinsetzen und manuell durch alle Bilder gehen. Das ist eine lange, teure und langweilige Aufgabe, daher macht man das meist nur für eine kleine Untermenge der verfügbaren Bilder. Im Ergebnis wird der mit Labels versehene Datensatz ziemlich klein sein und die Leistung des Klassifikators eher enttäuschen. Zudem muss der gesamte Prozess jedes Mal, wenn die Firma Änderungen an ihrem Produkt vornimmt, komplett neu durchlaufen werden. Wäre es nicht toll, wenn der Algorithmus die ungelabelten Daten einfach untersuchen könnte, ohne dass Menschen jedes Bild labeln müssen? Hier kommt das unüberwachte Lernen ins Spiel.

In [Kapitel 8](#) haben wir uns die Aufgabe angesehen, bei der am häufigsten unüberwachtes Lernen zum Einsatz kommt – die Dimensionsreduktion. In diesem Kapitel werden wir uns ein paar weitere Aufgaben und Algorithmen zum unüberwachten Lernen vornehmen:

Clustering

Das Ziel ist, ähnliche Instanzen in *Clustern* zu gruppieren. Das Clustering ist ein großartiges Werkzeug zur Datenanalyse, Kundensegmentierung, für Empfehlungssysteme, Suchmaschinen, Bildsegmentierung, teilüberwachtes Lernen, Dimensionsreduktion und mehr.

Anomalieerkennung

Hier geht es darum, zu lernen, wie »normale« Daten aussehen, und diese Erkenntnisse dann zu nutzen, um anormale Instanzen zu finden, wie zum Beispiel fehlerhafte Teile auf einem

Fließband oder einen neuen Trend in einer Zeitserie.

Dichteabschätzung

Bei dieser Aufgabe wird die *Wahrscheinlichkeitsdichtefunktion* (WDF; Probability Density Function, PDF) des Zufallsprozesses geschätzt, der den Datensatz erzeugt hat. Die Dichteabschätzung wird meist zur Anomalieerkennung genutzt: Instanzen, die sich in Regionen mit einer sehr geringen Dichte befinden, sind eher Anomalien. Zudem ist sie zur Datenanalyse und Visualisierung nützlich.

Sind Sie jetzt bereit für ein wenig Kuchen? Wir werden mit dem Clustering beginnen, K-Means und DBSCAN nutzen und dann gaußsche Mischverteilungsmodelle besprechen, um herauszufinden, wie wir sie zur Dichteabschätzung, zum Clustering und zur Anomalieerkennung einsetzen können.

Clustering

Sie erfreuen sich gerade an einer Bergwanderung, als Sie über eine Pflanze stolpern, die Sie noch nie zuvor gesehen haben. Sie schauen sich um und sehen noch ein paar mehr davon. Die Pflanzen sind nicht identisch, sehen sich aber so ähnlich, dass Sie davon ausgehen können, dass es sich um die gleiche Art handelt (oder zumindest um die gleiche Gattung). Jetzt wäre ein Botaniker praktisch, der Ihnen sagen könnte, um was für eine Art es sich hier handelt, aber Sie brauchen sicherlich keinen Experten, um eine Gruppe ähnlich aussehender Objekte zu erkennen. Das nennt man *Clustering* – das Identifizieren ähnlicher Instanzen und ihr Zuweisen zu *Clustern* oder Gruppen aus ähnlichen Instanzen.

Wie bei der Klassifikation wird jede Instanz einer Gruppe zugewiesen. Aber anders als dort handelt es sich beim Clustering um eine unüberwachte Aufgabe. Schauen Sie sich [Abbildung 9-1](#) an: Links befindet sich der Iris-Datensatz (den wir in [Kapitel 4](#) kennengelernt haben), bei der jede Art einer Instanz durch einen eigenen Marker dargestellt wird. Es handelt sich um einen gelabelten Datensatz, für den Klassifikationsalgorithmen wie logistische Regression, SVMs oder Random Forests gut geeignet sind. Rechts sehen Sie den gleichen Datensatz, aber ohne die Labels, sodass Sie keinen Klassifikationsalgorithmus mehr anwenden können. Hier kommen die Clustering-Algorithmen ins Spiel: Viele von ihnen können das Cluster links unten gut erkennen. Es lässt sich auch mit unseren eigenen Augen gut erkennen, nicht so offensichtlich ist aber, dass das Cluster oben rechts aus zwei verschiedenen Unterclustern besteht. Allerdings besitzt der Datensatz zwei zusätzliche Merkmale (Länge und Breite der Kelchblätter), die hier nicht gezeigt werden, und die Clustering-Algorithmen können sich auf alle Merkmale beziehen, sodass sich die drei Cluster tatsächlich recht gut trennen lassen (zum Beispiel werden bei einem gaußschen Mischverteilungsmodell nur 5 von 150 Instanzen dem falschen Cluster zugewiesen).

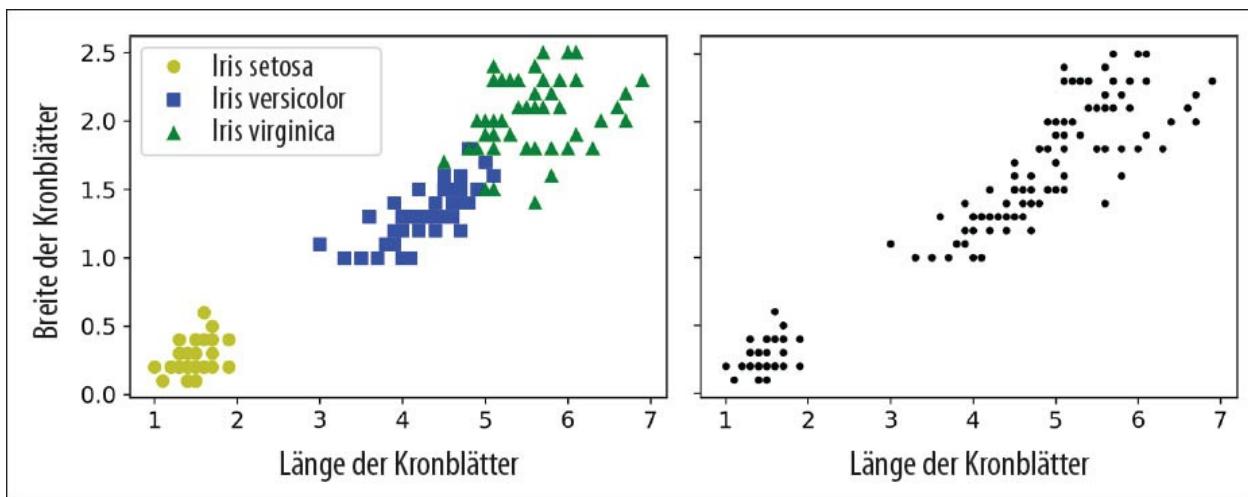


Abbildung 9-1: Klassifikation (links) versus Clustering (rechts)

Für das Clustering gibt es eine Vielzahl von Anwendungsmöglichkeiten, unter anderem:

Kundensegmentierung

Sie können Ihre Kunden anhand ihrer Einkäufe und ihrer Aktivitäten auf Ihrer Website clustern. Das ist nützlich, um zu verstehen, wer Ihre Kunden sind und was sie brauchen, sodass Sie Ihre Produkte und Marketingkampagnen an jedes Segment anpassen können. So kann die Kundensegmentierung beispielsweise in *Empfehlungssystemen* dabei helfen, Inhalte vorzuschlagen, an denen schon andere Kunden im gleichen Cluster Gefallen gefunden haben.

Datenanalyse

Analysieren Sie einen neuen Datensatz, kann es hilfreich sein, einen Clustering-Algorithmus laufen zu lassen und dann jedes Cluster getrennt auszuwerten.

Technik zur Dimensionsreduktion

Wurde ein Datensatz geclustert, ist es meist möglich, die *Affinität* jeder Instanz zu jedem Cluster zu messen (die Affinität ist ein Messverfahren, um zu bestimmen, wie gut eine Instanz in ein Cluster passt). Jeder Merkmalsvektor \mathbf{x} einer Instanz lässt sich dann durch den Vektor seiner Cluster-Affinitäten ersetzen. Gibt es k Cluster, ist dieser Vektor k -dimensional. Er ist meist deutlich niedrigdimensionaler als der ursprüngliche Merkmalsvektor, kann aber weiterhin genug Informationen für eine folgende Verarbeitung enthalten.

Anomalieerkennung (auch als Ausreißererkennung bezeichnet)

Jede Instanz mit einer niedrigen Affinität zu allen Clustern ist sehr wahrscheinlich eine Anomalität. Haben Sie beispielsweise die Besucher Ihrer Website anhand ihres Verhaltens geclustert, können Sie solche mit unüblichem Verhalten erkennen, wie zum Beispiel mit einer ungewöhnlich hohen Zahl an Requests pro Sekunde. Die Anomalieerkennung ist besonders nützlich beim Erkennen von Fehlern in der Herstellung oder zur *Betrugserkennung*.

Teilüberwachtes Lernen

Haben Sie nur wenige Labels, könnten Sie ein Clustering durchführen und die Labels allen Instanzen des gleichen Clusters zuweisen. Diese Technik kann die Menge an verfügbaren Labels für ein darauffolgendes überwachtes Lernen deutlich erhöhen und damit dessen Performance verbessern.

Suchmaschinen

Manche Suchmaschinen lassen Sie nach Bildern suchen, die einem Referenzbild ähnlich sehen. Um solch ein System aufzubauen, würden Sie zuerst einen Clustering-Algorithmus auf alle Bilder in Ihrer Datenbank anwenden – ähnliche Bilder würden dann im gleichen Cluster landen. Liefert ein Benutzer nun ein Referenzbild, müssen Sie nur das trainierte Clustering-Modell nutzen, um das Cluster zum Bild zu finden, und können dann alle Bilder dieses Clusters zurückgeben.

Bildsegmentierung

Durch ein Clustern der Pixel nach ihrer Farbe und das Ersetzen jeder Pixelfarbe durch den Mittelwert seines Clusters ist es möglich, die Menge an unterschiedlichen Farben im Bild deutlich zu verringern. Die Bildsegmentierung wird häufig in Systemen zur Objekterkennung und zum Tracken eingesetzt, da es das Erkennen der Kontur jedes Objekts leichter macht.

Es gibt keine universelle Definition eines Clusters – es hängt vom Kontext ab, und unterschiedliche Algorithmen werden auch unterschiedliche Arten von Clustern finden. Manche Algorithmen suchen nach Instanzen, die um einen bestimmten Punkt herum zentriert sind, der als *Schwerpunkt* bezeichnet wird. Andere halten Ausschau nach zusammenhängenden Gebieten dicht gepackter Instanzen – diese Cluster können jede Form annehmen. Manche Algorithmen sind hierarchisch und suchen nach Clustern aus Clustern. Und die Liste lässt sich beliebig verlängern.

In diesem Abschnitt werden wir uns zwei häufig genutzte Clustering-Algorithmen anschauen (K-Means und DBSCAN) und einige ihrer Anwendungsmöglichkeiten untersuchen, wie zum Beispiel nichtlineare Dimensionsreduktion, teilüberwachtes Lernen und Anomalieerkennung.

K-Means

Schauen Sie sich den ungelabelten Datensatz in [Abbildung 9-2](#) an: Sie erkennen deutlich fünf Bereiche mit Instanzen. Der K-Means-Algorithmus ist ein einfacher Algorithmus, der diese Art von Datensätzen sehr schnell und effizient in Cluster unterteilen kann – oft in nur wenigen Iterationen. Er wurde 1957 von Stuart Lloyd an den Bell Labs als Technik zur Puls-Code-Modulation vorgeschlagen, aber außerhalb der Firma erst 1982 veröffentlicht (<https://homl.info/36>).¹ Im Jahr 1965 veröffentlichte Edward W. Forgy nahezu den gleichen Algorithmus, daher wird K-Means manchmal auch als Lloyd-Forgy bezeichnet.

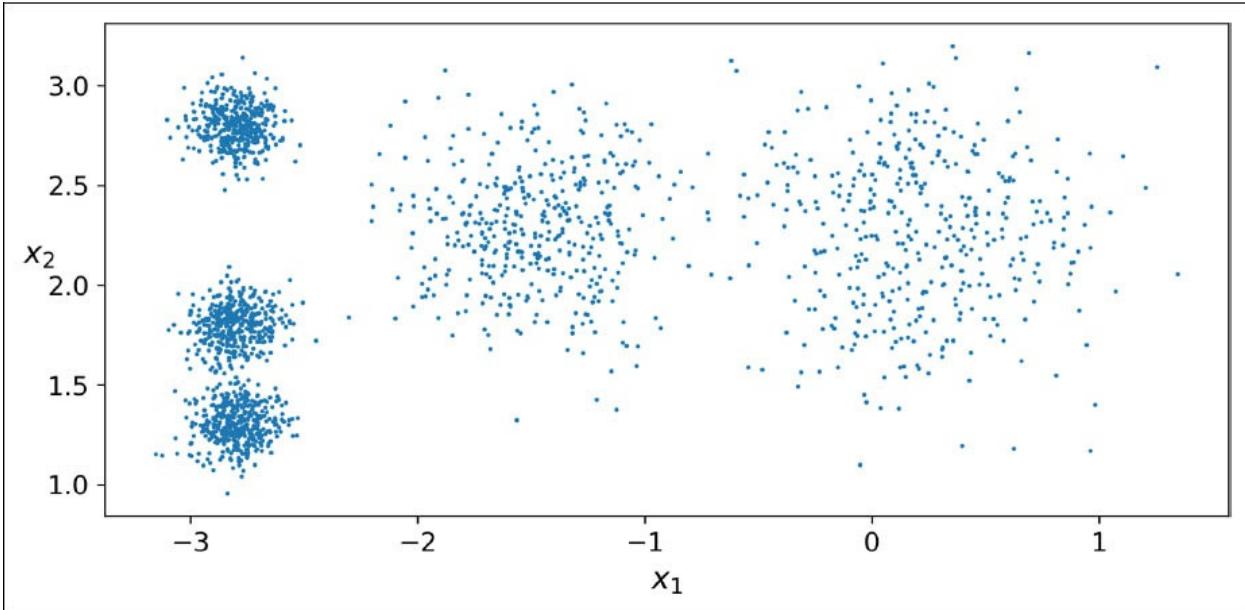


Abbildung 9-2: Ein ungelabelter Datensatz, der aus fünf Bereichen mit Instanzen besteht

Trainieren wir ein K-Means-Cluster mit diesem Datensatz. Es wird versuchen, den Mittelwert jedes Bereichs zu finden, und jede Instanz dem nächstgelegenen Bereich zuweisen:

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

Beachten Sie, dass Sie die Anzahl k an Clustern angeben müssen, die der Algorithmus finden muss. In diesem Beispiel reicht schon ein Blick auf die Daten, um festzustellen, dass k auf 5 gesetzt werden sollte, aber im Allgemeinen ist es nicht so einfach. Wir werden darauf in Kürze noch eingehen.

Jede Instanz wurde einem der fünf Cluster zugewiesen. Im Rahmen des Clusterings ist das *Label* einer Instanz der Index des Clusters, dem diese Instanz durch den Algorithmus zugewiesen wurde. Es ist nicht zu verwechseln mit den Kategorie-Labels aus der Klassifikation (vergessen Sie nicht: Clustering ist unüberwachtes Lernen). Die KMeans-Instanz bewahrt eine Kopie der Labels der Instanzen auf, mit denen es trainiert wurde. Sie erhalten sie über die Instanzvariable `labels_`:

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
```

True

Wir können uns auch die fünf Schwerpunkte anschauen, die der Algorithmus gefunden hat:

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

Neue Instanzen lassen sich leicht dem Cluster zuweisen, deren Schwerpunkt am nächsten liegt:

```
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

Plotten Sie die Entscheidungsgrenzen der Cluster, erhalten Sie ein Voronoi-Diagramm (siehe [Abbildung 9-3](#), in dem jeder Schwerpunkt durch ein X markiert ist).

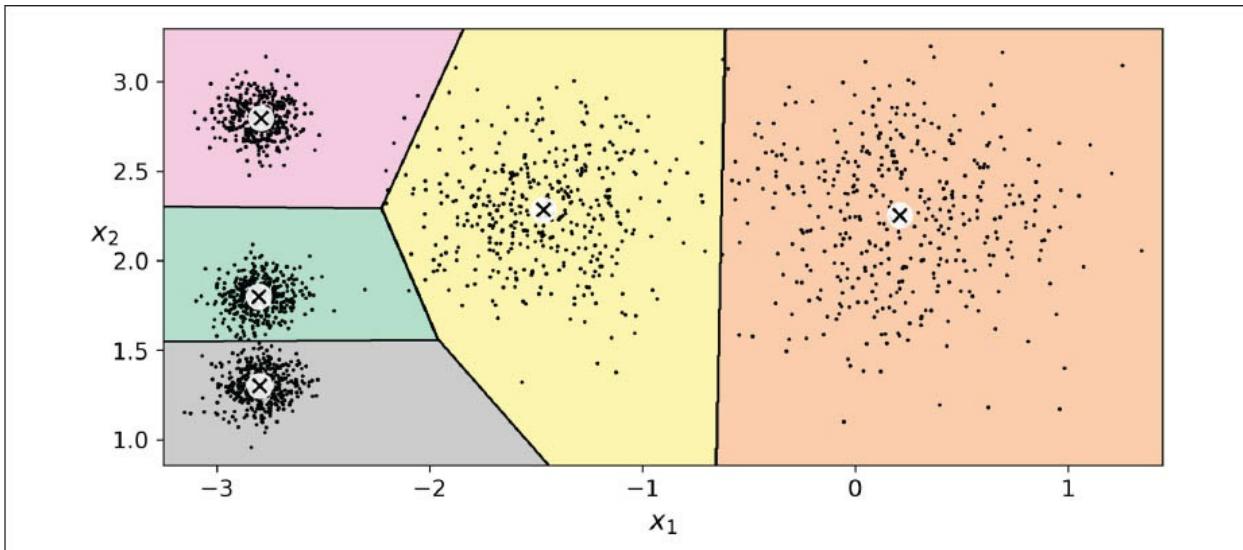


Abbildung 9-3: Entscheidungsgrenzen nach K-Means (Voronoi-Diagramm)

Der größte Teil der Instanzen wurde klar dem passenden Cluster zugewiesen, aber ein paar haben vermutlich das falsche Label erhalten (insbesondere nahe der Grenze zwischen dem oberen linken und dem zentralen Cluster). Tatsächlich verhält sich der K-Means-Algorithmus nicht sehr gut, wenn die Bereiche sehr unterschiedliche Durchmesser haben, denn er kümmert sich beim

Zuweisen einer Instanz zu einem Cluster nur um den Abstand zum Schwerpunkt.

Statt jede Instanz einem einzelnen Cluster zuzuweisen, was als *Hard Clustering* bezeichnet wird, kann es sinnvoll sein, jeder Instanz einen Score pro Cluster zu geben, was man *Soft Clustering* nennt. Der Score kann der Abstand zwischen der Instanz und dem Schwerpunkt sein – umgekehrt kann es sich auch um einen Ähnlichkeitsscore (oder eine Affinität) wie die gaußsche radiale Basisfunktion handeln (die wir in [Kapitel 5](#) kennengelernt haben). Bei der Klasse KMeans misst die Methode `transform()` den Abstand jeder Instanz zu jedem Schwerpunkt:

```
>>> kmeans.transform(X_new)

array([[2.81093633, 0.32995317, 2.9042344 , 1.49439034, 2.88633901],
       [5.80730058, 2.80290755, 5.84739223, 4.4759332 , 5.84236351],
       [1.21475352, 3.29399768, 0.29040966, 1.69136631, 1.71086031],
       [0.72581411, 3.21806371, 0.36159148, 1.54808703, 1.21567622]])
```

In diesem Beispiel findet sich die erste Instanz in `X_new` in einem Abstand von 2,81 zum ersten Schwerpunkt, 0,33 zum zweiten Schwerpunkt, 2,90 zum dritten Schwerpunkt, 1,49 zum vierten Schwerpunkt und 2,89 zum fünften Schwerpunkt. Haben Sie einen hochdimensionalen Datensatz und transformieren ihn auf diese Weise, landen Sie bei einem k -dimensionalen Datensatz: Diese Transformation kann eine sehr effiziente, nichtlineare Technik zur Dimensionsreduktion sein.

Der K-Means-Algorithmus

Wie funktioniert denn nun der Algorithmus? Nun, nehmen wir an, wir hätten schon die Schwerpunkte. Sie könnten dann alle Instanzen im Datensatz leicht labeln, indem Sie jede von ihnen dem Cluster zuweisen, deren Schwerpunkt am nächsten liegt. Wenn Sie umgekehrt alle Instanz-Labels hätten, könnten Sie alle Schwerpunkte bestimmen, indem Sie den Mittelwert der Instanzen für jedes Cluster berechnen. Aber wie sollen Sie vorgehen, wenn Sie weder Labels noch Schwerpunkte haben? Beginnen Sie einfach mit zufällig platzierten Schwerpunkten (zum Beispiel indem Sie zufällig k Instanzen auswählen und deren Positionen als Schwerpunkte wählen). Dann weisen Sie den Instanzen Labels zu, aktualisieren die Schwerpunkte, weisen den Instanzen Labels zu, aktualisieren die Schwerpunkte und so weiter, bis sich die Schwerpunkte nicht mehr verschieben. Der Algorithmus konvergiert garantiert in einer endlichen Anzahl von Schritten (meist ziemlich wenigen) – er oszilliert nicht für immer.²

Sie können den Algorithmus in [Abbildung 9-4](#) in Aktion sehen: Die Schwerpunkte wurden zufällig initialisiert (oben links), dann erhalten die Instanzen ihre Labels (oben rechts), dann werden die Schwerpunkte aktualisiert (Mitte links), die Instanzen neu gelabelt (Mitte rechts) und so weiter. Mit nur drei Iterationen haben die Algorithmen ein Clustering erreicht, das nahe am Optimum zu liegen scheint.

Die Komplexität der Rechenzeit des Algorithmus ist im Allgemeinen linear bezüglich der Anzahl an Instanzen m , Clustern k und Dimensionen n . Aber das gilt nur, wenn die Daten eine Cluster-Struktur besitzen. Ist das nicht so, kann im schlimmsten Fall die Komplexität exponentiell mit

der Anzahl an Instanzen wachsen. In der Praxis geschieht das nur selten, und K-Means ist im Allgemeinen einer der schnellsten Clustering-Algorithmen.

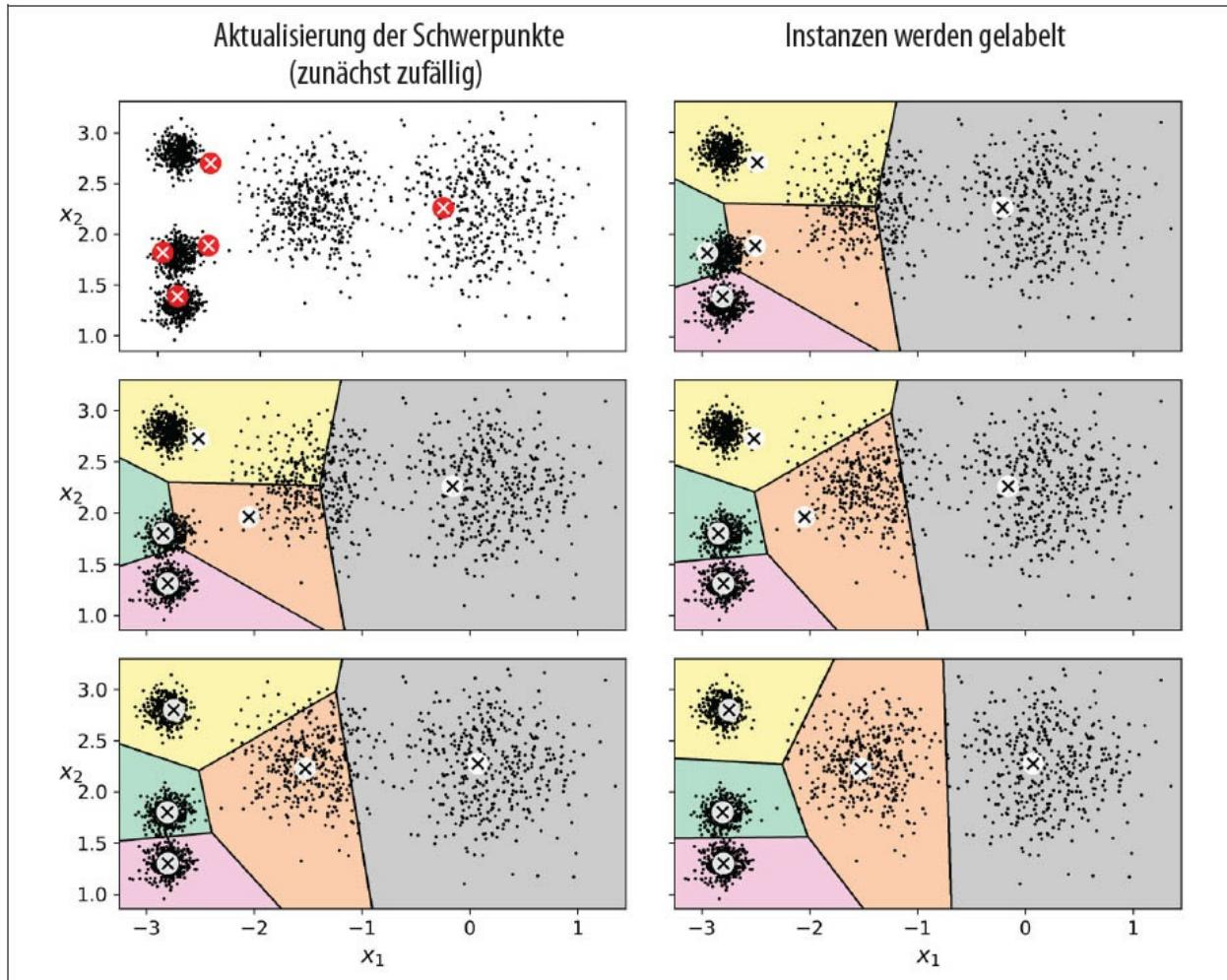


Abbildung 9-4: Der K-Means-Algorithmus

Auch wenn der Algorithmus garantiert konvergiert, tut er das eventuell nicht hin zur richtigen Lösung (sondern vielleicht nur zu einem lokalen Optimum). Das hängt von der Initialisierung der Schwerpunkte ab. In [Abbildung 9-5](#) sehen Sie zwei suboptimale Lösungen, zu denen der Algorithmus konvergiert kann, wenn Sie beim zufälligen ersten Schritt Pech haben.

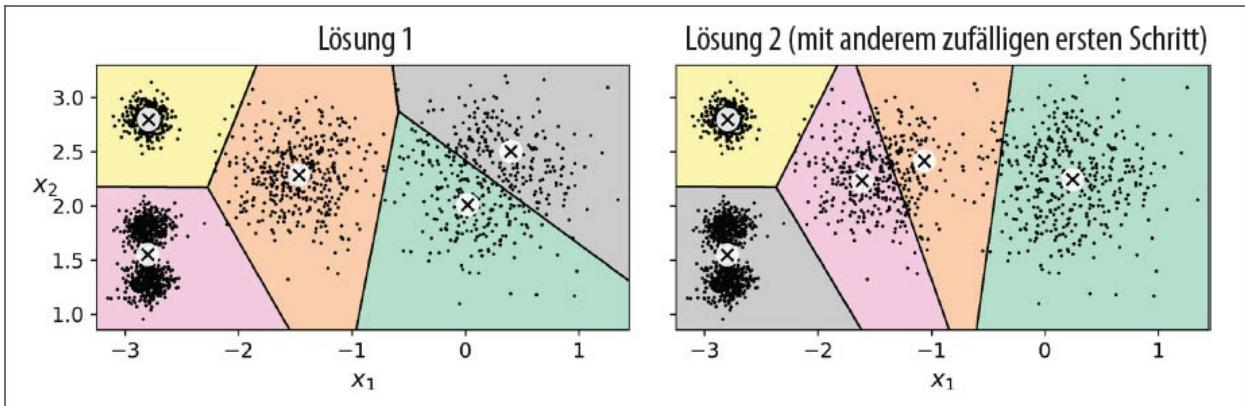


Abbildung 9-5: Suboptimale Lösungen durch eine unglückliche Initialisierung der Schwerpunkte

Schauen wir uns ein paar Möglichkeiten dazu an, wie Sie dieses Risiko durch das Verbessern der Initialisierung der Schwerpunkte reduzieren können.

Methoden zur Schwerpunktinitialisierung

Wenn Sie zufällig ungefähr wissen, wo sich die Schwerpunkte befinden sollten (zum Beispiel weil Sie zuvor schon einen anderen Clustering-Algorithmus laufen lassen), können Sie den Hyperparameter `init` auf ein NumPy-Array mit der Liste der Schwerpunkte setzen und für `n_init` den Wert 1 vorgeben:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

Eine andere Möglichkeit besteht darin, den Algorithmus mehrfach mit unterschiedlichen zufälligen Initialisierungen laufen zu lassen und die beste Lösung zu verwenden. Die Anzahl an zufälligen Initialisierungen wird durch den Hyperparameter `n_init` gesteuert: Standardmäßig hat er den Wert 10, was heißt, dass der gesamte zuvor beschriebene Algorithmus genau zehn Mal läuft, wenn Sie `fit()` aufrufen, und Scikit-Learn die beste Lösung auswählt. Aber wie genau weiß es, was die beste Lösung ist? Es verwendet eine Leistungsmetrik! Diese Metrik wird als *Trägheit* (Inertia) des Modells bezeichnet, und es handelt sich bei ihr um den mittleren quadratischen Abstand zwischen jeder Instanz und seinem nächstgelegenen Schwerpunkt. Für das Modell auf der linken Seite von Abbildung 9-5 beträgt dieser ungefähr 223,3, für das Modell auf der rechten Seite von Abbildung 9-5 ungefähr 237,5 und für das Modell in Abbildung 9-3 in etwa 211,6. Die Klasse `KMeans` lässt den Algorithmus `n_init` Mal laufen und behält dann das Modell mit der geringsten Trägheit. In diesem Beispiel wird das Modell in Abbildung 9-3 ausgewählt (sofern wir nicht bei `n_init` aufeinanderfolgenden zufälligen Initialisierungen sehr viel Pech haben). Wenn Sie neugierig sind: Die Trägheit eines Modells finden Sie in der Instanzvariablen `inertia_`:

```
>>> kmeans.inertia_
```

```
211.59853725816856
```

Die Methode `score()` gibt die negative Trägheit zurück. Warum negativ? Weil eine Methode `score()` eines Prädiktors immer die Regel »größer ist besser« von Scikit-Learn befolgen muss: Ist ein Prädiktor besser als ein anderer, sollte seine Methode `score()` einen größeren Wert zurückliefern.

```
>>> kmeans.score(X)
-211.59853725816856
```

Eine wichtige Verbesserung des K-Means-Algorithmus mit dem Namen *K-Means++* wurde in einem Artikel aus dem Jahr 2006 (<https://homl.info/37>) von David Arthur und Sergei Vassilvitskii vorgeschlagen.³ Die Autoren haben einen schlaueren Initialisierungsschritt eingeführt, der dazu tendiert, Schwerpunkte mit einem gewissen Abstand zueinander auszuwählen, was den K-Means-Algorithmus weniger anfällig für eine Konvergenz hin zu einer suboptimalen Lösung werden lässt. Dabei haben sie gezeigt, dass der zusätzliche Rechenaufwand für die schlauere Initialisierung gut investiert ist, weil sich damit die Menge an Ausführungen des Algorithmus massiv reduzieren lässt, um die optimale Lösung zu finden. Der Initialisierungsalgorithmus von K-Means++ sieht wie folgt aus:

- Wähle aus dem Datensatz gleichmäßig zufällig einen Schwerpunkt $\mathbf{c}^{(1)}$.
- Wähle einen neuen Schwerpunkt $\mathbf{c}^{(i)}$, wobei eine Instanz $\mathbf{x}^{(i)}$ mit der Wahrscheinlichkeit $D(\mathbf{x}^{(i)})^2 / \sum_{j=1}^m D(\mathbf{x}^{(j)})^2$ gewählt wird, wo $D(\mathbf{x}^{(i)})$ die Distanz zwischen der Instanz $\mathbf{x}^{(i)}$ und dem nächstgelegenen schon gewählten Schwerpunkt ist. Diese Wahrscheinlichkeitsverteilung stellt sicher, dass Instanzen, die sich weiter weg von schon gewählten Schwerpunkten befinden, eher als neue Schwerpunkte gewählt werden.
- Wiederhole den vorherigen Schritt, bis alle k Schwerpunkte ausgewählt wurden.

Die Klasse `KMeans` nutzt standardmäßig diese Initialisierungsmethode. Wollen Sie sie dazu zwingen, die ursprüngliche Methode zu verwenden (also völlig zufällig k Instanzen für die initialen Schwerpunkte auszuwählen), können Sie den Hyperparameter `init` auf "random" setzen. Aber das werden Sie nur selten benötigen.

Accelerated K-Means und Mini-Batch-K-Means

Eine weitere wichtige Verbesserung des K-Means-Algorithmus wurde in einem Artikel aus dem Jahr 2003 (<https://homl.info/38>) von Charles Elkan vorgeschlagen.⁴ Sie beschleunigt den Algorithmus merklich, indem sie viele unnötige Abstandsberechnungen vermeidet. Elkan erreicht das, indem er die Dreiecksungleichung ausnutzt (die besagt, dass eine gerade Linie immer die kürzeste Strecke zwischen zwei Punkten ist⁵) und sich die unteren und oberen Grenzen für Abstände zwischen Instanzen und Schwerpunkten merkt. Dies ist der Algorithmus, den die Klasse `KMeans` standardmäßig einsetzt (Sie können sie zum Anwenden des ursprünglichen Algorithmus zwingen, indem Sie den Hyperparameter `algorithm` auf "full" setzen, aber das werden Sie vermutlich nie tun müssen).

Eine weitere wichtige Variante des K-Means-Algorithmus wurde in einem Artikel aus dem Jahr 2010 (<https://homl.info/39>) von David Sculley vorgeschlagen.⁶ Statt bei jeder Iteration den kompletten Datensatz zu verwenden, kann der Algorithmus Mini-Batches nutzen und die Schwerpunkte bei jeder Iteration nur wenig verschieben. Das beschleunigt ihn typischerweise um einen Faktor 3 oder 4 und ermöglicht das Clustern riesiger Datensätze, die nicht in den Speicher passen. Scikit-Learn implementiert diesen Algorithmus in der Klasse `MiniBatchKMeans`. Sie können sie genauso einsetzen wie die Klasse `KMeans`:

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5)

minibatch_kmeans.fit(X)
```

Passt der Datensatz nicht in den Speicher, ist es am einfachsten, die Klasse `memmap` einzusetzen, wie wir das für die inkrementelle PCA in [Kapitel 8](#) gemacht haben. Alternativ können Sie auch immer einen Mini-Batch pro Durchlauf an die Methode `partial_fit()` übergeben, aber dafür ist mehr Aufwand nötig, denn Sie müssen mehrere Initialisierungen durchführen und die beste selbst auswählen (schauen Sie sich für ein Beispiel den Mini-Batch-K-Means-Abschnitt im Notebook an).

Der Mini-Batch-K-Means-Algorithmus ist zwar viel schneller als der normale K-Means-Algorithmus, aber seine Trägheit ist im Allgemeinen etwas schlechter, insbesondere mit zunehmender Anzahl an Clustern. Sie können das in [Abbildung 9-6](#) sehen: Der Plot auf der linken Seite vergleicht die Trägheiten von Mini-Batch-K-Means- mit der von normalen K-Means-Modellen, die mit dem vorherigen Datensatz für verschiedene Werte k an Cluster-Mengen trainiert wurden. Der Unterschied zwischen den beiden Kurven bleibt mehr oder weniger konstant, wird aber mit steigendem k signifikanter, da die Trägheit kleiner und kleiner wird. Im Plot auf der rechten Seite können Sie sehen, dass Mini-Batch-K-Means viel schneller als das normale K-Means ist und dieser Unterschied mit wachsendem k zunimmt.

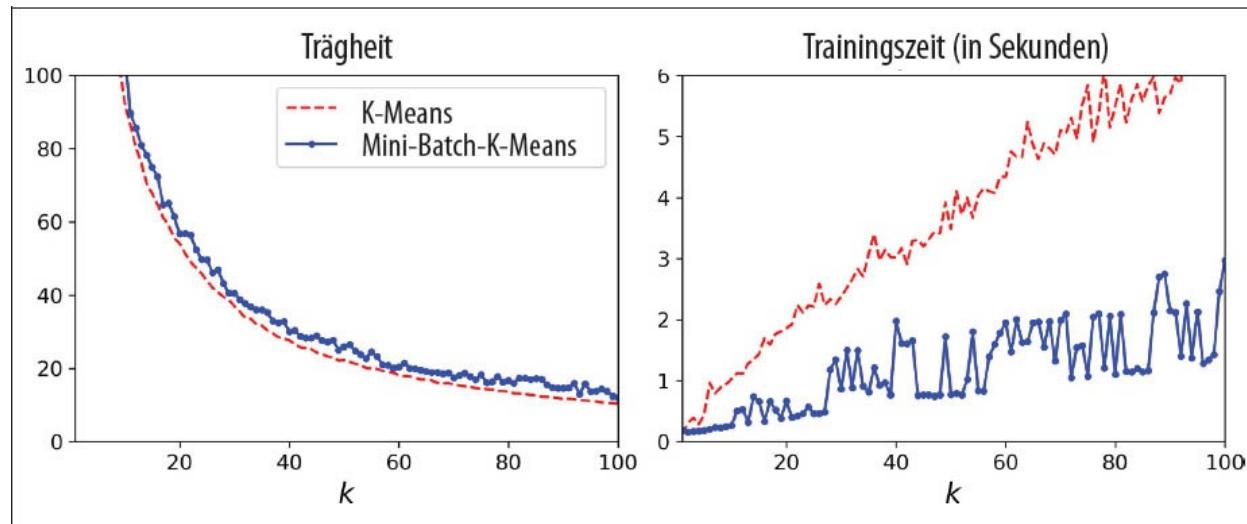


Abbildung 9-6: Mini-Batch-K-Means hat eine größere Trägheit als K-Means (links), ist aber viel schneller (rechts), insbesondere mit wachsendem k

Die optimale Zahl an Clustern finden

Bisher haben wir die Anzahl k an Clustern auf 5 gesetzt, weil durch einen Blick auf die Daten offensichtlich war, dass dies die korrekte Zahl war. Aber im Allgemeinen wird es nicht so einfach sein, zu wissen, worauf k zu setzen ist, und das Ergebnis kann ziemlich schlecht ausfallen, wenn Sie es auf den falschen Wert setzen. Wie Sie in Abbildung 9-7 sehen, führt ein Wert von 3 oder 8 für k zu wenig guten Modellen.

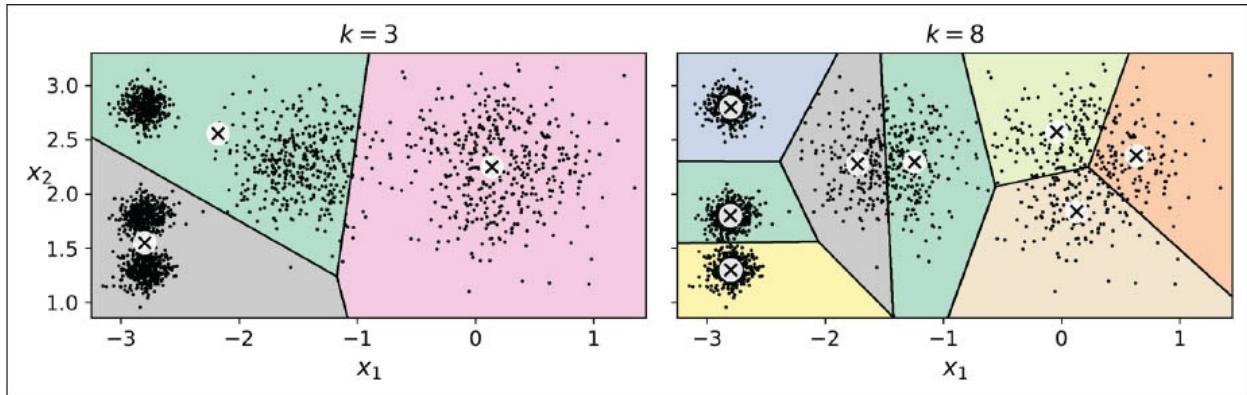


Abbildung 9-7: Schlechte Wahl für die Anzahl an Clustern: Ist k zu klein, werden getrennte Cluster zusammengeführt (links), ist k zu groß, werden manche Cluster in mehrere Stücke geteilt (rechts).

Vielleicht denken Sie jetzt, dass wir doch das Modell mit der kleinsten Trägheit wählen könnten. Aber so einfach ist es leider nicht. Die Trägheit für $k=3$ ist 653,2, was viel größer ist als die für $k=5$ (dort betrug sie 211,6). Aber mit $k=8$ hat die Trägheit einen Wert von nur 119,1. Sie ist also keine gute Leistungsmetrik zur Auswahl von k , weil sie mit zunehmendem k immer kleiner wird. Denn je mehr Cluster es gibt, desto kleiner wird jeder Abstand zum nächstgelegenen Schwerpunkt, und damit verringert sich auch die Trägheit. Plotten wir die Trägheit als Funktion von k (siehe Abbildung 9-8).

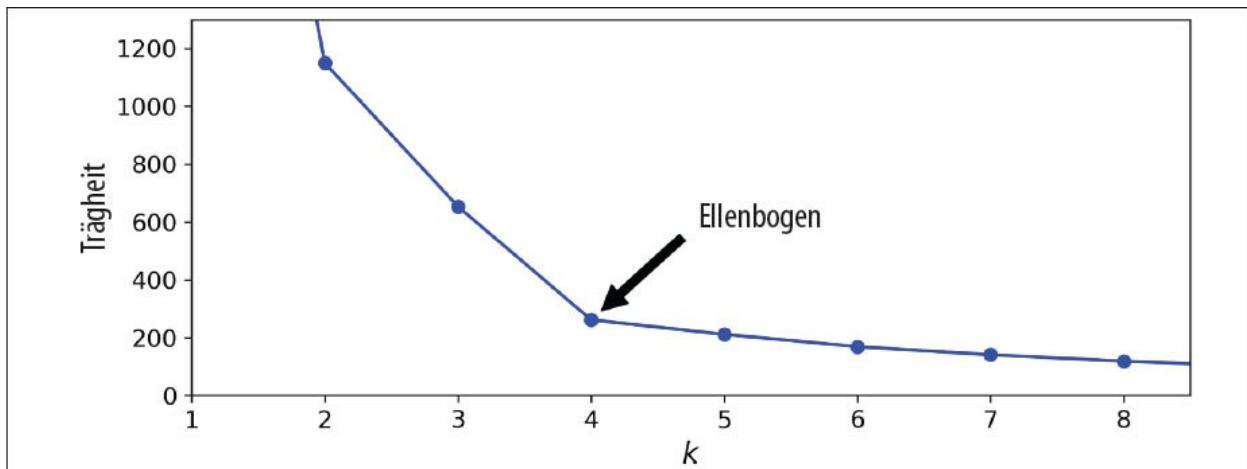


Abbildung 9-8: Beim Auftragen der Trägheit als Funktion der Anzahl an Clustern k gibt es in der Kurve oft einen Knickpunkt, der als »Ellenbogen« bezeichnet wird.

Wie Sie sehen, sinkt die Trägheit sehr schnell, wenn k bis auf 4 steigt, danach geht es mit wachsendem k nur noch langsam nach unten. Wenn wir es also nicht besser wissen, wäre 4 eine gute Wahl: Jeder niedrigere Wert wäre dramatisch, während jeder höhere Wert nicht viel helfen würde und wir gute Cluster ohne Grund aufteilen würden.

Diese Technik zum Auswählen des besten Werts für die Anzahl an Clustern ist eher grob. Präziser (aber auch rechenaufwendiger) ist die Verwendung des *Silhouettenkoeffizienten*, bei dem es sich um die gemittelten *Silhouetten* aller Instanzen handelt. Die Silhouette einer Instanz berechnet sich aus $(b - a) / \max(a, b)$, wobei a der mittlere Abstand zu den anderen Instanzen im gleichen Cluster (also die mittlere Intracluster-Distanz) und b die mittlere Distanz zu den Instanzen des am nächsten liegenden Clusters ist (also der mittlere Abstand zum nächstliegenden Cluster, definiert als das mit dem kleinsten b [ohne das Cluster der Instanz selbst]). Die Silhouette kann Werte zwischen -1 und $+1$ annehmen. Ein Koeffizient nahe an $+1$ besagt, dass sich die Instanz gut innerhalb ihres eigenen Clusters und weit weg von anderen Clustern befindet, ein Koeffizient nahe 0 bedeutet, dass sie sich nahe an einer Cluster-Grenze befindet, und ein Koeffizient von -1 zeigt schließlich, dass die Instanz eventuell dem falschen Cluster zugewiesen wurde.

Um den Silhouettenkoeffizienten zu berechnen, können Sie die Funktion `silhouette_score()` von Scikit-Learn verwenden und dabei alle Instanzen des Datensatzes sowie die zugewiesenen Labels übergeben:

```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```

Vergleichen wir nun die Silhouettenkoeffizienten für verschiedene Cluster-Anzahlen (siehe Abbildung 9-9).

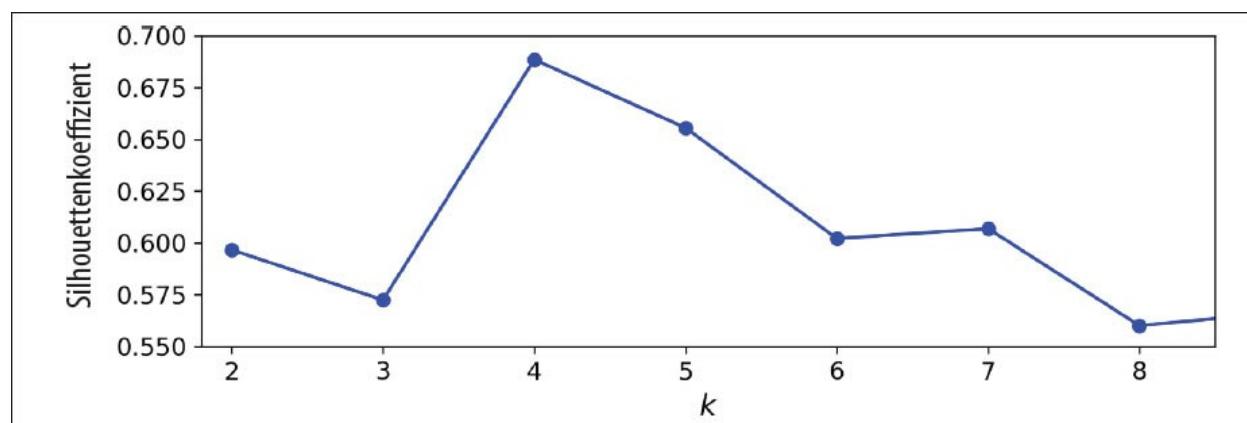


Abbildung 9-9: Auswahl der Anzahl an Clustern k mithilfe des Silhouettenkoeffizienten

Wie Sie sehen, ist diese Visualisierung deutlich hilfreicher als die vorherige: Sie bestätigt, dass $k = 4$ eine sehr gute Wahl ist, zeigt aber auch, dass $k = 5$ ebenfalls nicht schlecht ist – viel besser

als $k = 6$ oder 7 . Das war beim Vergleich der Trägheiten nicht zu erkennen.

Eine noch bessere Visualisierung erhalten Sie, wenn Sie die Silhouetten jeder einzelnen Instanz ausgeben – sortiert nach dem zugewiesenen Cluster und dem Wert selbst. Das wird als *Silhouettenplot* bezeichnet (siehe Abbildung 9-10). Jedes Diagramm enthält eine messerförmige Darstellung pro Cluster. Die Höhe dieses »Messers« wird durch die Anzahl an Instanzen im Cluster bestimmt, die Breite durch die sortierten Silhouetten der Instanzen im Cluster (breiter ist besser). Die gestrichelte Linie zeigt den Silhouettenkoeffizienten an.

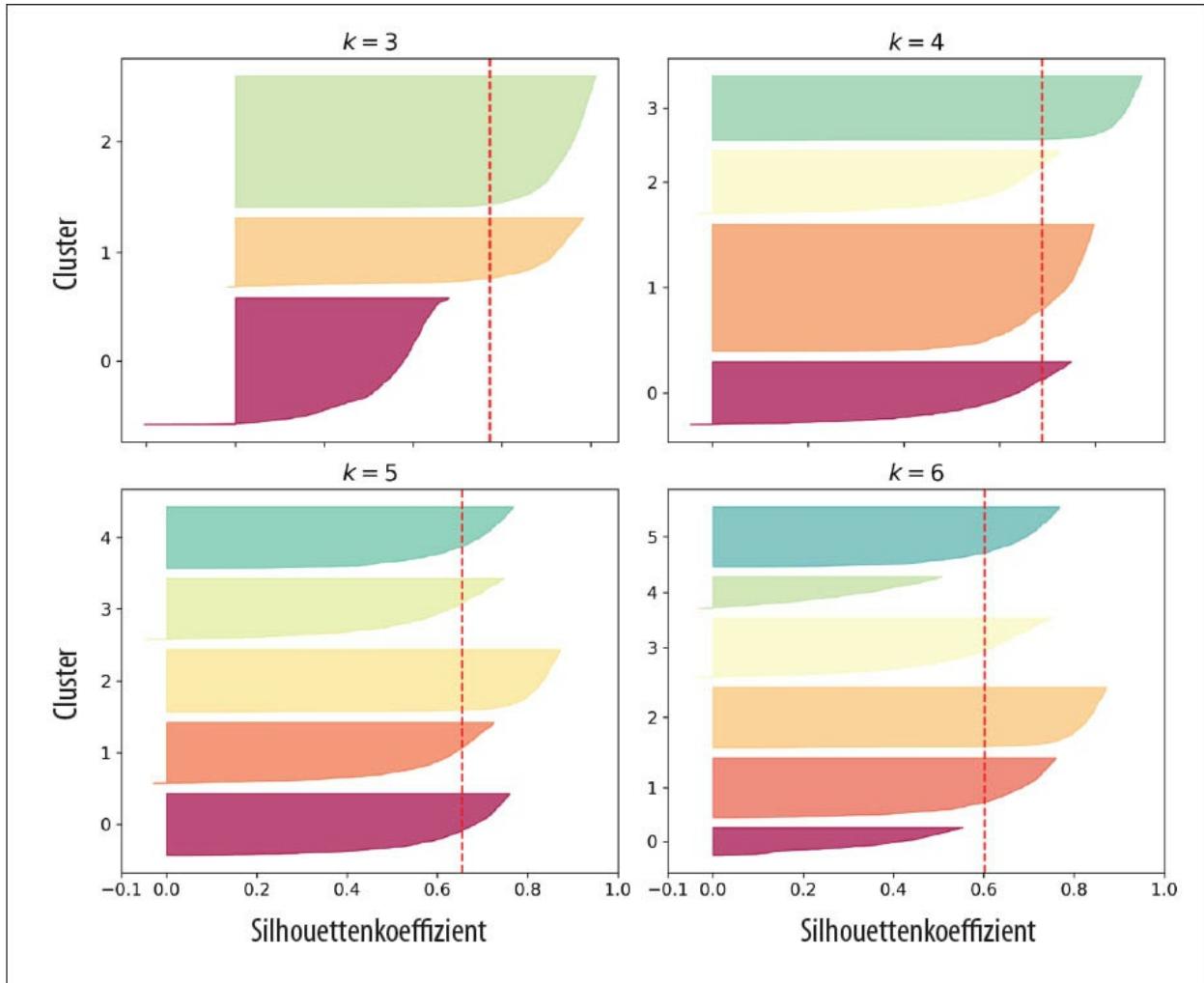


Abbildung 9-10: Analyse des Silhouettenplots für verschiedene Werte von k

Haben die meisten Instanzen in einem Cluster einen niedrigeren Silhouettenwert als dieser Koeffizient (wenn also viele von ihnen links davon enden), ist das Cluster eher schlecht, denn seine Instanzen liegen viel zu nahe an anderen Clustern. Wir sehen, dass wir bei $k = 3$ oder $k = 6$ schlechte Cluster erhalten. Aber bei $k = 4$ oder $k = 5$ sehen die Cluster ziemlich gut aus: Die meisten Instanzen gehen über die gestrichelte Linie nach rechts (und damit näher an 1,0 hinaus). Bei $k = 4$ ist das Cluster bei Index 1 (das dritte von oben) ziemlich groß. Bei $k = 5$ sind alle Cluster ähnlich groß. Also auch wenn der gesamte Silhouettenkoeffizient bei $k = 4$ etwas größer ist als bei $k = 5$, scheint es eine gute Idee zu sein, $k = 5$ zu nutzen, um Cluster ähnlicher Größe zu

erhalten.

Grenzen von K-Means

Trotz seiner vielen Vorteile – insbesondere bei der Geschwindigkeit und der Skalierbarkeit – ist K-Means nicht perfekt. Wie wir gesehen haben, muss der Algorithmus mehrfach laufen, um suboptimale Lösungen zu vermeiden, zudem müssen Sie die Anzahl an Clustern festlegen, was selbst ziemlich schwierig sein kann. Und K-Means verhält sich nicht sehr gut, wenn die Cluster unterschiedliche Größen, Dichten oder nicht kreisförmige Formen haben. So zeigt beispielsweise [Abbildung 9-11](#), wie K-Means einen Datensatz mit drei ellipsenförmigen Clustern unterschiedlicher Größe, Dichte und Orientierung unterteilt.

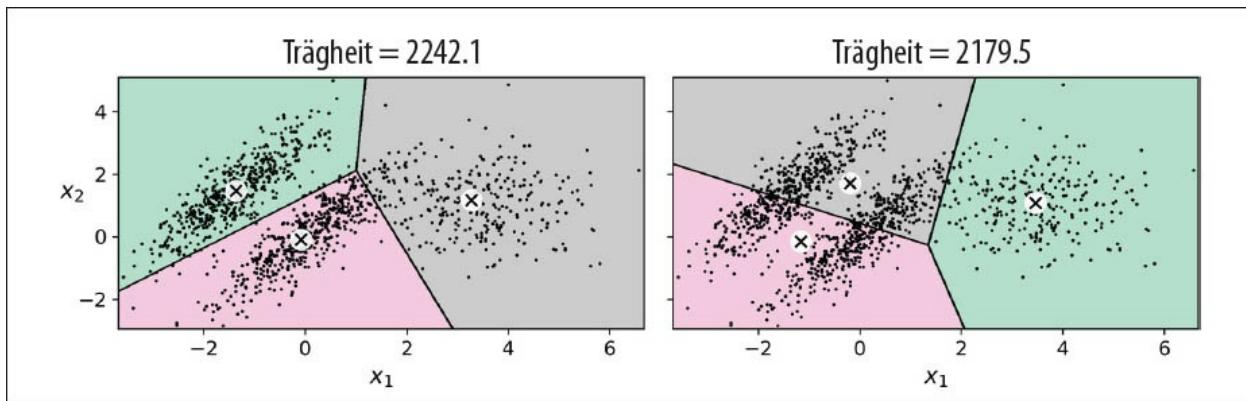


Abbildung 9-11: K-Means hat Probleme, diese ellipsenförmigen Bereiche sauber zu clustern.

Sie sehen, dass keine dieser Lösungen gut ist. Die linke Lösung ist besser, trennt aber immer noch 25% des mittleren Clusters ab und weist es dem rechten Cluster zu. Die rechte Lösung ist einfach nur furchtbar, auch wenn ihre Trägheit niedriger ist. Also können abhängig von den Daten verschiedene Clustering-Algorithmen besser funktionieren. Bei dieser Art ellipsenförmiger Cluster klappt es mit gaußschen Mischverteilungsmodellen wunderbar.

- ☞ Es ist wichtig, die Eingangsmerkmale zu skalieren, bevor Sie K-Means laufen lassen, denn sonst können die Cluster sehr gestreckt sein, und K-Means funktioniert nur schlecht. Das Skalieren der Merkmale garantiert nicht, dass alle Cluster nett und kreisförmig sind, aber es führt meist zu besseren Ergebnissen.

Schauen wir uns jetzt ein paar Möglichkeiten an, vom Clustering zu profitieren. Wir werden K-Means nutzen, aber Sie dürfen auch gern mit anderen Clustering-Algorithmen experimentieren.

Bildsegmentierung per Clustering

Bei der *Bildsegmentierung* geht es darum, ein Bild in mehrere Segmente zu unterteilen. Bei der *semantischen Segmentierung* werden alle Pixel, die Teil des gleichen Objekttyps sind, dem gleichen Segment zugewiesen. So können beispielsweise bei einem Sichtsystem für selbstfahrende Autos alle Pixel, die Teil eines Fußgängerbilds sind, dem Segment »Fußgänger« zugewiesen werden (das wäre ein Segment mit allen Fußgängern). Bei der *Instanzsegmentierung* werden alle Pixel, die Teil des gleichen individuellen Objekts sind, dem gleichen Segment

zugewiesen. In diesem Fall würde es ein eigenes Segment für jeden Fußgänger geben. Heutzutage erreichen die besten Systeme zur semantischen oder Instanzsegmentierung ihre Ergebnisse mithilfe komplexer Architekturen, die auf Convolutional Neural Networks basieren (siehe [Kapitel 14](#)). Hier werden wir viel einfacher vorgehen – mithilfe von *Farbsegmentierung*. Dabei weisen wir Pixeln dem gleichen Segment zu, wenn sie eine ähnliche Farbe haben. In manchen Anwendungen kann das schon ausreichen. Müssen Sie zum Beispiel Satellitenbilder analysieren, um zu messen, wie viel Wald es in einem Gebiet gibt, kann die Farbsegmentierung wunderbar funktionieren.

Nutzen wir zuerst die Matplotlib-Funktion `imread()`, um das Bild zu laden (siehe das obere linke Bild in [Abbildung 9-12](#)):

```
>>> from matplotlib.image import imread # oder `from imageio import imread`  
>>> image = imread(os.path.join("images", "unsupervised_learning", "ladybug.png"))  
>>> image.shape  
(533, 800, 3)
```

Das Bild wird als 3-D-Array repräsentiert. Die Größe der ersten Dimension ist die Höhe, die zweite die Breite und die dritte die Anzahl an Farbkanälen – in diesem Fall Rot, Grün und Blau (RGB). Mit anderen Worten: Für jedes Pixel gibt es einen 3-D-Vektor mit den Intensitäten von Rot, Grün und Blau jeweils zwischen 0,0 und 1,0 (oder zwischen 0 und 255, wenn Sie `imageio.imread()` verwenden). Manche Bilder haben vielleicht weniger Kanäle, wie beispielsweise Graustufenbilder (ein Kanal). Und manche Bilder können auch mehr Kanäle haben, wie zum Beispiel solche mit einem zusätzlichen *Alpha-Kanal* für die Transparenz oder Satellitenbilder, die oft zusätzliche Kanäle für verschiedene Frequenzen enthalten (zum Beispiel Infrarot). Der folgende Code passt das Array so an, dass eine lange Liste an RGB-Farben herauskommen, und clustert diese Farben dann mithilfe von K-Means:

```
x = image.reshape(-1, 3)  
kmeans = KMeans(n_clusters=8).fit(X)  
segmented_img = kmeans.cluster_centers_[kmeans.labels_]  
segmented_img = segmented_img.reshape(image.shape)
```

Beispielsweise kann so ein Farbcluster für alle Grüntöne gefunden werden. Dann wird für jede Farbe (zum Beispiel Dunkelgrün) nach der Durchschnittsfarbe des Farbclusters geschaut. So könnten alle Grünschattierungen durch das gleiche Hellgrün ersetzt werden (sofern das die Durchschnittsfarbe des grünen Clusters ist). Schließlich wird noch diese lange Liste an Farben so angepasst, dass sie wieder die gleiche Form wie das ursprüngliche Bild hat. Und wir sind fertig!

Das führt zu dem Bild, das Sie oben links in [Abbildung 9-12](#) sehen. Sie können mit unterschiedlich vielen Farbclustern experimentieren – so wie das auch in der Abbildung zu sehen

ist. Nutzen Sie weniger als acht Cluster, werden Sie feststellen, dass das knallige Rot des Marienkäfers kein eigenes Cluster mehr bekommt – es wird mit Farben aus der Umgebung verschmolzen. Das liegt daran, dass K-Means Cluster ähnlicher Größe bevorzugt. Der Marienkäfer ist klein, viel kleiner als der Rest des Bilds – selbst wenn seine Farbe sehr intensiv ist, kann K-Means ihr kein eigenes Cluster spendieren.

Das war nicht allzu schwer, oder? Schauen wir uns jetzt eine andere Anwendung für das Clustern an – eine Vorverarbeitung.

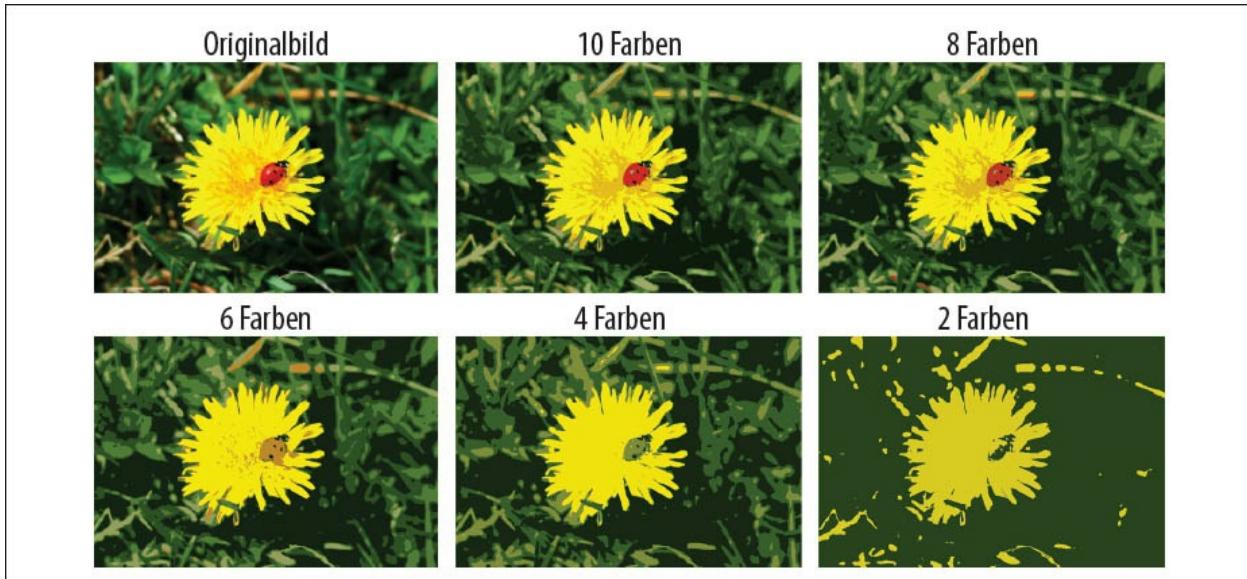


Abbildung 9-12: Bildsegmentierung mit K-Means mit verschiedenen vielen Farbclustern

Vorverarbeitung per Clustering

Das Clustering kann ein effizienter Ansatz für die Dimensionsreduktion sein, insbesondere als Vorverarbeitungsschritt vor einem überwachten Lernen. Als Beispiel für das Clustering zur Dimensionsreduktion wollen wir den Ziffern-Datensatz angehen, bei dem es sich um einfach um einen MNIST-ähnlichen Datensatz mit 1.797 Graustufenbildern der Größe 8×8 handelt und auf denen die Ziffern 0 bis 9 zu sehen sind. Laden wir als Erstes die Daten:

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
```

Jetzt teilen wir sie in einen Trainingsdatensatz und einen Testdatensatz auf:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits)
```

Als Nächstes passen wir ein logistisches Regressionsmodell an:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
```

Werten wir die Genauigkeit für den Testdatensatz aus:

```
>>> log_reg.score(X_test, y_test)
0.9688888888888889
```

Okay, das ist unser Ausgangspunkt: 96,9% Genauigkeit. Schauen wir, ob wir das durch den Einsatz von K-Means als Vorverarbeitungsschritt besser hinbekommen. Wir werden eine Pipeline erstellen, die erst den Trainingsdatensatz in 50 Cluster unterteilt und die Bilder durch ihre Distanzen zu diesen 50 Clustern ersetzt, um danach ein logistisches Regressionsmodell anzuwenden:

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression()),
])
pipeline.fit(X_train, y_train)
```



Da es zehn verschiedene Ziffern gibt, ist es verlockend, die Anzahl der Cluster auf 10 zu setzen. Aber jede Ziffer kann auf sehr unterschiedliche Art und Weise geschrieben werden, daher ist es besser, eine größere Zahl von Clustern – zum Beispiel 50 – zu verwenden.

Werten wir diese Klassifikationspipeline aus:

```
>>> pipeline.score(X_test, y_test)
0.9777777777777777
```

Na bitte. Wir haben die Fehlerrate um fast 30% reduziert (von etwa 3,1% auf etwa 2,2%)!

Aber wir haben die Anzahl k an Clustern einfach so festgelegt – das können wir bestimmt besser. Da K-Means nur ein Vorverarbeitungsschritt in einer Klassifikationspipeline ist, ist das Finden eines guten Werts für k viel einfacher als zuvor. Man muss keine Silhouettenanalyse durchführen oder die Trägheit minimieren – der beste Wert von k ist einfach der, der zu der besten

Klassifikation bei der Kreuzvalidierung führt. Wir können GridSearchCV nutzen, um die optimale Zahl von Clustern zu finden:

```
from sklearn.model_selection import GridSearchCV

param_grid = dict(kmeans__n_clusters=range(2, 100))

grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)

grid_clf.fit(X_train, y_train)
```

Schauen wir uns den besten Wert für k und die Qualität der entsprechenden Pipeline an:

```
>>> grid_clf.best_params_
{'kmeans__n_clusters': 99}

>>> grid_clf.score(X_test, y_test)
0.9822222222222222
```

Mit $k = 99$ Clustern erhalten wir eine deutlich bessere Genauigkeit und erreichen 98,22% für den Testdatensatz. Cool! Sie können auch versuchen, noch höhere Werte für k zu prüfen, da 99 der größte Wert im zu prüfenden Bereich war.

Clustering für teilüberwachtes Lernen einsetzen

Ein weiterer Anwendungsfall für das Clustering ist das teilüberwachte Lernen, bei dem wir viele ungelabelte und sehr wenige gelabelte Instanzen haben. Trainieren wir ein logistisches Regressionsmodell mit einem Beispieldatensatz aus 50 gelabelten Instanzen aus dem Zifferndatensatz:

```
n_labeled = 50

log_reg = LogisticRegression()

log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

Wie ist die Qualität dieses Modells für den Testdatensatz?

```
>>> log_reg.score(X_test, y_test)
0.8333333333333334
```

Die Genauigkeit beträgt nur 83,3%. Es sollte wenig überraschen, dass das viel schlechter als zuvor ist, da wir das Modell mit dem kompletten Trainingsdatensatz trainiert haben. Mal schauen, ob wir es besser hinbekommen. Zuerst teilen wir den Trainingsdatensatz in 50 Cluster auf. Dann finden wir für jedes Cluster das Bild, das möglichst nahe am Schwerpunkt liegt. Diese

Bilder bezeichnen wir als *repräsentative Bilder*:

```
k = 50  
  
kmeans = KMeans(n_clusters=k)  
  
X_digits_dist = kmeans.fit_transform(X_train)  
  
representative_digit_idx = np.argmin(X_digits_dist, axis=0)  
  
X_representative_digits = X_train[representative_digit_idx]
```

[Abbildung 9-13](#) zeigt diese 50 repräsentativen Bilder.

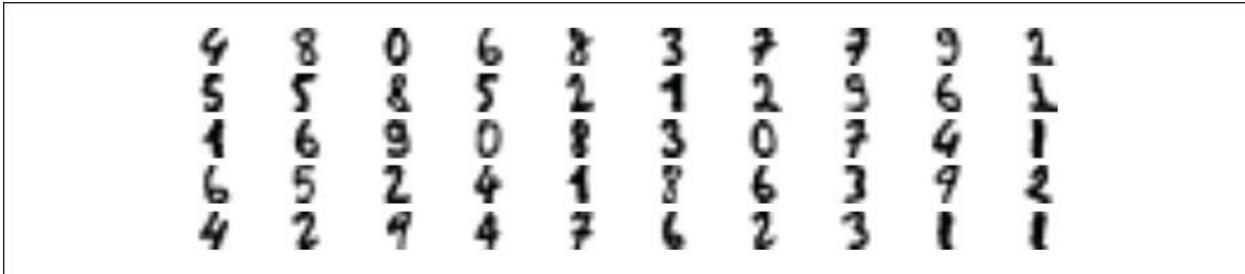


Abbildung 9-13: 50 repräsentative Ziffernbilder (eines pro Cluster)

Schauen wir uns jetzt jedes Bild einzeln an und verpassen wir ihm manuell ein Label:

```
y_representative_digits = np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])
```

Nun haben wir einen Datensatz mit nur 50 gelabelten Instanzen, aber statt dass diese zufällige Instanzen sind, handelt es sich bei jeder von ihnen um ein repräsentatives Bild seines Clusters. Mal sehen, ob die Qualität dadurch besser wird:

```
>>> log_reg = LogisticRegression()  
  
>>> log_reg.fit(X_representative_digits, y_representative_digits)  
  
>>> log_reg.score(X_test, y_test)  
  
0.9222222222222223
```

Wow, wir sind von 83,3% auf 92,2% Genauigkeit gesprungen, obwohl wir das Modell nur mit 50 Instanzen trainiert haben. Da es oft teuer und aufwendig ist, Instanzen zu labeln, insbesondere wenn das manuell von Experten vorgenommen werden muss, ist es von Vorteil, das mit repräsentativen statt mit zufälligen Instanzen zu tun.

Aber vielleicht können wir noch einen Schritt weitergehen: Was, wenn wir die Labels auf alle anderen Instanzen im gleichen Cluster anwenden? Das wird als *Label Propagation* bezeichnet:

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
```

```

for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]

```

Trainieren wir das Modell erneut und schauen wir uns seine Qualität an:

```

>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
>>> log_reg.score(X_test, y_test)
0.9333333333333333

```

Wir haben eine deutliche Steigerung der Genauigkeit erhalten, aber nichts wirklich Überraschendes. Das Problem liegt darin, dass wir die Labels jeder repräsentativen Instanz auf alle Instanzen im gleichen Cluster übertragen haben – auch auf die, die nahe an den Cluster-Grenzen liegen und die dieses Label vielleicht eher nicht verdient haben. Schauen wir uns an, was passiert, wenn wir die Labels nur an die 20% der Instanzen weiterreichen, die am nächsten an den Schwerpunkten liegen:

```

percentile_closest = 20

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]

for i in range(k):
    in_cluster = (kmeans.labels_ == i)

    cluster_dist = X_cluster_dist[in_cluster]

    cutoff_distance = np.percentile(cluster_dist, percentile_closest)

    above_cutoff = (X_cluster_dist > cutoff_distance)

    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)

X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]

```

Trainieren wir das Modell jetzt erneut mit diesem teilweise propagierten Datensatz:

```

>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)

```

```
>>> log_reg.score(X_test, y_test)
```

```
0.94
```

Nett! Mit nur 50 gelabelten Instanzen (nur fünf Beispiele pro Klasse im Schnitt!) haben wir 94,0% Genauigkeit erhalten, was ziemlich nahe an der Leistung der logistischen Regression mit dem vollständig gelabelten Ziffern-Datensatz liegt (die 96,9% betrug). Diese gute Performance ist der Tatsache geschuldet, dass die weitergegebenen Labels tatsächlich ziemlich gut sind – ihre Genauigkeit ist nahe an 99 %, wie der folgende Code zeigt:

```
>>> np.mean(y_train_partially_propagated == y_train[partially_propagated])
```

```
0.9896907216494846
```

Aktives Lernen

Um Ihr Modell und Ihren Trainingsdatensatz weiter zu verbessern, kann der nächste Schritt darin bestehen, ein paar Runden *aktives Lernen* durchzuführen, bei dem ein menschlicher Experte mit dem Lernalgorithmus interagiert und Labels für bestimmte Instanzen vergibt, die der Algorithmus anfragt. Es gibt viele verschiedene Strategien zum aktiven Lernen, aber eine der verbreitetsten wird als *Uncertainty Sampling* bezeichnet. So funktioniert es:

- Das Modell wird mit den bisher gelabelten Instanzen trainiert und dann genutzt, um Vorhersagen für alle noch nicht gelabelten Instanzen zu treffen.
- Die Instanzen, für die das Modell am unsichersten ist (also bei denen die geschätzte Wahrscheinlichkeit am niedrigsten ist), werden den Experten zum Labeln gegeben.
- Sie iterieren durch diesen Prozess, bis die Performance nicht mehr so viel besser wird, dass sich der Labeling-Aufwand lohnt.

Bei anderen Strategien werden die Instanzen gelabelt, die zur stärksten Modelländerung führen würden oder zur größten Verbesserung beim Validierungsfehler, oder es werden die Instanzen genommen, für die verschiedene Modelle unterschiedlicher Ansicht sind (zum Beispiel eine SVM und ein Random Forest).

Bevor wir uns den gaußschen Mischverteilungsmodellen zuwenden, wollen wir noch einen Blick auf DBSCAN werfen – einen weiteren beliebten Clustering-Algorithmus, der ein ganz anderes Vorgehen nutzt, das auf lokalen Dichteabschätzungen basiert. Dieser Ansatz erlaubt es dem Algorithmus, Cluster beliebiger Form zu identifizieren.

DBSCAN

Dieser Algorithmus definiert Cluster als kontinuierliche Regionen hoher Dichte. Und so funktioniert er:

- Für jede Instanz zählt der Algorithmus, wie viele Instanzen es in einem kleinen Abstand

ε (Epsilon) zu ihr gibt. Diese Region wird dann als ε -Nachbarschaft der Instanz bezeichnet.

- Hat eine Instanz mindestens `min_samples` Instanzen in ihrer ε -Nachbarschaft (einschließlich sich selbst), wird sie als *Kerninstanz* angesehen. Mit anderen Worten: Kerninstanzen sind solche, die sich in dichten Regionen befinden.
- Alle Instanzen in der Nachbarschaft einer Kerninstanz gehören zum gleichen Cluster. Diese Nachbarschaft kann andere Kerninstanzen beinhalten – daher bildet eine lange Folge von benachbarten Kerninstanzen ein einzelnes Cluster.
- Jede Instanz, bei der es sich nicht um eine Kerninstanz handelt und die auch keine in ihrer Nachbarschaft vorfindet, wird als Anomalie betrachtet.

Dieser Algorithmus funktioniert gut, wenn alle Cluster dicht genug sind und sie durch Regionen geringer Dichte gut voneinander getrennt werden. Die Klasse `DBSCAN` in Scikit-Learn ist so einfach, wie Sie es wahrscheinlich auch erwarten. Testen wir sie mit dem moons-Datensatz, der in [Kapitel 5](#) vorgestellt wurde:

```
from sklearn.cluster import DBSCAN

from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)

dbSCAN = DBSCAN(eps=0.05, min_samples=5)

dbSCAN.fit(X)
```

Die Labels aller Instanzen finden Sie nun in der Instanzvariablen `labels_`:

```
>>> dbSCAN.labels_

array([ 0,  2, -1, -1,  1,  0,  0,  0, ...,  3,  2,  3,  3,  4,  2,  6,  3])
```

Beachten Sie, dass manche Instanzen einen Cluster-Index -1 besitzen, was heißt, dass sie vom Algorithmus als Anomalien betrachtet werden. Die Indizes der Kerninstanzen finden sich in der Instanzvariablen `core_sample_indices_`, die Kerninstanzen selbst in `components_`:

```
>>> len(dbSCAN.core_sample_indices_)

808

>>> dbSCAN.core_sample_indices_

array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])

>>> dbSCAN.components_

array([[ -0.02137124,  0.40618608],
```

```

[-0.84192557, 0.53058695],  

...  

[-0.94355873, 0.3278936 ],  

[ 0.79419406, 0.60777171]])

```

Dieses Clustering wird im linken Plot von [Abbildung 9-14](#) dargestellt. Wie Sie sehen, findet es ziemlich viele Anomalien und dazu sieben verschiedene Cluster. Wie enttäuschend! Zum Glück reicht es, die Nachbarschaft jeder Instanz zu vergrößern, indem wir `eps` auf 0,2 setzen, und schon funktioniert das Clustering perfekt. Machen wir mit diesem Modell weiter.

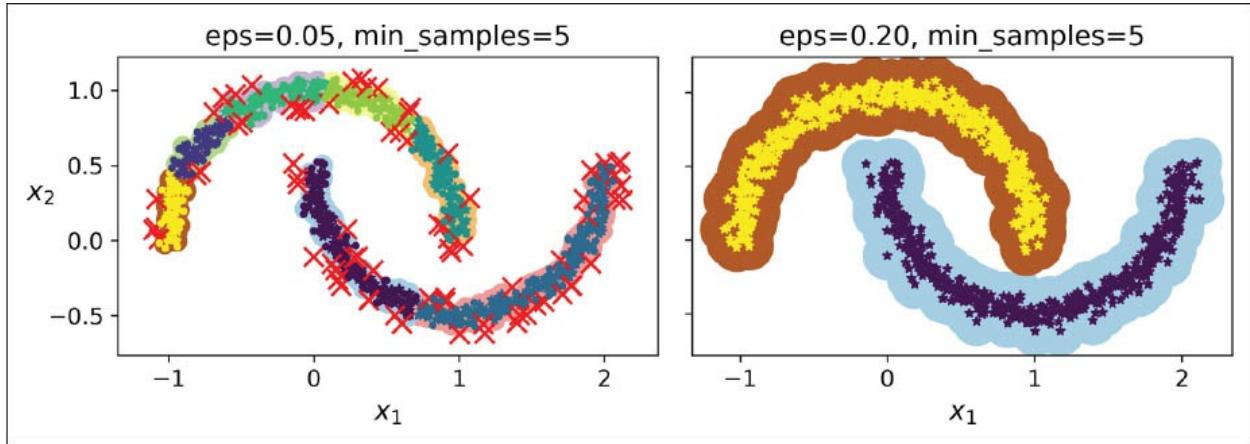


Abbildung 9-14: DBSCAN-Clustering mit zwei verschiedenen Nachbarschaftsradien

Ein wenig überraschend besitzt die Klasse `DBSCAN` keine Methode `predict()`, aber es gibt eine Methode `fit_predict()`. Mit anderen Worten: Sie kann nicht vorhersagen, zu welchem Cluster eine neue Instanz gehört. Diese Implementierungsentscheidung wurde getroffen, weil für verschiedene Aufgaben unterschiedliche Klassifizierungsalgorithmen sinnvoller sein können. Daher haben die Autoren entschieden, den Anwender auswählen zu lassen. Zudem ist die Methode nicht schwierig zu implementieren. Trainieren wir beispielsweise einen `KNeighborsClassifier`:

```

from sklearn.neighbors import KNeighborsClassifier  
  

knn = KNeighborsClassifier(n_neighbors=50)  
  

knn.fit(dbSCAN.components_, dbSCAN.labels_[dbSCAN.core_sample_indices_])

```

Mit ein paar neuen Instanzen können wir jetzt vorhersagen, zu welchem Cluster sie am ehesten gehören werden, und sogar eine Wahrscheinlichkeit für jedes Cluster angeben:

```

>>> X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])  

>>> knn.predict(X_new)

```

```

array([1, 0, 1, 0])

>>> knn.predict_proba(X_new)

array([[0.18, 0.82],
       [1. , 0. ],
       [0.12, 0.88],
       [1. , 0. ]])

```

Beachten Sie, dass wir den Klassifikator nur mit den Kerninstanzen trainiert haben. Wir hätten uns aber auch dazu entscheiden können, ihn mit allen Instanzen zu trainieren – oder mit allen ohne die Anomalien: Diese Entscheidung hängt von der eigentlichen Aufgabe ab.

Die Entscheidungsgrenze wird in [Abbildung 9-15](#) dargestellt (die Kreuze stehen für die vier Instanzen in X_{new}). Da es keine Anomalien im Trainingsdatensatz gibt, wählt der Klassifikator immer ein Cluster, auch wenn dieses weit weg ist. Man kann leicht einen maximalen Abstand einbauen, sodass dann die beiden Instanzen, die von beiden Clustern weit entfernt sind, als Anomalien klassifiziert werden. Dazu nutzen Sie die Methode `kneighbors()` des `KNeighborsClassifier`. Gibt man ihr einen Satz Instanzen, liefert sie die Distanzen und Indizes der k nächsten Nachbarn im Trainingsdatensatz (zwei Matrizen, jede mit k Spalten):

```

>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)

>>> y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]

>>> y_pred[y_dist > 0.2] = -1

>>> y_pred.ravel()

array([-1, 0, 1, -1])

```

Zusammengefasst, handelt es sich bei DBSCAN um einen sehr einfachen, trotzdem aber sehr leistungsfähigen Algorithmus, der eine beliebige Zahl von Clustern beliebiger Form erkennen kann. Er reagiert robust auf Ausreißer und besitzt nur zwei Hyperparameter (`eps` und `min_samples`). Variiert die Dichte zwischen den Clustern deutlich, kann es aber unmöglich werden, alle Cluster sauber zu erkennen. Seine Rechenkomplexität ist ungefähr $O(m \log m)$, was ihn in Bezug auf die Instanzen fast linear skalieren lässt. Die Implementierung in Scikit-Learn kann aber bis zu $O(m^2)$ erfordern, wenn `eps` groß ist.

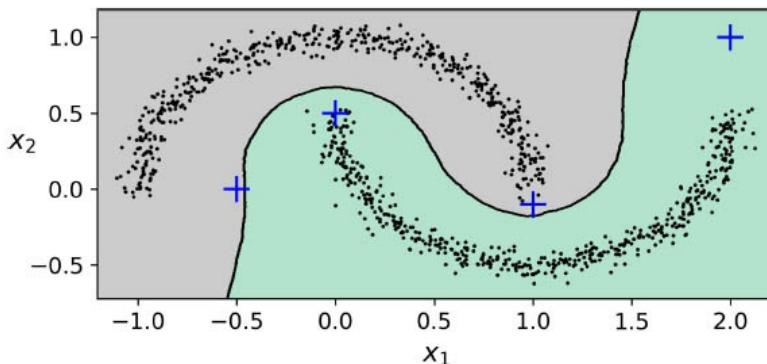


Abbildung 9-15: Entscheidungsgrenze zwischen zwei Clustern



Sie können auch den *Hierarchical DBSCAN* (HDBSCAN) ausprobieren, der im Projekt scikit-learn-contrib (<https://github.com/scikit-learn-contrib/hdbscan/>) implementiert ist.

Andere Clustering-Algorithmen

Scikit-Learn implementiert eine Reihe weiterer Clustering-Algorithmen, die Sie sich einmal anschauen sollten. Wir können sie hier zwar nicht alle im Detail behandeln, aber zumindest einen kurzen Überblick geben:

Agglomeratives Clustering

Hier wird eine Hierarchie aus Clustern von klein nach groß aufgebaut (»Bottom Up«). Stellen Sie sich viele kleine Bläschen vor, die auf dem Wasser schwimmen und sich nach und nach miteinander verbinden, bis es eine große Gruppe von Bläschen gibt. Genauso verbindet agglomeratives Clustering bei jeder Iteration das am nächsten beieinanderliegende Paar von Clustern (beginnend bei den individuellen Instanzen). Würden Sie einen Baum mit einem Zweig für jedes Paar zusammengeführter Cluster zeichnen, erhalten Sie einen Binärbaum aus Clustern, bei dem die Blätter die einzelnen Instanzen sind. Dieses Vorgehen skaliert sehr gut für große Mengen an Instanzen oder Clustern. Es kann Cluster unterschiedlichster Form erfassen, erzeugt einen flexiblen und informativen Cluster-Baum, statt Sie dazu zu zwingen, eine bestimmte Cluster-Größe zu wählen, und es kann mit beliebigen paarweisen Abständen genutzt werden. Bei sehr vielen Instanzen funktioniert der Algorithmus gut, wenn Sie eine Adjazenzmatrix bereitstellen, bei der es sich um eine dünn besetzte $m \times m$ -Matrix handelt (zum Beispiel aus `sklearn.neighbors.kneighbors_graph()`). Ohne eine Adjazenzmatrix skaliert der Algorithmus für große Datensätze nicht gut.

BIRCH

Der BIRCH-Algorithmus (Balanced Iterative Reducing and Clustering using Hierarchies) wurde speziell für sehr große Datensätze entworfen und kann mit ähnlichen Ergebnissen schneller als Batch-K-Means sein, sofern die Menge an Merkmalen nicht zu groß ist (unter 20). Während des Trainings baut er eine Baumstruktur auf, die gerade genug Informationen enthält, um eine neue Instanz schnell einem Cluster zuzuweisen, ohne alle Instanzen im

Baum speichern zu müssen. Dieses Vorgehen erlaubt die Verwendung begrenzter Speicherressourcen beim Umgang mit riesigen Datensätzen.

Mean-Shift

Dieser Algorithmus beginnt damit, einen Kreis um jede Instanz zu ziehen. Dann berechnet er für jeden Kreis den Mittelwert aller Instanzen, die sich in ihm befinden, und verschiebt den Kreis so, dass sein Mittelpunkt auf diesem Mittelwert liegt. Als Nächstes iteriert er diesen Mean-Shift-Schritt, bis sich keine Kreise mehr bewegen (also bis jeder von ihnen auf dem Mittelwert der in ihm enthaltenen Instanzen zentriert ist). Mean-Shift verschiebt die Kreise in die Richtung höherer Dichte, bis jeder ein lokales Dichtemaximum gefunden hat. Und schließlich werden alle Instanzen, deren Kreise sich an der gleichen Stelle (oder nahe genug beieinander) angesammelt haben, dem gleichen Cluster zugewiesen. Mean-Shift besitzt ein paar ähnliche Features wie DBSCAN – so kann es eine beliebige Zahl von Clustern beliebiger Form finden, besitzt nur sehr wenige Hyperparameter (nur einen: den Radius des Kreises, als *Bandbreite* bezeichnet) und baut auf der Schätzung der lokalen Dichte auf. Aber anders als DBSCAN tendiert Mean-Shift dazu, Cluster in einzelne Stücke aufzuteilen, wenn sie interne Dichteunterschiede aufweisen. Leider ist seine Rechenkomplexität $O(m^2)$, daher ist dieser Algorithmus für große Datensätze nicht passend.

Affinity Propagation

Dieser Algorithmus nutzt ein Abstimmungssystem, bei dem die Instanzen über ähnliche Instanzen abstimmen, die ihre Repräsentanten sein sollen. Konvergiert der Algorithmus, bildet jeder Repräsentant mit seinen Wählern ein Cluster. Affinity Propagation kann beliebig viele Cluster unterschiedlicher Größe finden. Leider hat dieser Algorithmus eine Rechenkomplexität von $O(m^2)$, daher ist er für große Datensätze nicht so geeignet.

Spektrales Clustering

Dieser Algorithmus nimmt eine Ähnlichkeitsmatrix zwischen den Instanzen und erstellt daraus ein niedrigdimensionales Embedding (reduziert also ihre Dimensionalität), dann verwendet er einen anderen Clustering-Algorithmus in diesem niedrigdimensionalen Raum (die Implementierung von Scikit-Learn verwendet K-Means). Spektrales Clustering kann komplexe Cluster-Strukturen erfassen und auch genutzt werden, um Graphen aufzuteilen (um zum Beispiel Freunde-Cluster in einem sozialen Netzwerk zu identifizieren). Der Algorithmus skaliert nicht gut für große Mengen an Instanzen und verhält sich nicht gut, wenn die Cluster sehr unterschiedliche Größen haben.

Kommen wir nun zu den gaußschen Mischverteilungsmodellen, die zur Dichteabschätzung, zum Clustering und zur Anomalieerkennung genutzt werden können.

Gaußsche Mischverteilung

Ein *gaußsches Mischverteilungsmodell* (Gaussian Mixture Model, GMM) ist ein Wahrscheinlichkeitsmodell, das davon ausgeht, dass die Instanzen aus einer Mischung diverser gaußscher Verteilungen erzeugt wurden, deren Parameter unbekannt sind. Alle Instanzen, die aus einer einzelnen gaußschen Verteilung generiert wurden, bilden ein Cluster, das typischerweise wie ein Ellipsoid aussieht. Jedes Cluster kann eine andere Ellipsenform, Größe, Dichte und

Ausrichtung haben, so wie in [Abbildung 9-11](#). Beobachten Sie eine Instanz, wissen Sie, dass sie aus einer der gaußschen Verteilungen erzeugt wurde, aber Sie wissen nicht, aus welcher, und auch die Parameter dieser Verteilung sind Ihnen nicht bekannt.

Es gibt diverse Varianten des gaußschen Mischverteilungsmodells. In der einfachsten Form – implementiert in der Klasse `GaussianMixture` – müssen Sie die Anzahl k der gaußschen Verteilungen wissen. Vom Datensatz \mathbf{X} wird angenommen, dass er durch den folgenden Wahrscheinlichkeitsprozess erzeugt wurde:

- Für jede Instanz wird ein Cluster zufällig aus k Clustern gewählt. Die Wahrscheinlichkeit, das j . Cluster zu nehmen, ist durch das Gewicht $\phi^{(j)}$ des Clusters definiert.⁷ Der Index des für die i . Instanz gewählten Clusters wird als $z^{(i)}$ bezeichnet.
- Ist $z^{(i)}=j$, wurde also die i . Instanz dem j . Cluster zugewiesen, wird die Position $\mathbf{x}^{(i)}$ dieser Instanz zufällig aus der gaußschen Verteilung mit dem Mittelwert $\boldsymbol{\mu}^{(j)}$ und der Kovarianzmatrix $\boldsymbol{\Sigma}^{(j)}$ bestimmt. Als Formel: $\mathbf{x}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)})$

Dieser generative Prozess lässt sich als grafisches Modell darstellen. In [Abbildung 9-16](#) sehen Sie die Struktur der bedingten Abhängigkeiten zwischen Zufallsvariablen.

So können Sie die Abbildung interpretieren:⁸

- Die Kreise stehen für Zufallsvariablen.
- Die Quadrate sind feste Werte (also Parameter des Modells).
- Die großen Rechtecke werden als *Plates* bezeichnet. Sie zeigen an, dass ihr Inhalt mehrfach wiederholt wird.
- Die Zahl unten rechts auf jeder Plate gibt an, wie oft der Inhalt wiederholt wird. Es gibt m Zufallsvariablen $z^{(i)}$ (von $z^{(1)}$ bis $z^{(m)}$) und m Zufallsvariablen $\mathbf{x}^{(i)}$. Zudem gibt es k Mittelwerte $\boldsymbol{\mu}^{(j)}$ und k Kovarianzmatrizen $\boldsymbol{\Sigma}^{(j)}$. Und schließlich gibt es nur einen Gewichtsvektor $\boldsymbol{\phi}$ mit allen Gewichten $\phi^{(1)}$ bis $\phi^{(k)}$.
- Jede Variable $z^{(i)}$ wird aus der kategorialen Verteilung mit den Gewichten $\boldsymbol{\phi}$ gezogen. Jede Variable $\mathbf{x}^{(i)}$ wird aus der Normalverteilung gezogen, wobei Mittelwert und Kovarianzmatrix durch ihr Cluster $z^{(i)}$ definiert sind.
- Die Pfeile mit durchgezogenen Linien stehen für bedingte Abhängigkeiten. So hängt beispielsweise die Wahrscheinlichkeitsverteilung für jede Zufallsvariable $z^{(i)}$ vom Gewichtsvektor $\boldsymbol{\phi}$ ab. Beachten Sie: Kreuzt ein Pfeil die Grenze einer Plate, wird er auf alle Wiederholungen dieser Plate angewendet. So beeinflusst beispielsweise der Gewichtsvektor $\boldsymbol{\phi}$ die Wahrscheinlichkeitsverteilungen aller Zufallsvariablen $\mathbf{x}^{(1)}$ bis $\mathbf{x}^{(m)}$.
- Der gewellte Pfeil von $z^{(i)}$ $\mathbf{x}^{(i)}$ nach $\mathbf{x}^{(i)}$ steht für einen Schalter: Abhängig vom Wert von $z^{(i)}$ wird die Instanz $\mathbf{x}^{(i)}$ aus einer unterschiedlichen gaußschen Verteilung ermittelt. Ist beispielsweise $z^{(i)}=j$, gilt $\mathbf{x}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)})$.
- Schattierte Knoten geben an, dass der Wert bekannt ist. In diesem Fall haben nur die Zufallsvariablen $\mathbf{x}^{(i)}$ bekannte Werte: Sie werden als *beobachtete Variablen* bezeichnet. Die unbekannten Zufallsvariablen $z^{(i)}$ heißen *latente Variablen*.

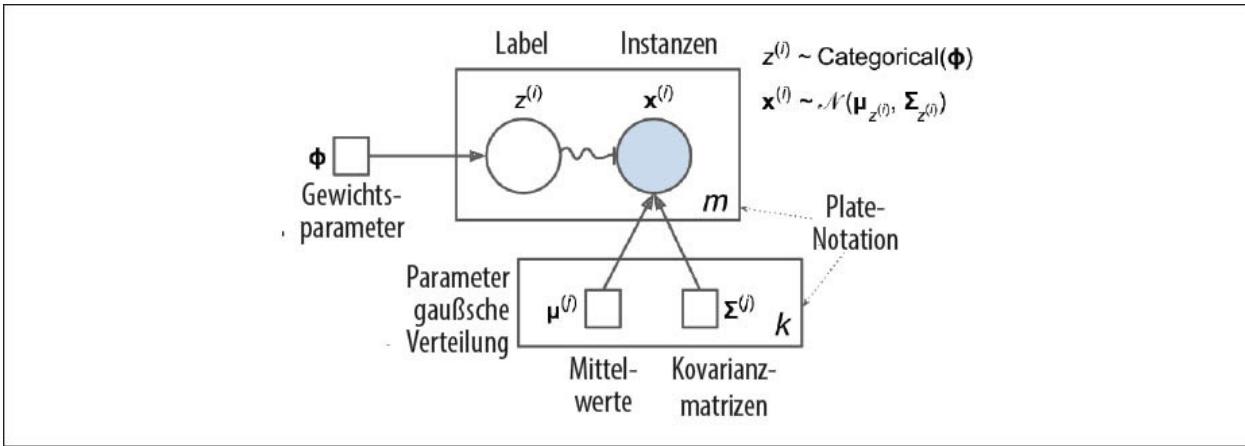


Abbildung 9-16: Eine grafische Darstellung des gaußschen Mischverteilungsmodells mit seinen Parametern (Quadrate), Zufallsvariablen (Kreise) und ihren bedingten Abhängigkeiten (Pfeile mit durchgehender Linie)

Was können Sie nun mit solch einem Modell tun? Nun, bei einem gegebenen Datensatz \mathbf{X} wollen Sie typischerweise damit beginnen, die Gewichte ϕ und alle Verteilungsparameter $\mu^{(1)}$ bis $\mu^{(k)}$ und $\Sigma^{(1)}$ bis $\Sigma^{(k)}$ zu schätzen. Die Klasse Gaussian Mixture von Scikit-Learn macht das ganz einfach:

```
from sklearn.mixture import GaussianMixture
```

```
gm = GaussianMixture(n_components=3, n_init=10)
```

```
gm.fit(X)
```

Schauen wir uns die Parameter an, die der Algorithmus geschätzt hat:

```
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
```

```
>>> gm.means_
array([[ 3.39909717,  1.05933727],
       [-1.40763984,  1.42710194],
       [ 0.05135313,  0.07524095]])
```

```
>>> gm.covariances_
array([[[ 1.14807234, -0.03270354],
       [-0.03270354,  0.95496237]],
      [[ 0.63478101,  0.72969804],
```

```
[ 0.72969804,  1.1609872 ]],  
[[ 0.68809572,  0.79608475],  
[ 0.79608475,  1.21234145]]])
```

Na bitte, das hat wunderbar funktioniert! Tasächlich waren die Gewichte, die wir zum Erzeugen der Daten genutzt haben, 0,2, 0,4 und 0,4 – und die Mittelwerte und Kovarianzmatrizen lagen auch sehr dicht an denen, die der Algorithmus gefunden hat. Aber wie? Diese Klasse greift auf den Expectation-Maximization-(EM-)Algorithmus zurück, der in vielem dem K-Means-Algorithmus ähnelt: Er initialisiert ebenfalls die Cluster-Parameter zufällig und wiederholt dann zwei Schritte zur Konvergenz. Zuerst weist er Clustern Instanzen zu (das wird als *Expectation-Schritt* bezeichnet), dann aktualisiert er die Cluster (das wird *Maximization-Schritt* genannt). Klingt vertraut, oder? In Kontext des Clusterings können Sie sich EM als Verallgemeinerung von K-Means vorstellen, das nicht nur die Cluster-Zentren findet ($\mu^{(1)}$ bis $\mu^{(k)}$), sondern auch ihre Größe, Form und Orientierung ($\Sigma^{(1)}$ bis $\Sigma^{(k)}$) sowie ihre relativen Gewichte ($\phi^{(1)}$ bis $\phi^{(k)}$). Anders als K-Means nutzt EM allerdings eine Softcluster-Zuweisung, keine harte Zuweisung. Für jede Instanz schätzt der Algorithmus während des Expectation-Schritts die Wahrscheinlichkeit der Zugehörigkeit zu jedem Cluster (basierend auf den aktuellen Cluster-Parametern). Dann wird während des Maximization-Schritts jedes Cluster mit *allen* Instanzen des Datensatzes aktualisiert, wobei jede Instanz anhand der geschätzten Wahrscheinlichkeit gewichtet wird, mit der sie zum Cluster gehört. Diese Wahrscheinlichkeiten werden als *Responsibilities* des Clusters für die Instanzen bezeichnet. Während des Maximization-Schritts wird jedes Cluster beim Aktualisieren vor allem von den Instanzen beeinflusst, für die es am verantwortlichsten ist.



Leider kann EM wie K-Means zu schlechten Lösungen konvergieren, daher muss es mehrfach ausgeführt und nur die beste Lösung muss behalten werden. Darum setzen wir `n_init` auf 10. Seien Sie vorsichtig: Standardmäßig ist `n_init` auf 1 gesetzt.

Sie können prüfen, ob der Algorithmus konvergiert ist und wie viele Iterationen genutzt wurden:

```
>>> gm.converged_
True
>>> gm.n_iter_
3
```

Nachdem Sie nun eine Schätzung für Position, Größe, Form, Orientierung und relative Gewichte jedes Clusters besitzen, kann das Modell problemlos jede Instanz dem wahrscheinlichsten Cluster zuweisen (Hard Clustering) oder die Wahrscheinlichkeit abschätzen, dass sie zu einem bestimmten Cluster gehört (Soft Clustering). Nutzen Sie einfach die Methode `predict()` für das Hard Clustering oder `predict_proba()` für das Soft Clustering:

```

>>> gm.predict(X)
array([2, 2, 1, ..., 0, 0, 0])

>>> gm.predict_proba(X)
array([[2.32389467e-02, 6.77397850e-07, 9.76760376e-01],
       [1.64685609e-02, 6.75361303e-04, 9.82856078e-01],
       [2.01535333e-06, 9.99923053e-01, 7.49319577e-05],
       ...,
       [9.99999571e-01, 2.13946075e-26, 4.28788333e-07],
       [1.00000000e+00, 1.46454409e-41, 5.12459171e-16],
       [1.00000000e+00, 8.02006365e-41, 2.27626238e-15]])

```

Ein gaußsches Mischverteilungsmodell ist ein *generatives Modell*, Sie können also neue Instanzen daraus erstellen (beachten Sie, dass sie anhand des Cluster-Index geordnet sind):

```

>>> X_new, y_new = gm.sample(6)

>>> X_new
array([[ 2.95400315,  2.63680992],
       [-1.16654575,  1.62792705],
       [-1.39477712, -1.48511338],
       [ 0.27221525,  0.690366 ],
       [ 0.54095936,  0.48591934],
       [ 0.38064009, -0.56240465]])

>>> y_new
array([0, 1, 2, 2, 2, 2])

```

Es ist zudem möglich, die Dichte des Modells an jedem gegebenen Ort zu schätzen. Dies wird mithilfe der Methode `score_samples()` erreicht: Für jede gegebene Instanz schätzt diese Methode den Logarithmus der Wahrscheinlichkeitsdichtefunktion (WDF) an diesem Ort. Je größer der Wert, desto höher die Dichte:

```

>>> gm.score_samples(X)
array([-2.60782346, -3.57106041, -3.33003479, ..., -3.51352783,
      ...
      ])

```

-4.39802535, -3.80743859])

Berechnen Sie die Exponentialfunktion dieser Werte, erhalten Sie den Wert der WDF an den Orten der angegebenen Instanzen. Das sind keine Wahrscheinlichkeiten, sondern *Wahrscheinlichkeitsdichten*: Sie können jeden positiven Wert annehmen, nicht nur solche zwischen 0 und 1. Um die Wahrscheinlichkeit abzuschätzen, dass eine Instanz in einem bestimmten Bereich liegen wird, müssten Sie die WDF über diesen Bereich integrieren (wenn Sie das über den gesamten Raum möglicher Instanzpositionen tun, wird das Ergebnis 1 sein).

[Abbildung 9-17](#) zeigt die Cluster-Mittelwerte, die Entscheidungsgrenzen (gestrichelte Linien) und die Dichtekonturen dieses Modells.

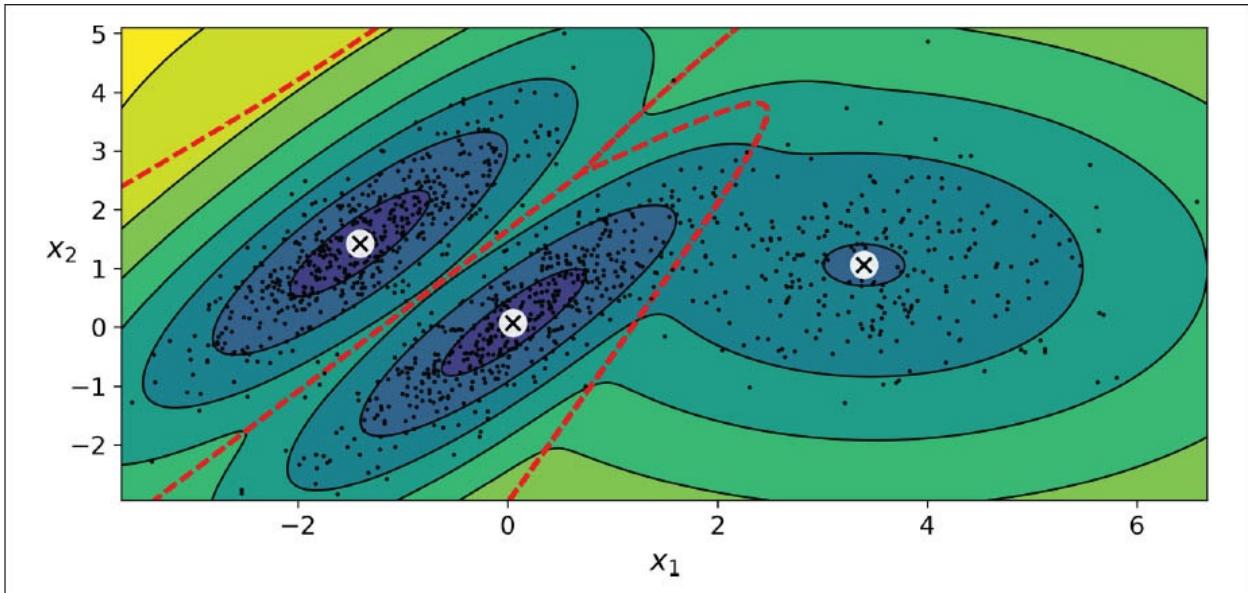


Abbildung 9-17: Cluster-Mittelwerte, Entscheidungsgrenzen und Dichtekonturen eines trainierten gaußschen Mischverteilungsmodells

Sehr schick! Der Algorithmus hat offensichtlich eine ausgezeichnete Lösung gefunden. Natürlich war die Aufgabe leichter, weil wir die Daten mit einem Satz zweidimensionaler Gaußverteilungen erzeugt haben (leider sind echte Daten nicht immer so gaußförmig verteilt und niedrigdimensional). Wir haben dem Algorithmus zudem die richtige Zahl an Clustern mitgegeben. Gibt es viele Dimensionen, viele Cluster oder wenige Instanzen, kann EM Probleme damit haben, zu einer optimalen Lösung zu konvergieren. Sie müssen dann eventuell die Schwierigkeit der Aufgabe reduzieren, indem Sie die Anzahl der Parameter begrenzen, die der Algorithmus zu lernen hat. Eine Möglichkeit dazu ist das Beschränken der Formen und Orientierungen, die die Cluster haben können. Das lässt sich erreichen, indem Sie Begrenzungen für die Kovarianzmatrizen vorgeben. Dazu setzen Sie den Hyperparameter `covariance_type` auf einen der folgenden Werte:

"spherical"

Alle Cluster müssen sphärisch sein, aber sie können unterschiedliche Durchmesser (also unterschiedliche Varianzen) haben.

"diag"

Cluster können eine beliebige ellipsoide Form jeglicher Größe annehmen, aber die Achsen der Ellipsoide müssen parallel zu den Koordinatenachsen sein (die Kovarianzmatrizen müssen also diagonal sein).

"tied"

Alle Cluster müssen die gleiche ellipsoide Form, Größe und Orientierung haben (sie müssen also alle die gleiche Kovarianzmatrix besitzen).

Standardmäßig ist `covariance_type` auf "full" gesetzt – jedes Cluster kann also eine beliebige Form, Größe und Orientierung annehmen (es hat seine eigene, uneingeschränkte Kovarianzmatrix). [Abbildung 9-18](#) gibt die Lösungen aus, die vom EM-Algorithmus gefunden werden, wenn `covariance_type` auf "tied" oder "spherical" gesetzt wurde.

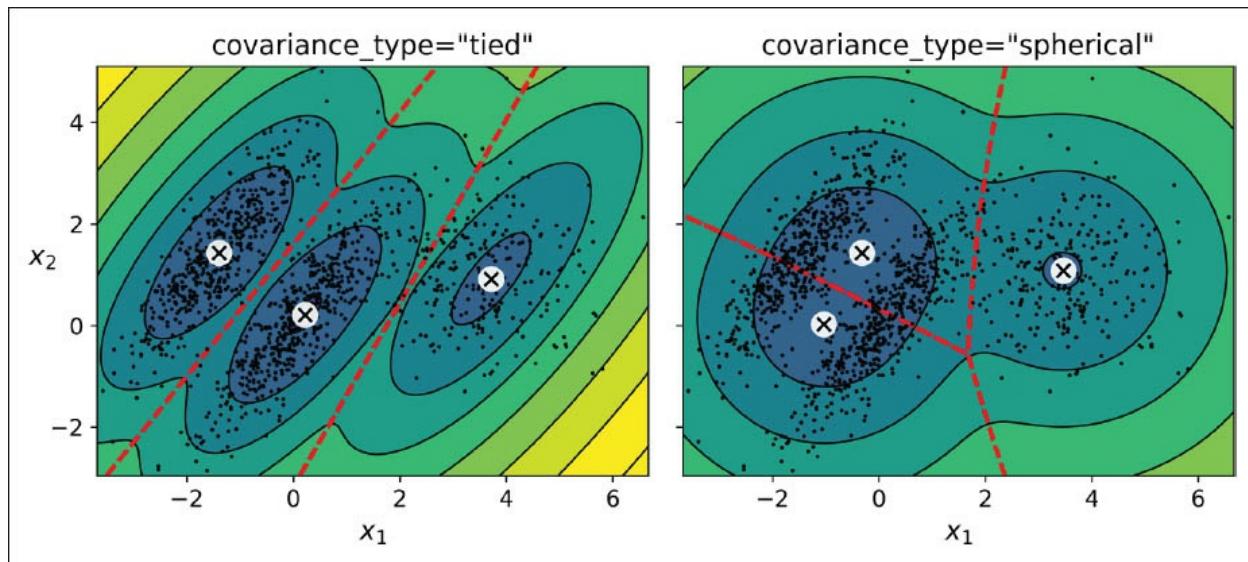


Abbildung 9-18: Gaußsche Mischverteilungsmodelle für »tied« Cluster (links) und »spherical« Cluster (rechts)

Die Rechenkomplexität beim Trainieren eines GaussianMixture-Modells hängt von der Anzahl an Instanzen m , der Dimensionen n , der Cluster k und den Beschränkungen für die Kovarianzmatrizen ab. Hat `covariance_type` den Wert "spherical" oder "diag", ist sie $O(kmn)$, sofern die Daten eine Cluster-Struktur besitzen. Hat `covariance_type` den Wert "tied" oder "full", ist sie $O(kmn^2 + kn^3)$, daher wird das Modell für eine große Zahl an Merkmalen nicht gut skalieren.

Gaußsche Mischverteilungsmodelle können auch zur Anomalieerkennung genutzt werden. Schauen wir uns das an.

Anomalieerkennung mit gaußschen Mischverteilungsmodellen

Bei der *Anomalieerkennung* (auch als *Ausreißererkennung* oder *Outlier Detection* bezeichnet) geht es darum, Instanzen zu erkennen, die stark von der Norm abweichen. Diese Instanzen werden als *Anomalien*, *Ausreißer* oder *Outlier* bezeichnet, während die normalen Instanzen die

Inlier sind. Die Anomalieerkennung ist in vielen Situationen nützlich, zum Beispiel bei der Betrugserkennung, beim Finden defekter Produkte in der Herstellung oder beim Entfernen von Ausreißern aus einem Datensatz, bevor ein anderes Modell trainiert wird (was die Leistung des daraus resultierenden Modells deutlich verbessern kann).

Der Einsatz eines gaußschen Mischverteilungsmodells zur Anomalieerkennung ist ziemlich einfach: Jede Instanz, die sich in einem Bereich mit niedriger Dichte befindet, kann als Anomalie betrachtet werden. Sie müssen definieren, welche Dichtegrenze Sie nutzen wollen. So ist beispielsweise in der Herstellung der Anteil an defekten Produkten gut bekannt. Nehmen wir an, er liegt bei 4%. Sie setzen dann die Dichtegrenze auf den Wert, der dazu führt, dass 4% der Instanzen in Bereichen unterhalb dieser Dichte liegen. Merken Sie, dass Sie zu viele falsch positive Ergebnisse erhalten (also korrekte Produkte, die als defekt markiert sind), können Sie die Grenze absenken. Haben Sie umgekehrt zu viele falsch negative Produkte (also defekte Produkte, die das System nicht als defekt markiert), setzen Sie die Grenze hoch. Das ist die übliche Wechselbeziehung zwischen Relevanz und Sensitivität (siehe [Kapitel 3](#)). So würden Sie die Ausreißer mithilfe der vierten Perzentile der Dichte als Grenze erkennen (also ungefähr 4% der Instanzen werden als Anomalien markiert):

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

[Abbildung 9-19](#) stellt diese Anomalien als Sternchen dar.

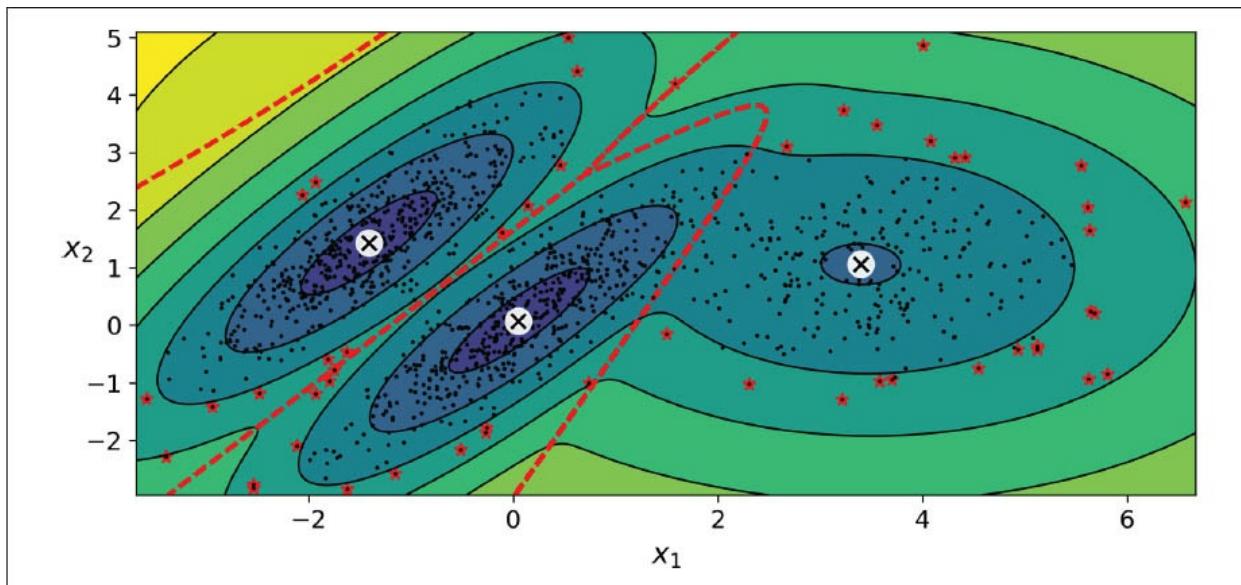


Abbildung 9-19: Anomalieerkennung mithilfe eines gaußschen Mischverteilungsmodells

Eine damit in engem Zusammenhang stehende Aufgabe ist die *Novelty Detection*: Sie unterscheidet sich von der Anomalieerkennung darin, dass angenommen wird, dass der Algorithmus auf einem »sauberen« Datensatz trainiert wurde, der nicht durch Ausreißer

verunreinigt ist, während dies bei der Anomalieerkennung nicht der Fall ist. Tatsächlich wird die Ausreißererkennung oft genutzt, um einen Datensatz zu bereinigen.

Gaußsche Mischverteilungsmodelle versuchen oft, alle Daten zu berücksichtigen – auch die Ausreißer. Haben Sie viele davon, wird das die Sicht des Modells auf die »Normalität« beeinflussen, und manche Ausreißer werden als normal angesehen werden. Wenn das der Fall ist, können Sie versuchen, das Modell einmal anzupassen, dann damit die extremsten Ausreißer zu erkennen und zu entfernen und das Modell erneut an den gesäuberten Datensatz anzupassen. Ein anderer Ansatz ist der Einsatz von robusten Kovarianz-Schätzmethoden (siehe die Klasse *Elliptic Envelope*).

Wie bei K-Means muss für den *GaussianMixture*-Algorithmus die Anzahl an Clustern vorgegeben werden. Wie können Sie die nun ermitteln?

Die Anzahl an Clustern auswählen

Bei K-Means konnten Sie die Trägheit oder den Silhouettenkoeffizienten nutzen, um die passende Anzahl an Clustern auszuwählen. Aber bei der gaußschen Mischverteilung ist es nicht möglich, diese Metriken einzusetzen, weil sie nicht zuverlässig sind, wenn die Cluster keine sphärische Form haben oder unterschiedliche Größen besitzen. Stattdessen können Sie versuchen, das Modell zu finden, das ein *theoretisches Informationskriterium* minimiert, wie zum Beispiel das *bayessche Informationskriterium* (BIC) oder das *Akaike-Informationskriterium* (AIC), die in [Formel 9-1](#) definiert sind.

Formel 9-1: Bayessches Informationskriterium (BIC) und Akaike-Informationskriterium

$$BIC = \log(m)p - 2 \log(\hat{L})$$

$$AIC = 2p - 2 \log(\hat{L})$$

In dieser Gleichung gilt:

- m ist die Anzahl an Instanzen (wie immer).
- p ist die Anzahl an Parametern, die vom Modell erlernt wurden.
- \hat{L} ist der maximierte Wert der *Likelihood-Funktion* des Modells.

Sowohl das BIC wie auch das AIC bestrafen Modelle, die mehr Parameter zu lernen haben (zum Beispiel mehr Cluster), und belohnen Modelle, die sich gut an die Daten anpassen. Häufig landen sie beim gleichen Modell. Gibt es Unterschiede, tendieren die vom BIC gewählten Modelle dazu, einfacher zu sein (weniger Parameter) als die vom AIC gewählten, aber nicht so gut an die Daten angepasst zu sein (das gilt besonders für größere Datensätze).

Likelihood-Funktion

Die Begriffe »Probability« und »Likelihood« werden in der englischen Sprache oft für das Gleiche verwendet, aber in der Statistik haben sie sehr unterschiedliche Bedeutungen. Bei einem statistischen Modell mit Parametern θ wird das Wort »Probability« genutzt, um zu beschreiben, wie plausibel ein zukünftiges Ergebnis x ist (bei bekannten Parameterwerten

θ), während das Wort »Likelihood« verwendet wird, um zu beschreiben, wie plausibel ein bestimmter Satz an Parameterwerten θ ist, nachdem das Ergebnis x bekannt ist.

Stellen Sie sich ein 1-D-Mischmodell zweier gaußscher Verteilungen vor, die bei -4 und $+1$ zentriert sind. Aus Gründen der Einfachheit hat dieses Spielmodell einen einzelnen Parameter θ , der die Standardabweichung beider Verteilungen steuert. Der obere linke Konturplot in Abbildung 9-20 zeigt das gesamte Modell $f(x; \theta)$ als Funktion von x und θ . Um die Wahrscheinlichkeitsverteilung eines zukünftigen Ergebnisses x abzuschätzen, müssen Sie den Modellparameter θ setzen. Setzen Sie beispielsweise θ auf $1,3$ (die horizontale Linie), erhalten Sie die Wahrscheinlichkeitsdichtefunktion $f(x; \theta=1,3)$ aus dem unteren linken Plot. Nehmen wir an, Sie wollten die Wahrscheinlichkeit abschätzen, dass x zwischen -2 und $+2$ liegt. Sie müssen das Integral der WDF für diesen Bereich berechnen (also die Oberfläche des schattierten Gebiets). Aber was, wenn Sie θ nicht kennen und stattdessen eine einzelne Instanz $x=2,5$ beobachtet haben (die vertikale Linie im oberen linken Plot)? In diesem Fall erhalten Sie die Likelihood-Funktion $\mathcal{L}(\theta|x=2,5)=f(x=2,5; \theta)$, zu sehen im oberen rechten Plot.

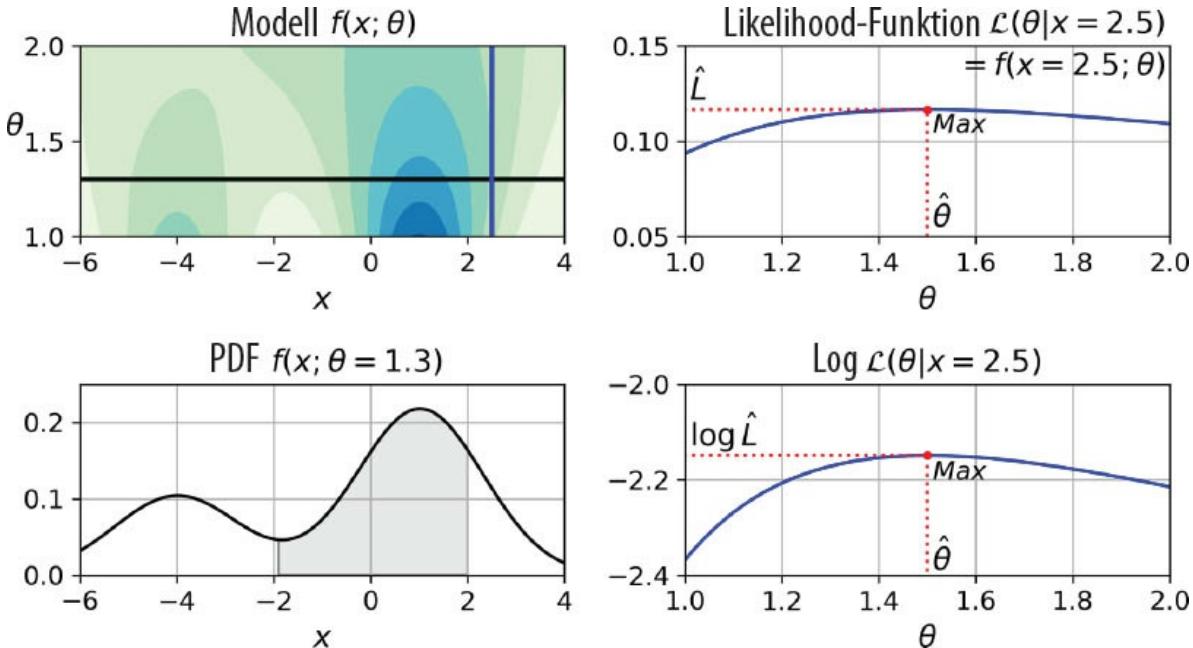


Abbildung 9-20: Eine parametrische Funktion des Modells (oben links) und ein paar abgeleitete Funktionen: eine WDF (unten links), eine Likelihood-Funktion (oben rechts) und eine Log-Likelihood-Funktion (unten rechts)

Kurz gesagt, ist die WDF eine Funktion von x (mit festem θ), während es sich bei der Likelihood-Funktion um eine Funktion von θ (mit festem x) handelt. Es ist wichtig, zu verstehen, dass die Likelihood-Funktion *keine* Wahrscheinlichkeitsverteilung ist: Integrieren Sie eine Wahrscheinlichkeitsverteilung über alle möglichen Werte von x , erhalten Sie immer 1; integrieren Sie die Likelihood-Funktion über alle möglichen Werte von θ , kann das Ergebnis ein beliebiger positiver Wert sein.

Bei einem vorhandenen Datensatz X will man häufig die wahrscheinlichsten Werte für die

Modellparameter schätzen. Dazu müssen Sie die Werte finden, die die Likelihood-Funktion maximieren – bei gegebenem X . Wenn Sie in diesem Beispiel eine einzelne Instanz $x=2,5$ beobachtet haben, liefert die *Maximum-Likelihood-Schätzung* (Maximum Likelihood Estimation, MLE) von θ einen Wert von $\hat{\theta}=1,5$. Gibt es eine frühere Wahrscheinlichkeitsverteilung g über θ , ist es möglich, diese zu berücksichtigen, indem $\mathcal{L}(\theta|x)g(\theta)$ maximiert wird statt nur $\mathcal{L}(\theta|x)$. Das wird als *Maximum-a-posteriori-Schätzung* (MAP) bezeichnet. Da die MAP die Parameterwerte beschränkt, können Sie sie sich als eine regulierte Version der MLE vorstellen.

Beachten Sie, dass das Maximieren der Likelihood-Funktion dem Maximieren ihres Logarithmus entspricht (zu sehen im unteren rechten Plot von [Abbildung 9-20](#)). Der Logarithmus ist eine streng monoton wachsende Funktion – maximiert θ die Log-Likelihood, maximiert es auch die Likelihood. Es zeigt sich, dass es meist einfacher ist, die Log-Likelihood zu maximieren. Beobachten Sie beispielsweise viele unabhängige Instanzen $x^{(1)}$ bis $x^{(m)}$, müssten Sie den Wert von θ finden, der das Produkt der individuellen Likelihood-Funktionen maximiert. Aber es ist äquivalent und viel einfacher, die Summe (nicht das Produkt) der Log-Likelihood-Funktionen zu maximieren – dank der Magie des Logarithmus, der Produkte in Summen verwandelt: $\log(ab) = \log(a)+\log(b)$.

Haben Sie mit $\hat{\theta}$ den Wert von θ geschätzt, der die Likelihood-Funktion maximiert, können Sie $\hat{L} = \mathcal{L}(\hat{\theta}, X)$ berechnen, was der Wert ist, der zum Berechnen von AIC und BIC genutzt wird – Sie können ihn sich als Messwert dafür vorstellen, wie gut das Modell auf die Daten passt.

Um BIC und AIC zu berechnen, rufen Sie die Methoden `bic()` und `aic()` auf:

```
>>> gm.bic(X)
8189.74345832983
>>> gm.aic(X)
8102.518178214792
```

In [Abbildung 9-21](#) sehen Sie das BIC für verschiedene Mengen k an Clustern. Sowohl das BIC wie auch das AIC sind bei $k=3$ am kleinsten, daher ist das sehr wahrscheinlich die beste Wahl. Beachten Sie, dass wir auch nach dem besten Wert für die Hyperparameter `covariance_type` suchen könnten. Ist er beispielsweise "spherical" statt "full", muss das Modell signifikant weniger Parameter lernen, passt aber auch nicht so gut auf die Daten.

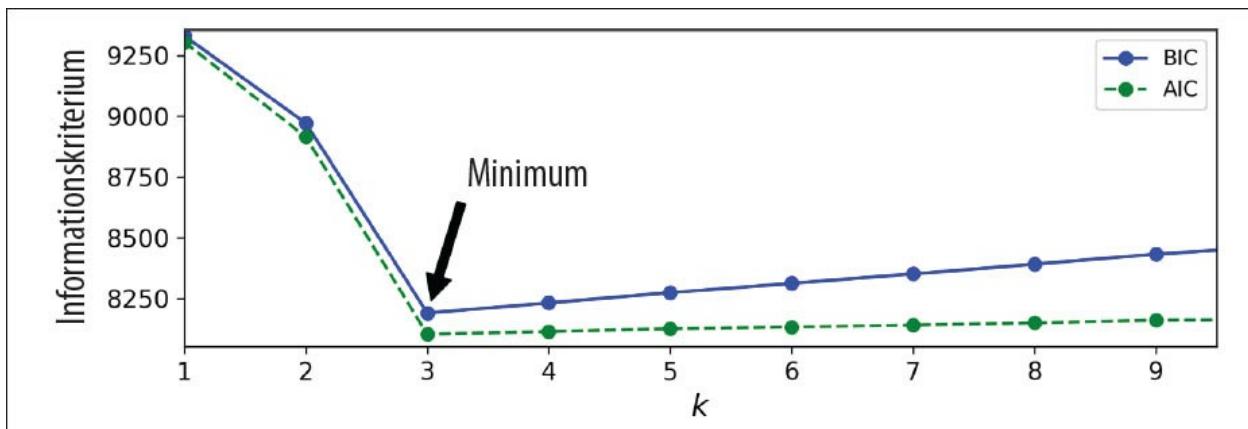


Abbildung 9-21: AIC und BIC für verschiedene Mengen k an Clustern

Bayessche gaußsche Mischverteilungsmodelle

Statt manuell nach der optimalen Zahl von Clustern zu suchen, können Sie die Klasse BayesianGaussianMixture verwenden, die Gewichte gleich (oder nahe an) null für unnötige Cluster vergeben kann. Setzen Sie die Zahl n_components an Clustern auf einen Wert, für den Sie gute Gründe dafür haben, dass er größer als die optimale Zahl an Clustern ist (dazu ist zumindest ein wenig Wissen über das aktuelle Problem notwendig), wird der Algorithmus die unnötigen Cluster automatisch eliminieren. Setzen wir beispielsweise die Anzahl an Clustern auf 10 und schauen wir, was passiert:

```
>>> from sklearn.mixture import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10)
>>> bgm.fit(X)
>>> np.round(bgm.weights_, 2)
array([0.4 , 0.21, 0.4 , 0. , 0. , 0. , 0. , 0. , 0. , 0. ])
```

Perfekt: Der Algorithmus hat automatisch erkannt, dass nur drei Cluster erforderlich sind, und die so entstehenden Cluster sind nahezu identisch zu denen aus [Abbildung 9-17](#).

In diesem Modell werden die Cluster-Parameter (einschließlich der Gewichte, Mittelwerte und Kovarianzmatrizen) nicht mehr als feste Modellparameter behandelt, sondern als latente Zufallsvariablen, so wie die Cluster-Zuweisungen (siehe [Abbildung 9-22](#)). Daher enthält \mathbf{z} nun sowohl die Cluster-Parameter wie auch die Cluster-Zuweisungen.

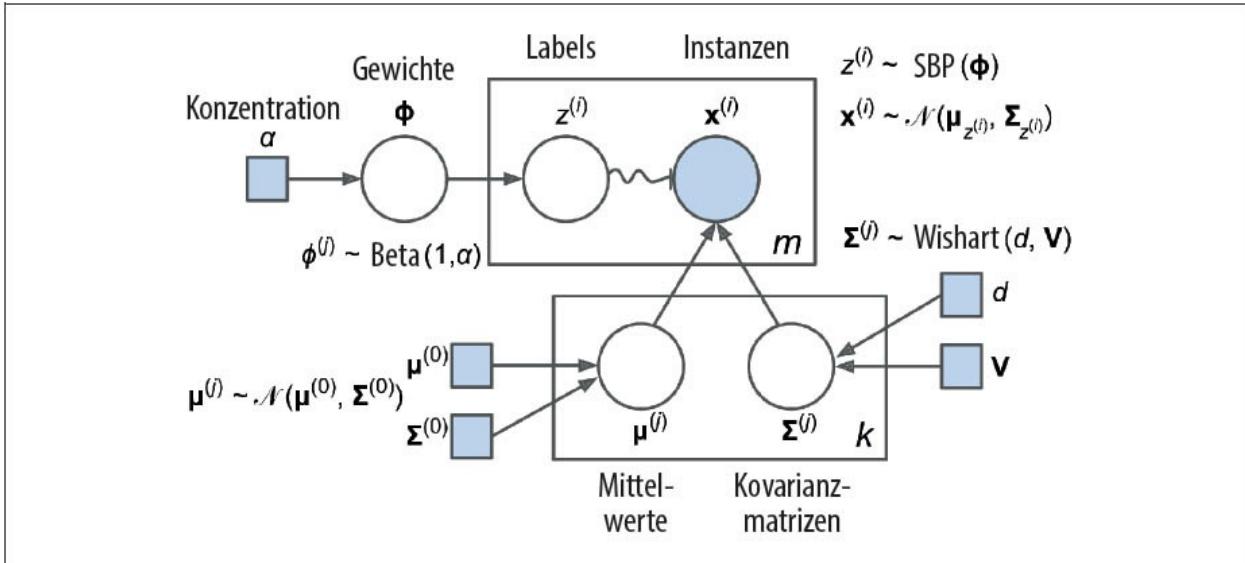


Abbildung 9-22: Bayessches gaußsches Mischverteilungsmodell

Die Beta-Verteilung wird meist genutzt, um Zufallsvariablen zu modellieren, deren Werte innerhalb eines festen Bereichs liegen. In diesem Fall ist dies der Bereich von 0 bis 1. Der Stick-Breaking-Prozess (SBP) lässt sich am besten durch ein Beispiel erläutern: Nehmen wir $\Phi=[0,3, 0,6, 0,5, \dots]$ an, werden 30% der Instanzen Cluster 0 zugewiesen, 60% der verbleibenden Instanzen gehen an Cluster 1, dann 50% der verbleibenden Instanzen an Cluster 2 und so weiter. Der Prozess ist ein gutes Modell für Datensätze, bei denen neue Instanzen eher größeren als kleineren Clustern zugewiesen werden (zum Beispiel Menschen, die eher in große als in kleine Städte ziehen). Ist die Konzentration α hoch, werden Werte von Φ wahrscheinlich nahe 0 liegen, und der SBP erzeugt viele Cluster. Ist die Konzentration umgekehrt eher niedrig, werden die Werte für Φ eher nahe bei 1 liegen, und es wird wenige Cluster geben. Schließlich wird die Wishart-Verteilung genutzt, um Kovarianzmatrizen zu erzeugen: Die Parameter d und \mathbf{V} steuern die Verteilung der Cluster-Formen.

Vorhandenes Wissen über die latenten Variablen \mathbf{z} kann in einer Wahrscheinlichkeitsverteilung $p(\mathbf{z})$ codiert werden, die als *A-priori-Verteilung* bezeichnet wird. So könnten wir beispielsweise schon vorab der Meinung sein, dass es nur wenige Cluster gibt (geringe Konzentration) oder umgekehrt sehr viele Cluster vorhanden sein werden (hohe Konzentration). Diese A-priori-Annahmen über die Anzahl an Clustern kann mithilfe des Hyperparameters `weight_concentration_prior` angepasst werden. Es führt zu sehr unterschiedlichem Clustering, wenn man ihn auf 0,01 oder 10.000 setzt (siehe Abbildung 9-23). Je mehr Daten wir haben, desto weniger sind aber die A-priori-Annahmen entscheidend. Und tatsächlich müssen Sie sehr starke A-priori-Annahmen und sehr wenige Daten haben, um Diagramme mit solch großen Unterschieden zu erhalten.

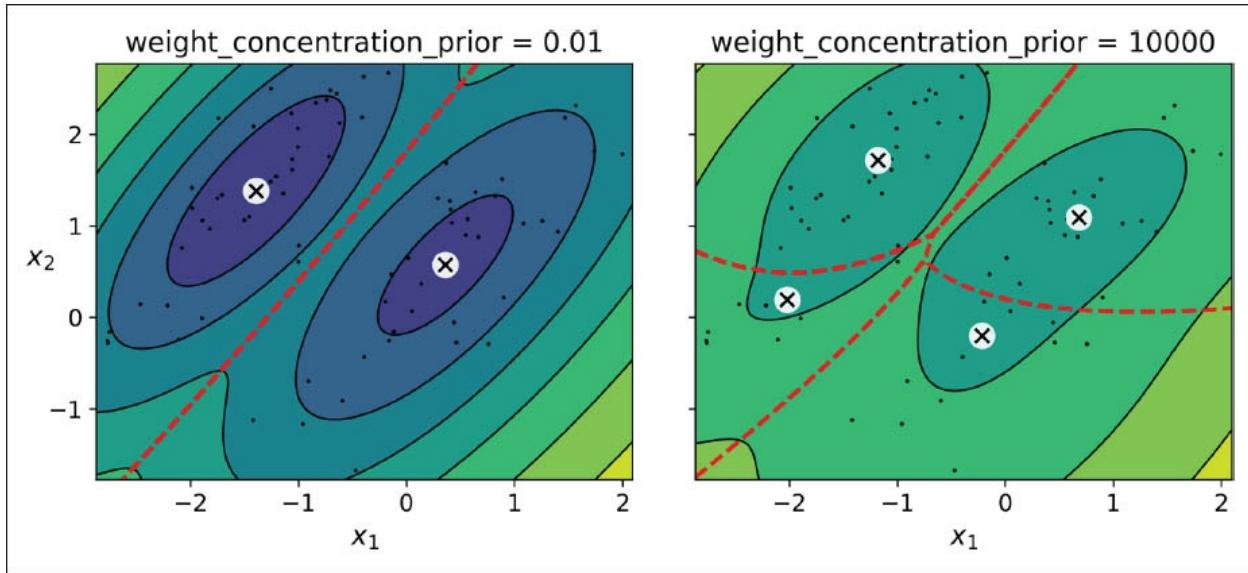


Abbildung 9-23: Verschiedene A-priori-Annahmen für die gleichen Daten führen zu unterschiedlich vielen Clustern.

Der Satz von Bayes ([Formel 9-2](#)) sagt uns, wie wir die Wahrscheinlichkeitsverteilung über die latenten Variablen aktualisieren können, nachdem wir bestimmte Daten \mathbf{X} beobachtet haben. Er berechnet die A-posteriori-Verteilung $p(\mathbf{z}|\mathbf{X})$, bei der es sich um die bedingte Wahrscheinlichkeit von \mathbf{z} bei gegebenem \mathbf{X} handelt.

Formel 9-2: Satz von Bayes

$$p(\mathbf{z} \mid \mathbf{X}) = \text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}} = \frac{p(\mathbf{X} \mid \mathbf{z})p(\mathbf{z})}{p(\mathbf{X})}$$

Leider ist in einem gaußschen Mischverteilungsmodell (und bei vielen anderen Problemen) der Nenner $p(\mathbf{X})$ unlösbar, da dafür über alle möglichen Werte von \mathbf{z} integriert werden muss ([Formel 9-3](#)), was ein Berücksichtigen aller möglichen Kombinationen aus Cluster-Parametern und Cluster-Zuweisungen erfordert würde.

Formel 9-3: Der Beweis, dass $p(\mathbf{X})$ oft unlösbar ist

$$p(\mathbf{X}) = \int p(\mathbf{X} \mid \mathbf{z})p(\mathbf{z})d\mathbf{z}$$

Diese Unlösbarkeit ist eines der zentralen Probleme in der bayesschen Statistik, und es gibt diverse Versuche, sie zu lösen. Einer davon ist die *Variational Inference*, die sich eine Familie von Verteilungen $q(\mathbf{z}; \boldsymbol{\lambda})$ mit ihren eigenen *Variational Parameters* $\boldsymbol{\lambda}$ (Lambda) nimmt und diese Parameter dann optimiert, um $q(\mathbf{z})$ zu einer guten Näherung von $p(\mathbf{z}|\mathbf{X})$ zu machen. Dies wird erreicht, indem der Wert von $\boldsymbol{\lambda}$ gefunden wird, der die KL-Divergenz von $q(\mathbf{z})$ nach $p(\mathbf{z}|\mathbf{X})$ minimiert, was als $D_{KL}(q \parallel p)$ bezeichnet wird. Die KL-Divergenz ist in [Formel 9-4](#) definiert und lässt sich umschreiben als Logarithmus der Evidenz ($\log p(\mathbf{X})$) minus der *Evidence Lower Bound*(ELBO). Da der Logarithmus der Evidenz nicht von q abhängt, handelt es sich um einen konstanten Term, daher muss zum Minimieren der KL-Divergenz nur die ELBO maximiert werden.

Formel 9-4: KL-Divergenz von $q(\mathbf{z})$ nach $p(\mathbf{z}|\mathbf{X})$

$$\begin{aligned}
 D_{KL}(q \parallel p) &= \mathbb{E}_q \left[\log \frac{q(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{X})} \right] \\
 &= \mathbb{E}_q [\log q(\mathbf{z}) - \log p(\mathbf{z} \mid \mathbf{X})] \\
 &= \mathbb{E}_q \left[\log q(\mathbf{z}) - \log \frac{p(\mathbf{z}, \mathbf{X})}{p(\mathbf{X})} \right] \\
 &= \mathbb{E}_q [\log q(\mathbf{z}) - \log p(\mathbf{z}, \mathbf{X}) + \log p(\mathbf{X})] \\
 &= \mathbb{E}_q [\log q(\mathbf{z})] - \mathbb{E}_q [\log p(\mathbf{z}, \mathbf{X})] + \mathbb{E}_q [\log p(\mathbf{X})] \\
 &= \mathbb{E}_q [\log p(\mathbf{X})] - (\mathbb{E}_q [\log p(\mathbf{z}, \mathbf{X})] - \mathbb{E}_q [\log q(\mathbf{z})]) \\
 &= \log p(\mathbf{X}) - \text{ELBO}
 \end{aligned}$$

wobei $\text{ELBO} = \mathbb{E}_q [\log p(\mathbf{z}, \mathbf{X})] - \mathbb{E}_q [\log q(\mathbf{z})]$

In der Praxis gibt es verschiedene Techniken, um die ELBO zu maximieren. Bei der *Mean Field Variational Inference* ist es notwendig, die Familie der Verteilungen $q(\mathbf{z}; \lambda)$ und den Prior $p(\mathbf{z})$ sehr sorgfältig auszuwählen, um sicherzustellen, dass die Gleichung für die ELBO so weit vereinfacht wird, dass sie berechnet werden kann. Leider gibt es hier keine allgemein anwendbare Vorgehensweise. Die Wahl der richtigen Familie von Verteilungen und des richtigen Priors hängt von der Aufgabe ab und erfordert einiges an mathematischen Fertigkeiten. So sind beispielsweise die Verteilungen und die Gleichungen für die ELBO für die Klasse *BayesianGaussian Mixture* von Scikit-Learn in der Dokumentation (<https://homl.info/40>) beschrieben. Von diesen Gleichungen ausgehend, ist es möglich, aktualisierte Gleichungen für die Cluster-Parameter und Zuweisungsvariablen abzuleiten – sie werden dann sehr ähnlich wie im Expectation-Maximization-Algorithmus verwendet. Tatsächlich entspricht die Rechenkomplexität der Klasse *BayesianGaussianMixture* der von *GaussianMixture* (sie ist aber im Allgemeinen langsamer). Ein einfacherer Ansatz zum Maximieren der ELBO nennt sich *Black Box Stochastic Variational Inference* (BBSVI): Bei jeder Iteration werden ein paar Proben aus q gezogen, und diese werden dann genutzt, um die Gradienten der ELBO in Bezug auf die Variational Parameters λ zu schätzen, die dann in einem Schritt des Gradientenverfahren zum Einsatz kommen. Dieses Vorgehen ermöglicht den Einsatz der bayesschen Statistik mit jeder Art von Modell (sofern es differenzierbar ist), selbst für neuronale Netze – die Anwendung der bayesschen Statistik auf Deep Neural Networks wird als Bayesian Deep Learning bezeichnet.



Wollen Sie tiefer in die bayessche Statistik einsteigen, werfen Sie einen Blick auf das Buch *Bayesian Data Analysis* (<https://homl.info/bda>) von Andrew Gelman et al. (Chapman & Hill).

Gaußsche Mischverteilungsmodelle funktionieren sehr gut bei Clustern mit ellipsoiden Formen,

aber wenn Sie versuchen, einen Datensatz mit einer anderen Form anzupassen, werden Sie eventuell eine böse Überraschung erleben. Schauen wir uns beispielsweise an, was passiert, wenn wir ein bayessches gaußsches Mischverteilungsmodell nutzen, um den *moons*-Datensatz zu clustern (siehe Abbildung 9-24).

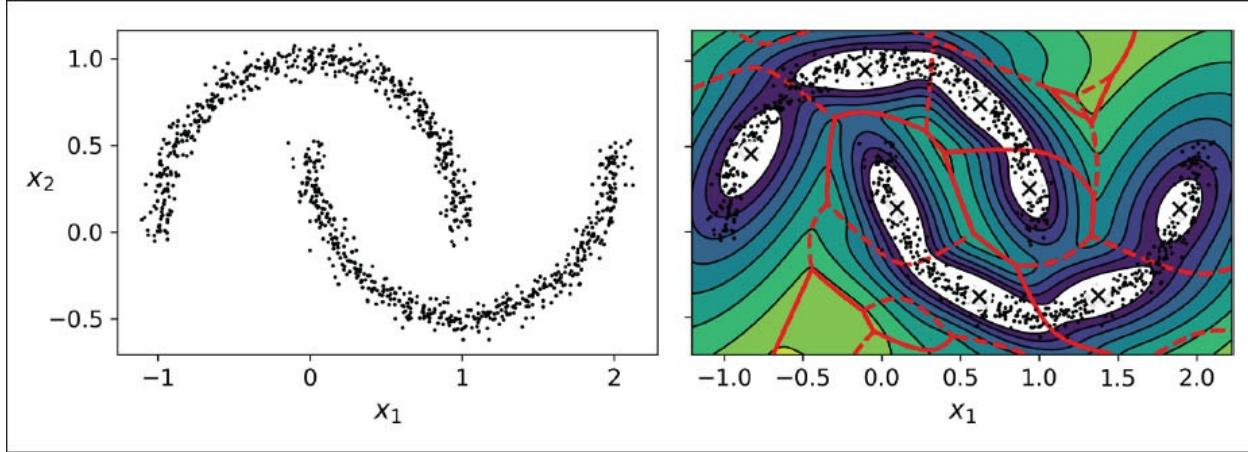


Abbildung 9-24: Eine gaußsche Mischverteilung auf nicht ellipsoide Cluster anwenden

Hoppla! Der Algorithmus hat verzweifelt nach Ellipsoiden gesucht, also hat er statt zweien ganze acht verschiedene Cluster gefunden. Die Dichteabschätzung ist gar nicht so schlecht, daher könnte dieses Modell zur Anomalieerkennung verwendet werden, aber es hat es nicht geschafft, die beiden Monde zu erkennen. Schauen wir uns noch ein paar Clustering-Algorithmen an, die dazu in der Lage sind, mit beliebig geformten Clustern umzugehen.

Andere Algorithmen zur Anomalie- und Novelty-Erkennung

Scikit-Learn implementiert noch weitere Algorithmen zur Anomalie- oder Novelty-Erkennung:

`PCA (und andere Techniken zur Dimensionsreduktion mit einer Methode inverse_transform())`

Vergleichen Sie den Rekonstruktionsfehler einer normalen Instanz mit dem einer Anomalie, wird Letzterer meist viel größer sein. Das ist ein einfaches und oft ziemlich effizientes Vorgehen bei der Anomalieerkennung (in den Übungen zu diesem Kapitel finden Sie eine Anwendung dieses Verfahrens).

Fast-MCD (Minimum Covariance Determinant)

Dieser Algorithmus ist in der Klasse `EllipticEnvelope` implementiert. Er lässt sich zur Ausreißererkennung einsetzen – insbesondere zum Säubern von Datensätzen. Der Algorithmus geht davon aus, dass die normalen Instanzen (Inliers) aus einer einzelnen gaußschen Verteilung erzeugt werden (keiner Mischverteilung). Zudem nimmt er an, dass der Datensatz mit Ausreißern kontaminiert ist, die nicht aus dieser gaußschen Verteilung entstammen. Schätzt der Algorithmus die Parameter der gaußschen Verteilung (also die Form der elliptischen Hüllkurve um die Inliers), achtet er darauf, die Instanzen zu ignorieren, die sehr wahrscheinlich Ausreißer sind. Diese Technik liefert eine bessere Schätzung der elliptischen Hüllkurve und macht den Algorithmus dadurch besser beim

Erkennen der Ausreißer.

Isolation Forest

Hierbei handelt es sich um einen effizienten Algorithmus zur Ausreißererkennung, insbesondere in hochdimensionalen Datensätzen. Er baut einen Random Forest auf, in dem jeder Entscheidungsbaum zufällig aufgezogen wird: Bei jedem Knoten wählt er ein Merkmal zufällig und dann einen zufälligen Grenzwert (zwischen dem minimalen und maximalen Wert), um den Datensatz aufzuteilen. Der Datensatz wird dadurch nach und nach in Bereiche unterteilt, bis alle Instanzen von den anderen Instanzen isoliert sind. Anomalien befinden sich meist weit weg von anderen Instanzen, daher tendieren sie im Durchschnitt (über alle Entscheidungsbäume hinweg) dazu, in weniger Schritten isoliert zu werden als normale Instanzen.

Local Outlier Factor (LOF)

Dieser Algorithmus ist ebenfalls hilfreich zur Ausreißererkennung. Er vergleicht die Dichte von Instanzen um eine gegebene Instanz mit der Dichte um ihre Nachbarn. Eine Anomalie ist meist isolierter als ihre k nächsten Nachbarn.

One-Class-SVM

Dieser Algorithmus passt besser für die Novelty Detection. Sie erinnern sich, dass ein Kernel-SVM-Klassifikator zwei Klassen aufteilt, indem er zuerst (implizit) alle Instanzen in einem höher dimensionalen Raum abbildet und die beiden Klassen dann in diesem Raum mithilfe eines linearen SVM-Klassifikators unterteilt (siehe [Kapitel 5](#)). Da wir nur eine Klasse an Instanzen haben, versucht der One-Class-SVM-Algorithmus stattdessen, die Instanzen in einem höher dimensionalen Raum zu trennen. Im ursprünglichen Raum entspricht dies dem Finden eines kleinen Bereichs, der alle Instanzen umschließt. Fällt eine neue Instanz nicht in diesen Bereich, handelt es sich um eine Anomalie. Es gibt ein paar Hyperparameter, an denen man drehen kann: die normalen für eine Kernel-SVM, dazu einen Margin-Hyperparameter, der der Wahrscheinlichkeit einer neuen Instanz entspricht, fälschlicherweise als Novelty betrachtet zu werden, wo sie doch tatsächlich normal wäre. Der Algorithmus funktioniert toll, insbesondere bei hochdimensionalen Datensätzen, aber wie alle SVMs skaliert er nicht gut bei großen Datensätzen.

Übungen

1. Wie würden Sie Clustering definieren? Können Sie ein paar Clustering-Algorithmen benennen?
2. Zählen Sie ein paar wichtige Anwendungsgebiete für Clustering-Algorithmen auf.
3. Beschreiben Sie zwei Techniken, um beim Einsatz von K-Means die richtige Zahl von Clustern auszuwählen.
4. Was ist Label Propagation? Warum und wie würden Sie es implementieren?
5. Können Sie zwei Clustering-Algorithmen aufzählen, die gut für große Datensätze skalieren? Und zwei, die nach Bereichen mit hoher Dichte suchen?
6. Können Sie sich einen Anwendungsfall vorstellen, bei dem aktives Lernen nützlich wäre?

Wie würden Sie es implementieren?

7. Was ist der Unterschied zwischen Anomalieerkennung und Novelty Detection?
8. Was ist eine gaußsche Mischverteilung? Für welche Aufgaben können Sie sie verwenden?
9. Können Sie zwei Techniken angeben, um beim Einsatz von gaußschen Mischverteilungsmodellen die richtige Zahl von Clustern zu finden?
10. Das klassische Olivetti Faces Dataset enthält 400 Graustufenbilder der Größe 64×64 mit Gesichtern. Jedes Bild ist zu einem 1-D-Vektor der Größe 4096 flachgeklopft. Es wurden 40 verschiedene Personen fotografiert (jeweils 10 Mal), und meist besteht die Aufgabe darin, ein Modell zu trainieren, das vorhersagt, welche Person in jedem Bild dargestellt wird. Laden Sie den Datensatz mit der Funktion `sklearn.datasets.fetch_olivetti_faces()`, dann teilen Sie ihn in einen Trainingdatensatz, einen Validierungsdatensatz und einen Testdatensatz auf (beachten Sie, dass der Datensatz schon zwischen 0 und 1 skaliert ist). Da der Datensatz ziemlich klein ist, wollen Sie vermutlich eine stratifizierte Stichprobe erstellen, um sicherzustellen, dass es in jedem Datensatz die gleiche Menge an Bildern pro Person gibt. Als Nächstes clustern Sie die Bilder mithilfe von K-Means und stellen sicher, dass Sie eine gute Zahl von Clustern haben (mit einer der Techniken aus diesem Kapitel). Visualisieren Sie die Cluster: Sehen Sie ähnliche Bilder in jedem Cluster?
11. Trainieren Sie einen Klassifikator mit dem Olivetti-Datensatz, um vorherzusagen, welche Person in jedem Bild zu sehen ist, und prüfen Sie das mit dem Validierungsdatensatz. Als Nächstes verwenden Sie K-Means zur Dimensionsreduktion und trainieren einen Klassifikator mit dem reduzierten Datensatz. Suchen Sie nach der Anzahl an Clustern, die es dem Klassifikator erlauben, seine beste Leistung zu bringen: Welche Qualität können Sie erreichen? Was passiert, wenn Sie die Merkmale aus dem reduzierten Datensatz an die ursprünglichen Merkmale anfügen? (Suchen Sie auch hier wieder nach der besten Menge an Clustern.)
12. Trainieren Sie ein gaußsches Mischverteilungsmodell mit dem Olivetti-Datensatz. Um den Algorithmus zu beschleunigen, sollten Sie vermutlich die Dimensionalität des Datensatzes verringern (zum Beispiel mit PCA unter Beibehaltung von 99% der Varianz). Nutzen Sie das Modell, um neue Gesichter zu erzeugen (mithilfe der Methode `sample()`), und visualisieren Sie diese (wenn Sie PCA eingesetzt haben, werden Sie dessen Methode `inverse_transform()` verwenden müssen). Versuchen Sie, ein paar Bilder zu verändern (zum Beispiel rotieren, spiegeln, dunkler machen), und schauen Sie, ob das Modell die Anomalien erkennen kann (vergleichen Sie also die Ausgabe der Methode `score_samples()` für normale Bilder und für Anomalien).
13. Einige Techniken zur Dimensionsreduktion können auch zur Anomalieerkennung verwendet werden. Nehmen Sie beispielsweise den Olivetti-Datensatz und reduzieren Sie ihn mithilfe von PCA unter Beibehaltung von 99% der Varianz. Dann berechnen Sie den Rekonstruktionsfehler für jedes Bild. Als Nächstes nehmen Sie ein paar der modifizierten Bilder aus der vorherigen Übung und schauen sich deren Rekonstruktionsfehler an: Beachten Sie, wie viel größer er ist. Plotten Sie ein wiederhergestelltes Bild, werden Sie auch sehen, warum: Es wird versucht, ein normales Bild zu rekonstruieren.

Lösungen zu diesen Übungen finden Sie in [Anhang A](#).

TEIL II

Neuronale Netze und Deep Learning

Einführung in künstliche neuronale Netze mit Keras

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): Ein ANN ist ein Machine-Learning-Modell, das von den Netzwerken der Neuronen in unseren Gehirnen inspiriert ist. 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 zu schlagen (AlphaGo von DeepMind).

Im ersten Teil dieses Kapitels werden Sie künstliche neuronale Netze kennenlernen. Wir beginnen mit einem kurzen Überblick über die ersten Architekturen von ANNs, um dann zu *mehrschichtige Perzeptrons* (MLPs) zu gelangen, die heutzutage sehr oft zum Einsatz kommen (andere Architekturen werden in den nächsten Kapiteln vorgestellt). Im zweiten Teil werden wir uns anschauen, wie wir neuronale Netze mithilfe der beliebten Keras-API implementieren. Dabei handelt es sich um eine sehr schöne und einfache High-Level-API zum Bauen, Trainieren, Auswerten und Ausführen neuronaler Netze. Aber lassen Sie sich nicht von ihrer Einfachheit täuschen: Sie ist ausdrucksstark und flexibel genug, um damit eine Vielzahl von Architekturen neuronaler Netze zu bauen. Tatsächlich wird sie für die meisten Ihrer Anwendungsfälle vermutlich ausreichen. Und sollten Sie jemals mehr Flexibilität benötigen, können Sie immer auch eigene Keras-Komponenten mithilfe ihrer tiefer gehenden API schreiben, wie Sie in [Kapitel 12](#) sehen werden.

Aber zuerst schauen wir zurück in die Vergangenheit, um zu erfahren, wie sich künstliche neuronale Netze entwickelt haben!

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://homl.info/43>)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 in die 1960er-Jahre 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 wurden neue Architekturen und bessere Trainingstechniken entwickelt, was zu einem wiedererweckten Interesse am Konnektionismus (der Forschung an neuronalen Netzen) führte. Aber es ging nur langsam voran, und erst in den 1990ern wurden andere mächtige Machine-Learning-Verfahren wie Support Vector Machines (siehe [Kapitel 5](#)) entwickelt. Diese versprachen bessere Ergebnisse und ein solideres theoretisches Fundament als ANNs, daher kam die Forschung an neuronalen Netzen erneut zum Stillstand.

Wir erleben heute ein erneutes gesteigertes Interesse an ANNs. Wird dies wieder wie die früheren Wellen abflauen? Nun, es gibt ein paar gute Gründe, warum es dieses Mal anders sein könnte und dass das erneute Interesse an ANNs grundlegendere Auswirkungen auf unser Leben haben 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 (die Anzahl an Komponenten in integrierten Schaltkreisen hat sich in den letzten 50 Jahren etwa alle zwei Jahre verdoppelt), teils an der Spieleindustrie, der wir leistungsfähige Grafikprozessoren verdanken. Dazu haben Cloud-Plattformen diese Kapazität für alle verfügbar gemacht.
- 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 schaffen es regelmäßig in die Schlagzeilen, was weitere Aufmerksamkeit auf sie lenkt und Förderung bewirkt, und das zieht wiederum weitere Fortschritte und noch faszinierende Produkte nach sich.

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 Gehirn von Tieren antrifft. 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.³ Biologische Neuronen erzeugen kurze elektrische Impulse oder *Aktionspotenziale* (APs oder einfach *Signale*), die die Axone entlangwandern und dafür sorgen, dass die Synapsen chemische Signale – sogenannte *Neurotransmitter* – emittieren. Erhält ein Neuron innerhalb weniger Millisekunden eine ausreichende Anzahl dieser Neurotransmitter, feuert es sein eigenes elektrisches Signal ab (tatsächlich hängt das von den Neurotransmittern ab, da manche von ihnen das Neuron daran hindern, zu feuern).

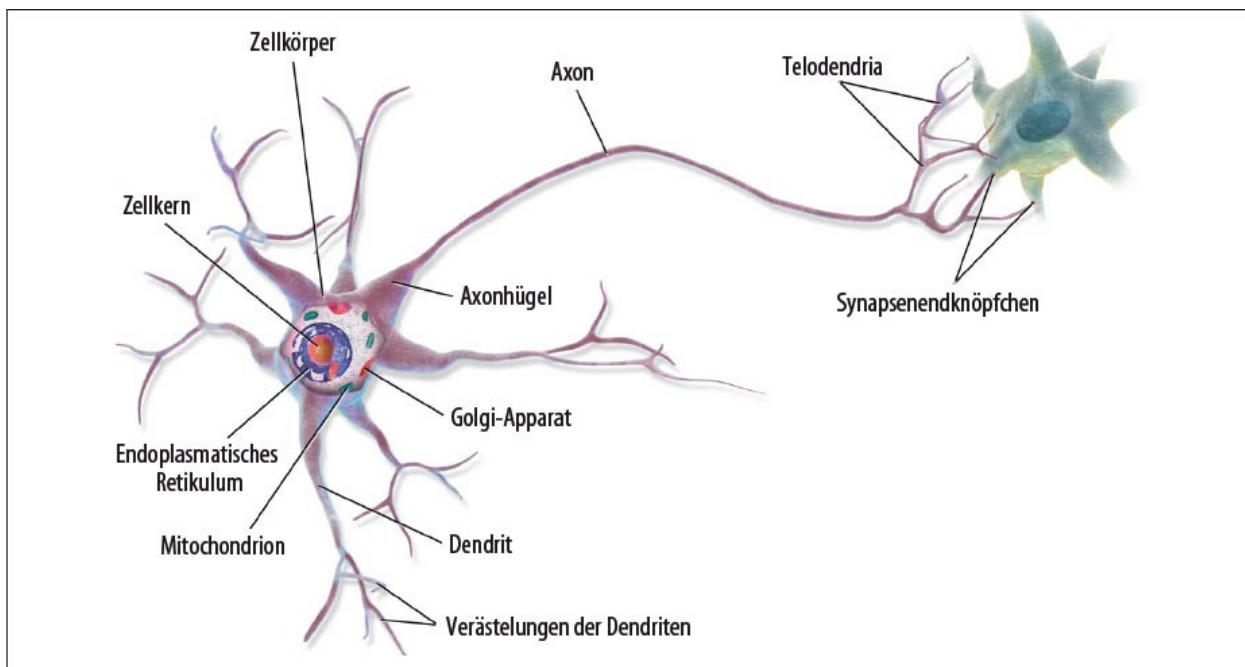


Abbildung 10-1: Biologisches Neuron⁴

Biologische Neuronen verhalten sich also auf scheinbar einfache Weise. Sie sind aber als riesiges Netzwerk mit Milliarden Neuronen organisiert, in dem jedes Neuron mit Tausenden anderer 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)⁵ 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, insbesondere im zerebralen Kortex (also der äußeren Schicht Ihres Gehirns), wie [Abbildung 10-2](#) zeigt.

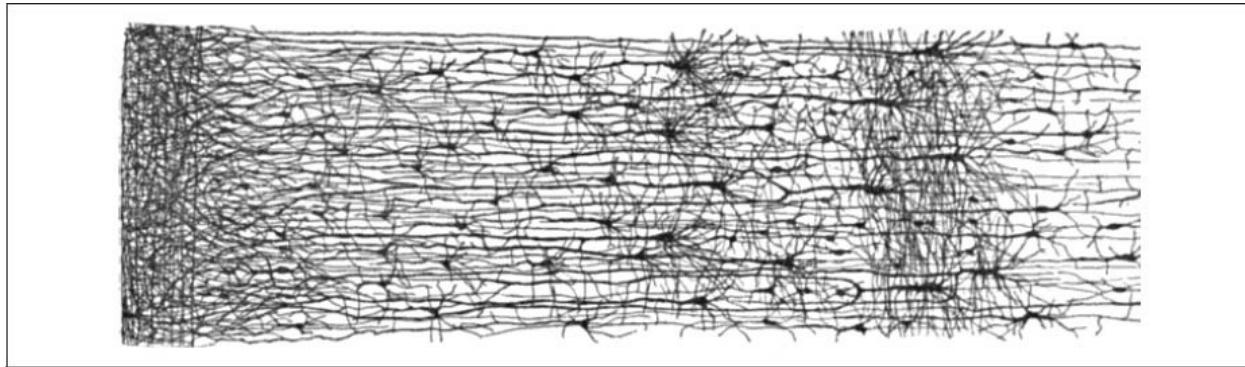


Abbildung 10-2: Mehrere Schichten eines biologischen neuronalen Netzes (menschlicher Kortex)⁶

Logische Berechnungen mit Neuronen

McCulloch und 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 dann aktiviert, wenn eine bestimmte Anzahl Eingaben aktiv sind. In ihrem Artikel haben sie nachgewiesen, dass sich selbst mit einem derart einfachen Modell ein neuronales Netz konstruieren lässt, das jeden möglichen logischen Ausdruck berechnet. Um zu sehen, wie solch ein Netz funktioniert, 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.

Schauen wir uns mal an, was diese Netze tun:

- 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).
- 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 (ein Beispiel dazu finden Sie in den Übungen am Ende des

Kapitels).

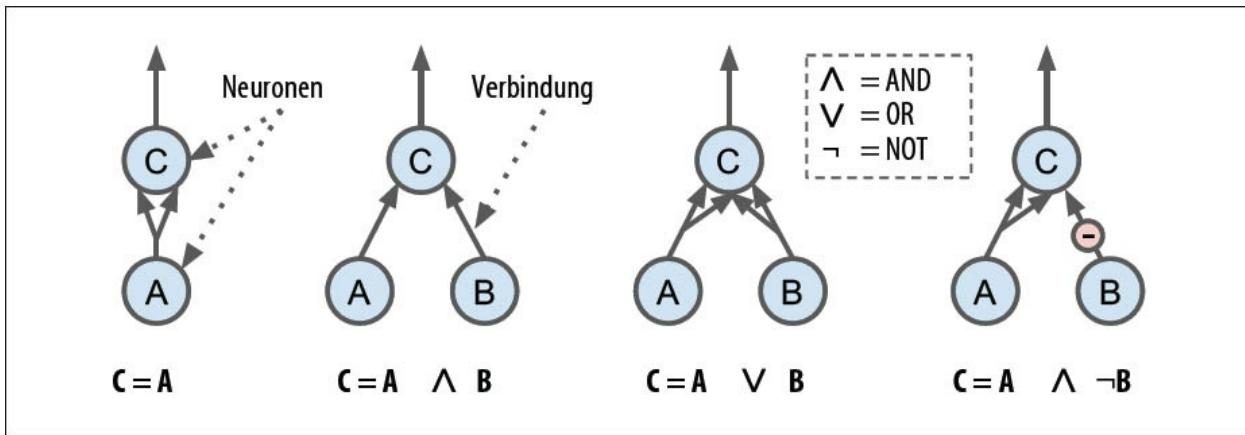


Abbildung 10-3: ANNs für einfache logische Berechnungen

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 *Threshold Logic Unit* (TLU), manchmal auch als *Linear Threshold Unit* (LTU) bezeichnet: Die Ein- und Ausgaben sind nun Zahlen (anstatt binäre Ein/Aus-Werte), und zu jeder Eingabeleitung gehört ein Gewicht. Die TLU berechnet eine gewichtete Summe ihrer Eingaben ($z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n = \mathbf{w}^T \mathbf{x}$), wendet dann eine *Aktivierungsfunktion* auf diese Summe an und gibt das Ergebnis aus: $h_{\mathbf{w}}(\mathbf{x}) = \text{Aktivierungsfunktion}(z)$ mit $z = \mathbf{w}^T \mathbf{x}$.

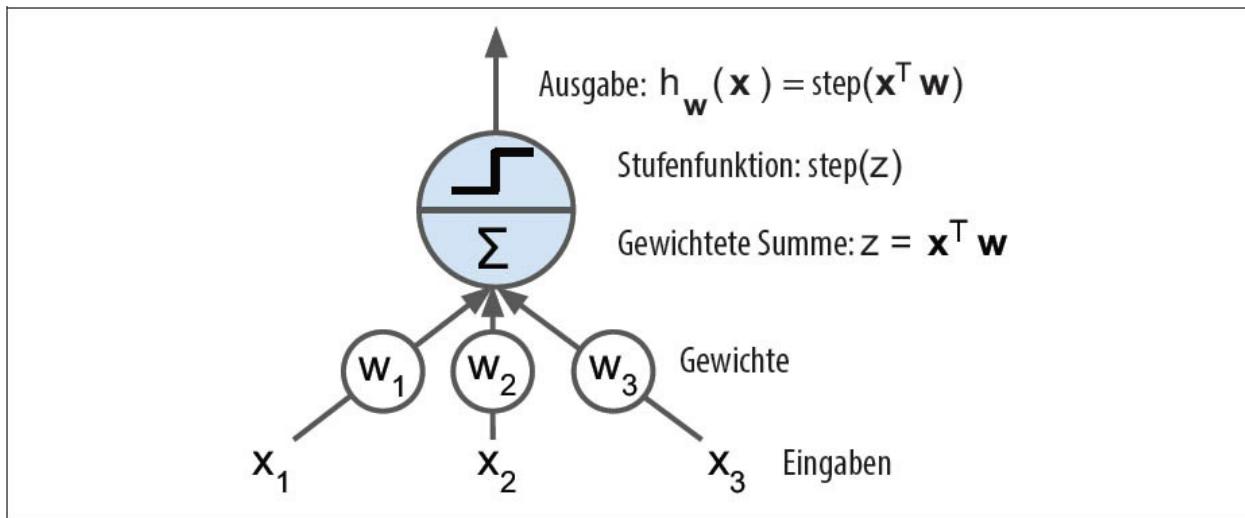


Abbildung 10-4: Threshold Logic Unit: ein künstliches Neuron, das eine gewichtete Summe seiner Eingaben berechnet und dann eine Aktivierungsfunktion anwendet

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 TLU 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 TLU 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 TLU 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 TLUs,⁷ wobei jede TLU mit allen Eingabewerten verbunden ist. Sind alle Neuronen in einer Schicht mit allen Neuronen in der vorherigen Schicht verbunden sind (also deren Eingabeneuronen), wird die Schicht als *vollständig verbundene Schicht* oder *Dense Layer* bezeichnet. Die Eingangswerte des Perzeptrons werden an spezielle Neuronen zum Durchreichen der Information übergeben, den *Eingabeneuronen*: Diese geben die erhaltenen Eingabedaten aus. Alle Eingabeneuronen bilden zusammen die *Eingabeschicht*. 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.

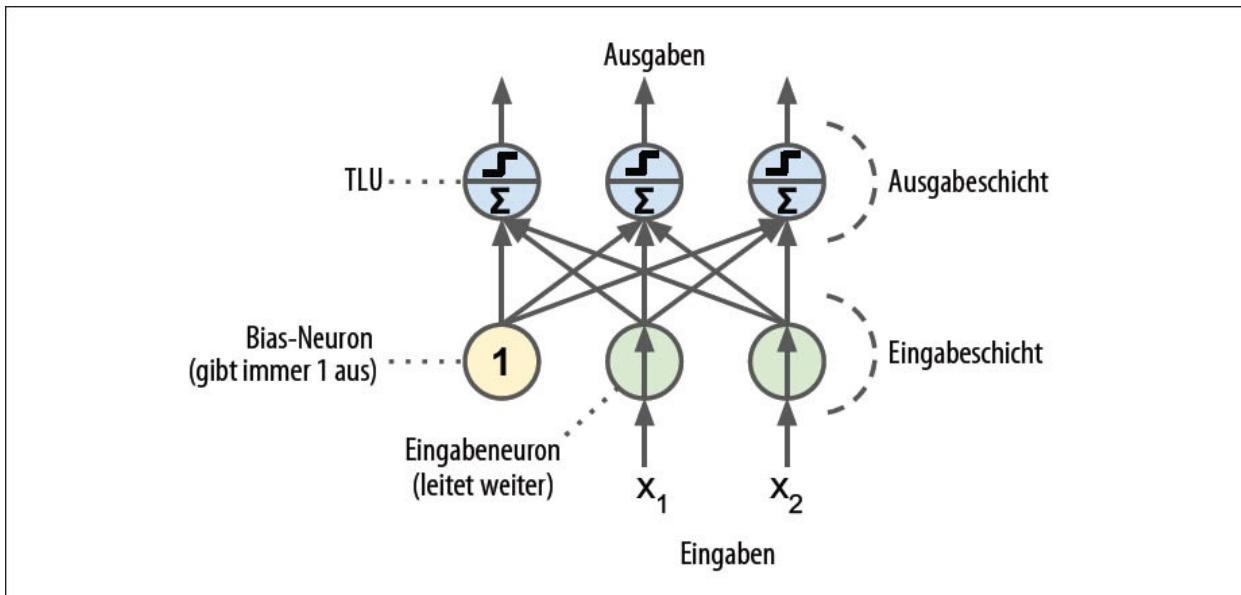


Abbildung 10-5: Perzeptron als Diagramm

Dank der Magie der linearen Algebra ermöglicht Formel 10-2 ein effizientes Berechnen der

Ausgabewerte einer Schicht künstlicher Neuronen für viele Instanzen auf einmal.

Formel 10-2: Die Ausgabewerte einer vollständig verbundenen Schicht berechnen

$$h_{\mathbf{w}, \mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

In dieser Gleichung gilt:

- Wie immer steht \mathbf{X} für die Matrix mit den Eingabemerkmalen. Sie enthält eine Zeile pro Instanz und eine Spalte pro Merkmal.
- Die Gewichtsmatrix \mathbf{W} enthält alle Verbindungsgewichte mit Ausnahme derjenigen des Bias-Neurons. Sie besitzt eine Zeile pro Eingangsneuron und eine Spalte pro künstliches Neuron in der Schicht.
- Der Bias-Vektor \mathbf{b} enthält alle Verbindungsgewichte zwischen dem Bias-Neuron und den künstlichen Neuronen. Er besitzt einen Bias-Term pro künstliches Neuron.
- Die Funktion ϕ wird als *Aktivierungsfunktion* bezeichnet: Handelt es sich bei den künstlichen Neuronen um TLUs, ist es eine Step-Funktion (aber wir werden gleich auch noch andere Aktivierungsfunktionen behandeln).

Wie aber lässt sich ein Perzepron trainieren? Der von Frank Rosenblatt vorgeschlagene Algorithmus zum Trainieren von Perzeptrons ist in weiten Teilen von der *hebbschen Lernregel* inspiriert. In seinem 1949 veröffentlichten Buch *The Organization of Behavior* (Wiley) 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 *hebbsche Lernregel* (oder *hebbsches Lernen*) bekannt; dabei wird die Verbindung zweier Neuronen immer dann verstärkt, wenn beide gemeinsam feuern. Perzeptrons lassen sich mit einer Variante dieser Regel trainieren, die die vom Netz bei der Vorhersage begangenen Fehler berücksichtigt; Verbindungen, die dabei helfen, den Fehler zu verringern, werden verstärkt. Beim Trainieren wird dem Perzepron 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 10-3](#) dargestellt.

Formel 10-3: 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 . Neuron und dem j . Ausgabeneuron.
- x_i ist der i . Wert des aktuellen Trainingsdatenpunkts.
- y_j ist die Ausgabe des j . Ausgabeneurons für den aktuellen Trainingsdatenpunkt.
- \hat{y}_j ist die Zielausgabe des j . 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.⁸ Dies nennt man das *Perzeptron-Konvergenztheorem*.

Scikit-Learn enthält die Klasse Perceptron, die ein einzelnes TTU-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

iris = load_iris()

X = iris.data[:, (2, 3)] # Länge, Breite von Kronblättern

y = (iris.target == 0).astype(np.int) # Iris setosa?

per_clf = Perceptron()

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 Grund, die logistische Regression dem Perzeptron 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. beim *exklusiven 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 Enttäuschung groß, und in der Folge wandten sich viele Wissenschaftler von neuronalen Netzen ganz ab, um sich übergeordneten Aufgabenstellungen wie Logik, Problemlösung und Suche zuzuwenden.

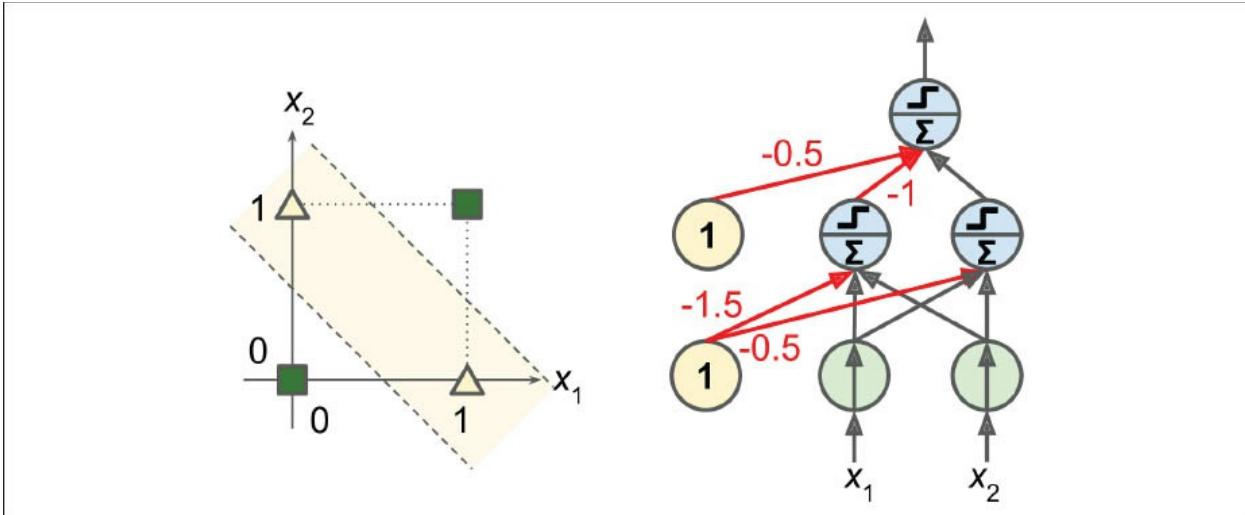


Abbildung 10-6: XOR-Klassifikationsproblem und ein MLP zu dessen Lösung

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 Perzeptron* (MLP). Ein MLP ist insbesondere dazu 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 Eingaben 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. Alle Verbindungen haben ein Gewicht von 1, abgesehen von den vier Verbindungen, bei denen das Gewicht angeschrieben steht. Versuchen Sie einmal, zu überprüfen, ob dieses Netz tatsächlich das XOR-Problem löst!

Mehrschichtiges Perzeptron und Backpropagation

Ein MLP setzt sich aus einer *Eingabeschicht* (zum Durchreichen) und einer oder mehreren Schichten von TLUs zusammen, den *verborgenen Schichten*, und einer letzten Schicht TLUs, der *Ausgabeschicht* (siehe Abbildung 10-7). Bis auf die Ausgabeschicht enthält jede Schicht ein Bias-Neuron und ist mit der nächsten Schicht vollständig verbunden. Die Schichten nahe an der Eingabeschicht werden meist als *untere Schichten* bezeichnet, die nahe an der Ausgabeschicht als *obere Schichten*. Alle Schichten außer der Ausgabeschicht enthalten ein Bias-Neuron und sind vollständig mit der nächsten Schicht verbunden.

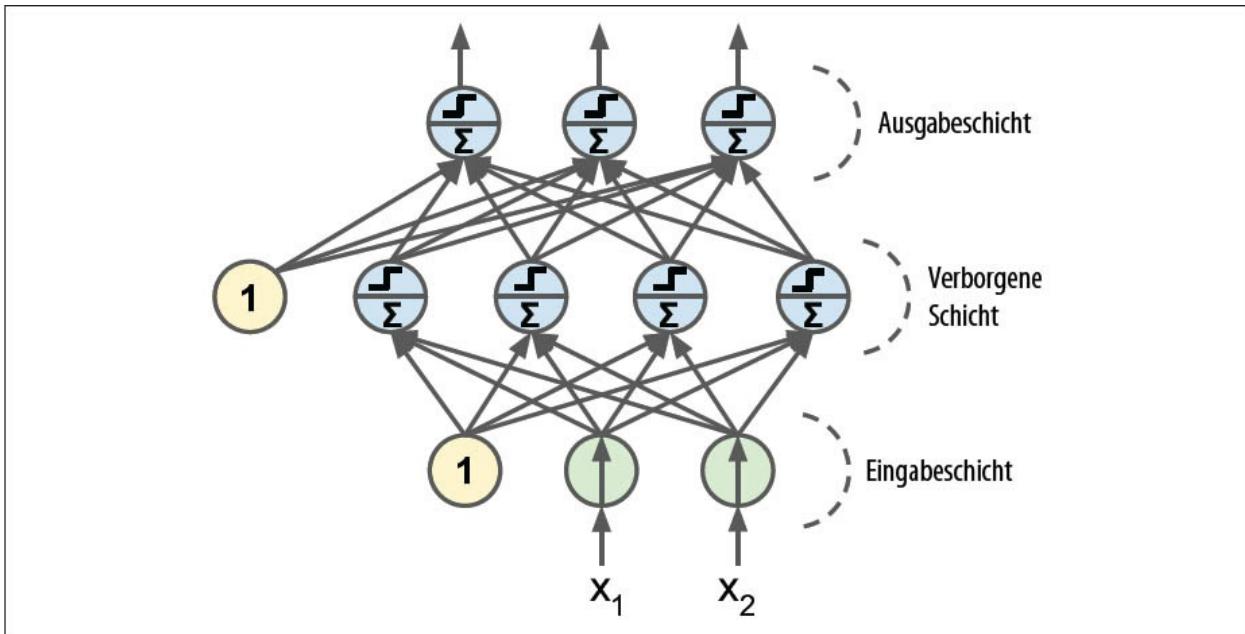


Abbildung 10-7: Architektur eines mehrschichtigen Perzeptrons mit zwei Eingabeneuronen, einer verborgenen Schicht mit vier Neuronen und drei Ausgabeneuronen (die Bias-Neuronen sind hier dargestellt, meist aber implizit mit gemeint)



Das Signal läuft nur in eine Richtung (von der Eingabe zur Ausgabe), daher handelt es sich bei dieser Architektur um ein Beispiel für ein *Feed-Forward-Netz* (Feedforward Neural Network, FNN).

Enthält ein ANN viele verborgene Schichten,⁹ nennt man es ein *tiefes neuronales Netz* (Deep Neural Network, DNN). Die Forschungsrichtung des Deep Learning studiert solche DNNs und Modelle mit tiefen Rechen-Stacks. Aber viele Menschen reden schon vom Deep Learning, wenn neuronale Netze (selbst ganz flache) überhaupt beteiligt sind.

Viele Jahre lang haben Forscher vergeblich nach einer Möglichkeit zum Trainieren von MLPs gesucht. Im Jahr 1986 aber publizierten D. E. Rumelhart, Geoffrey Hinton und Ronald Williams einen wegweisenden Artikel (<https://homl.info/44>)¹⁰, der den heutzutage immer noch verwendeten *Backpropagation*-Algorithmus bekannt machte. Kurz gesagt, handelt es sich um das Gradientenverfahren (siehe [Kapitel 4](#)) mit einer effizienten Technik zum automatischen Berechnen der Gradienten¹¹ in nur zwei Durchgängen durch das Netz (einer vorwärts, einer rückwärts). Der Backpropagation-Algorithmus kann den Gradienten des Fehlers im Netz bezüglich jedes einzelnen Modellparameters berechnen. Mit anderen Worten: Er kann ermitteln, wie jedes Verbindungsgewicht und jeder Bias-Term angepasst werden sollte, um den Fehler zu reduzieren. Hat er diese Gradienten bestimmt, führt er einfach einen normalen Schritt im Gradientenverfahren durch, und der gesamte Prozess wird wiederholt, bis sich das Netz der Lösung genähert hat.

Das automatische Berechnen von Gradienten wird *Automatic Differentiation* oder *Autodiff* genannt. Es gibt diverse Autodiff-Techniken mit unterschiedlichen Vor- und Nachteilen. Die bei



der Backpropagation genutzte Technik nennt sich *Reverse-Mode-Autodiff*. Sie ist schnell und exakt und passt sehr gut, wenn die zu differenzierende Funktion viele Variablen (zum Beispiel Verbindungsgewichte) und wenige Ausgabewerte (zum Beispiel einen Fehlerwert) besitzt. Wollen Sie mehr über Autodiff erfahren, schauen Sie sich [Anhang D](#) an.

Betrachten wir uns diesen Algorithmus etwas genauer:

- Es wird immer ein Mini-Batch gleichzeitig verarbeitet (beispielsweise mit 32 Datenpunkten), und der gesamte Trainingsdatensatz wird mehrfach durchlaufen. Jeder Durchlauf wird als *Epoche* bezeichnet.
- Jeder Mini-Batch wird an die Eingabeschicht des Netzes übergeben, die ihn an die erste verborgene Schicht weitergibt. Der Algorithmus berechnet dann die Ausgabe aller Neuronen in dieser Schicht (für jede Instanz im Mini-Batch). Das Ergebnis wird an die nächste Schicht weitergegeben, deren Ausgabe wird berechnet und an die nächste Schicht weitergegeben und so weiter, bis wir das Ergebnis der letzten Ausgabeschicht erhalten. Das ist der *Vorwärtsdurchlauf* – er entspricht dem Erstellen von Vorhersagen, nur dass man alle Zwischenergebnisse aufhebt, da sie für den Rückwärtsdurchlauf benötigt werden.
- Als Nächstes misst der Algorithmus den Ausgabefehler des Netzes (er nutzt eine Fehlerfunktion, die die gewünschte Ausgabe mit der tatsächlichen Ausgabe vergleicht und einen Messwert für den Fehler liefert).
- Nun berechnet er, wie viel jede Ausgabeverbindung zum Fehler beigetragen hat. Das geschieht analytisch durch das Anwenden der *Kettenregel* (die vielleicht grundlegendste Rechenregel), wodurch dieser Schritt schnell und exakt ist.
- Dann misst der Algorithmus, wie viele der Beiträge zum Fehler von jeder Verbindung in der Schicht darunter kamen (erneut unter Anwendung der Kettenregel), und er arbeitet sich rückwärts durch die Schichten, bis er die Eingabeschicht erreicht. Wie zuvor erwähnt, misst dieser Rückwärtsdurchlauf effizient den Fehlergradienten über alle Verbindungsgewichte im Netz, indem er ihn rückwärts durch das Netz propagiert (daher der Name des Algorithmus).
- Schließlich führt der Algorithmus einen Schritt im Gradientenverfahren durch, um alle Verbindungsgewichte im Netz anzupassen. Dabei greift er auf die Fehlergradienten zurück, die er gerade berechnet hat.

Dieser Algorithmus ist so wichtig, dass es sich lohnt, ihn nochmals zusammenzufassen: 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).



Es ist wichtig, alle Verbindungsgewichte der verborgenen Schichten mit Zufallswerten zu initialisieren, denn sonst wird das Training fehlschlagen. Initialisieren Sie zum Beispiel alle Gewichte und Biase mit null, werden alle Neuronen in einer Schicht identisch sein, und die Backpropagation wird sie exakt gleich behandeln, womit sie identisch bleiben. Oder anders

gesagt: Auch wenn Sie Hunderte von Neuronen pro Schicht haben, wird sich Ihr Modell so verhalten, als gäbe es nur ein Neuron pro Schicht – das wäre nicht so klug. Wenn Sie stattdessen die Gewichte zufällig initialisieren, brechen Sie die Symmetrie und erlauben der Backpropagation, ein diverses Team von Neuronen zu trainieren.

Damit dieser Algorithmus gut funktioniert, nahmen die Autoren eine wichtige Änderung an der Architektur des MLP vor: Sie ersetzten die Stufenfunktion mit der logistischen (Sigmoid-)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 zwischen 0 und 1 bei der logistischen Funktion). Damit wird die Ausgabe jeder Schicht zu Beginn des Trainings tendenziell auf 0 zentriert. Dies beschleunigt bisweilen die Konvergenz.

Die ReLU-Funktion $\text{ReLU}(z) = \max(0, z)$

Diese Funktion ist stetig, aber bei $z = 0$ leider nicht differenzierbar (die Steigung ändert sich abrupt, wodurch die Gradientenmethode umherspringen kann), und seine Ableitung ist für $z < 0$ gleich null. In der Praxis funktioniert diese Funktion aber sehr gut und ist außerdem schnell berechenbar, weshalb sie sich zum Standard entwickelt hat.¹² Wichtiger ist, dass sie keinen maximalen Ausgabewert besitzt, was einige Probleme des Gradientenverfahrens umgeht (wir kommen in [Kapitel 11](#) darauf zu sprechen).

Diese beliebten Aktivierungsfunktionen und ihre Ableitungen sind in [Abbildung 10-8](#) dargestellt. Aber Moment mal! Warum brauchen wir die Aktivierungsfunktionen überhaupt? Nun, wenn Sie mehrere lineare Transformationen verketteten, erhalten Sie immer eine lineare Transformation. Sind beispielsweise $f(x) = 2x + 3$ und $g(x) = 5x - 1$, dann führt ein Verketten dieser beiden linearen Funktionen zu einer anderen linearen Funktion: $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$.

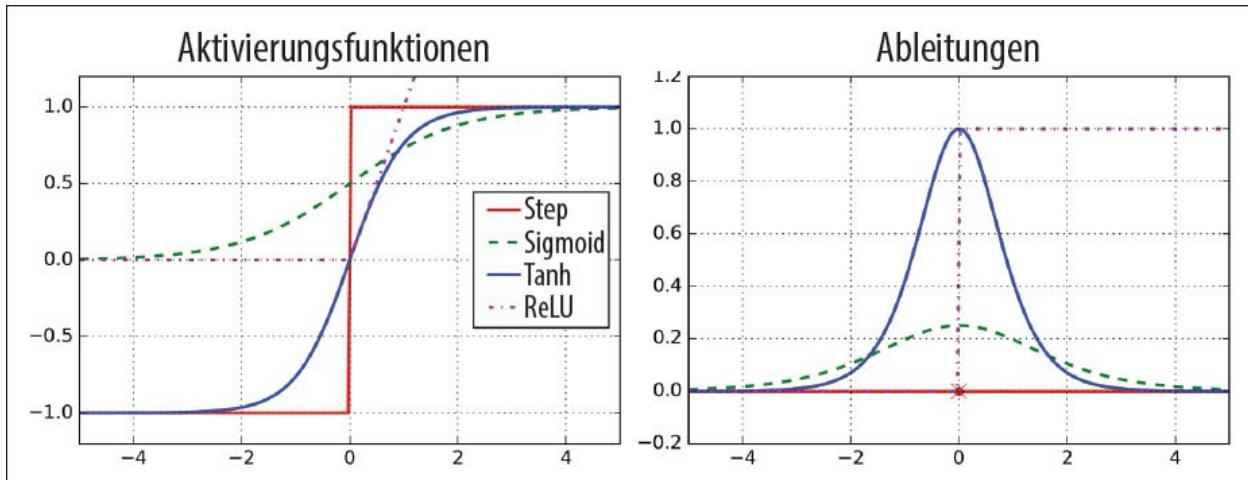


Abbildung 10-8: Aktivierungsfunktionen und ihre Ableitungen

Haben Sie also keine Nichtlinearität zwischen Ihren Schichten, entspricht selbst ein tiefer Stack mit Schichten einer einzelnen Schicht, und Sie können damit keine sehr komplexen Probleme lösen. Umgekehrt kann ein ausreichend großes ANN mit nichtlinearen Aktivierungsfunktionen theoretisch jede stetige Funktion approximieren.

Okay, Sie wissen jetzt, woher neuronale Netze stammen, wie ihre Architektur aussieht und wie deren Ergebnisse berechnet werden. Zudem haben Sie den Backpropagation-Algorithmus kennengelernt. Aber was genau können Sie damit machen?

Regressions-MLPs

MLPs können für Regressionsaufgaben eingesetzt werden. Wollen Sie einen einzelnen Wert vorhersagen (zum Beispiel einen Hauspreis, wenn man viele seiner Merkmale kennt), brauchen Sie nur ein einzelnes Ausgabeneuron: Seine Ausgabe ist der vorhergesagte Wert. Für eine multivariate Regression (also die Vorhersage mehrerer Werte auf einmal) benötigen Sie ein Ausgabeneuron pro Ausgabedimension. Um beispielsweise die Mitte eines Objekts in einem Bild zu finden, müssen Sie zweidimensionale Koordinaten vorhersagen, also brauchen Sie zwei Ausgabeneuronen. Wollen Sie auch noch eine Bounding Box um das Objekt legen, benötigen Sie zwei weitere Zahlen – die Breite und Höhe des Objekts. Also landen Sie so bei vier Ausgabeneuronen.

Im Allgemeinen wollen Sie beim Erstellen eines MLP zur Regression keine Aktivierungsfunktionen für die Ausgabeneuronen verwenden, sodass sie beliebige Zahlenwerte liefern können. Wollen Sie garantieren, dass die Ausgabe immer positiv sein wird, können Sie die ReLU-Aktivierungsfunktion in der Ausgabeschicht nutzen. Alternativ setzen Sie die *Softplus*-Aktivierungsfunktion ein, bei der es sich um eine geglättete Form der ReLU handelt: $\text{softplus}(z) = \log(1 + \exp(z))$. Sie ist nahe null, wenn z negativ ist, und nahe an z für positive Werte. Wollen Sie schließlich sicherstellen, dass die Vorhersagen in einen gegebenen Wertebereich fallen, können Sie die logistische Funktion oder den Tangens hyperbolicus nutzen und die Labels dann auf den passenden Bereich skalieren: 0 bis 1 für die logistische Funktion und -1 bis 1 für den Tangens hyperbolicus.

Die Fehlerfunktion, die während des Trainings genutzt wird, ist meist der mittlere quadratische Fehler, aber wenn Sie viele Ausreißer im Trainingsdatensatz haben, ziehen Sie eventuell den mittleren absoluten Fehler vor. Alternativ können Sie den Huber-Fehler verwenden, bei dem es sich um eine Kombination aus beiden handelt.

- * Der Huber-Fehler ist quadratisch, wenn der Fehler kleiner als ein Grenzwert δ ist (meist 1), aber linear, wenn der Fehler größer als δ ist. Der lineare Teil sorgt dafür, dass die Funktion weniger empfindlich als der mittlere quadratische Fehler auf Ausreißer reagiert, während es der quadratische Teil erlaubt, schneller zu konvergieren und genauer als der mittlere absolute Fehler zu sein.

Tabelle 10-1 fasst die typische Architektur eines Regressions-MLP zusammen.

Tabelle 10-1: Typische Architektur eines Regressions-MLP

Hyperparameter	Typische Werte
Anzahl Eingabeneuronen	eines pro Eingabemerkmal (z.B. $28 \times 28 = 784$ für MNIST)
Anzahl verborgene Schichten	abhängig vom Problem, meist 1 bis 5
Anzahl Neuronen pro verborgene Schicht	abhängig vom Problem, meist 10 bis 100
Anzahl Ausgabeneuronen	1 pro Vorhersagedimension
Verborgene Aktivierung	ReLU (oder SELU, siehe Kapitel 11)
Ausgabeaktivierung	keine, ReLU/Softplus (für positive Ausgaben) oder logistisch/tanh (für beschränkte Ausgaben)
Verlustfunktion	MSE oder MAE/Huber (bei Ausreißern)

Klassifikations-MLPs

MLPs können auch zum Klassifizieren eingesetzt werden. Für ein binäres Klassifikationsproblem benötigen Sie nur ein einzelnes Ausgabeneuron, das die logistische Aktivierungsfunktion verwendet: Die Ausgabe wird eine Zahl zwischen 0 und 1 sein, die Sie als geschätzte Wahrscheinlichkeit der positiven Kategorie interpretieren können. Die geschätzte Wahrscheinlichkeit der negativen Kategorie ist 1 minus dieser Zahl.

MLPs können auch einfach zur binären Klassifikation mit mehreren Labels eingesetzt werden (siehe [Kapitel 3](#)). Sie könnten zum Beispiel ein E-Mail-Klassifikationssystem nutzen, das vorhersagt, ob es sich bei einer eintreffenden E-Mail um Ham oder Spam handelt, und gleichzeitig schätzt, ob es eine dringende oder keine dringende E-Mail ist. In diesem Fall bräuchten Sie zwei Ausgabeneuronen, die beide die logistische Aktivierungsfunktion verwenden: Die erste würde die Wahrscheinlichkeit ausgeben, dass es sich bei der E-Mail um Spam handelt, die zweite, ob sie dringend ist. Allgemeiner gesagt, würden Sie ein Ausgabeneuron pro positive Kategorie nutzen. Beachten Sie, dass sich die Ausgabewahrscheinlichkeiten nicht zu 1 aufsummieren müssen. Dadurch kann das Modell eine beliebige Kombination von Labels ausgeben: Sie können nicht dringenden Ham, dringenden Ham, nicht dringenden Spam und vielleicht sogar dringenden Spam haben (auch wenn das vermutlich ein Fehler sein dürfte).

Kann jede Instanz nur zu einer einzigen von drei oder mehr Kategorien gehören (zum Beispiel die Kategorien 0 bis 9 für die Ziffernbilderkennung), brauchen Sie ein Ausgabeneuron pro Kategorie, und Sie sollten die Softmax-Aktivierungsfunktion für die gesamte Ausgabeschicht verwenden (siehe Abbildung 10-9). Die Softmax-Funktion (vorgestellt in Kapitel 4) stellt sicher, dass alle geschätzten Wahrscheinlichkeiten zwischen 0 und 1 liegen und aufsummiert 1 ergeben (was notwendig ist, wenn die Kategorien exklusiv sind). Das nennt sich Multiclass-Klassifikation.

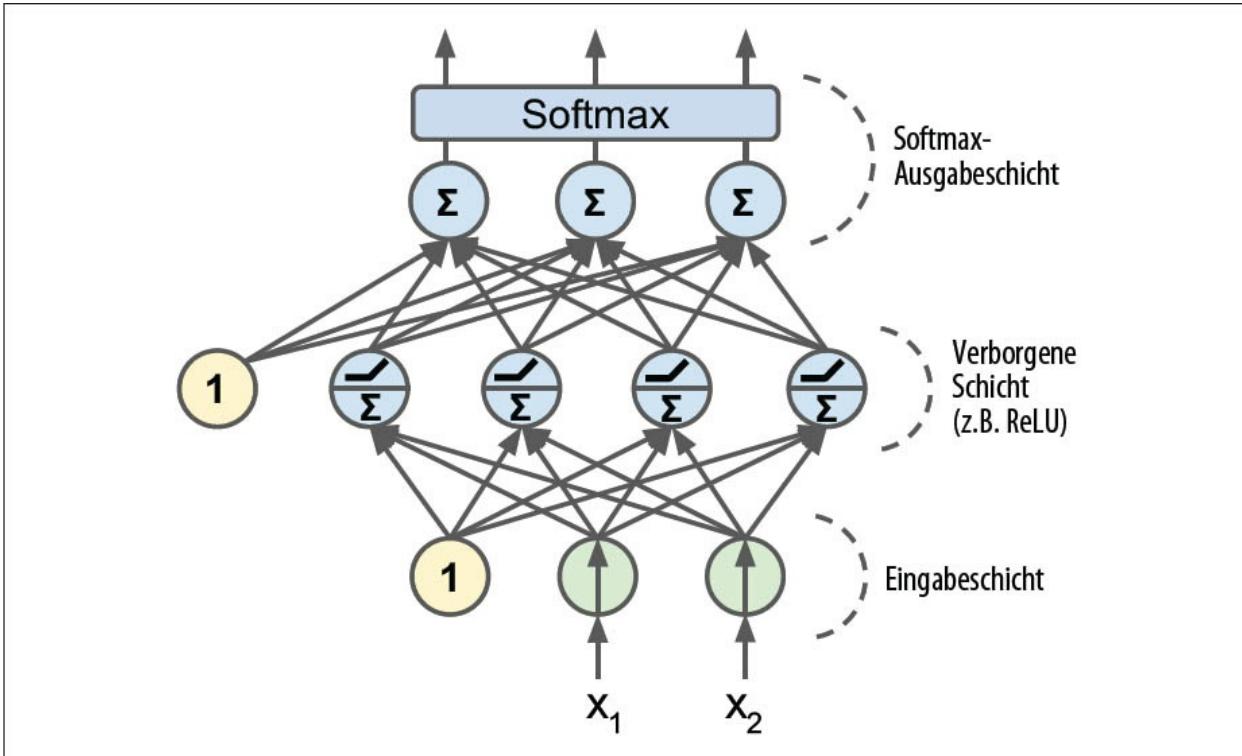


Abbildung 10-9: Ein modernes MLP (mit ReLU und Softmax) zur Klassifikation

Bei der Verlustfunktion ist die Kreuzentropie-Verlustfunktion (auch als Log-Loss bezeichnet, siehe Kapitel 4) im Allgemeinen eine gute Wahl, da wir Wahrscheinlichkeitsverteilungen vorhersagen.

In Tabelle 10-2 ist die typische Architektur eines Klassifikations-MLP zusammengefasst.

Tabelle 10-2: Typische Architektur eines Klassifikations-MLP

Hyperparameter	Binäre Klassifikation	Multilabel-Binärklassifikation	Multiclass-Klassifikation
Eingangs- und verborgene Schichten	wie bei Regression	wie bei Regression	wie bei Regression
Anzahl Ausgabeneuronen	1	1 pro Label	1 pro Kategorie
Ausgabeschichtaktivierung	logistisch	logistisch	Softmax
Verlustfunktion	Kreuzentropie	Kreuzentropie	Kreuzentropie

Bevor wir weitermachen, empfehle ich Ihnen, Übung 1 am Ende dieses Kapitels durchzuarbeiten. Sie werden mithilfe des *TensorFlow Playground* mit verschiedenen Architekturen neuronaler Netze herumspielen und ihre Ergebnisse visualisieren. Das wird sehr nützlich sein, um MLPs besser zu verstehen – vor allem die Effekte all der Hyperparameter (Anzahl der Schichten und Neuronen, Aktivierungsfunktionen und so weiter).

Jetzt haben wir alle Konzepte zusammen, um damit zu beginnen, MLPs mit Keras zu implementieren!

MLPs mit Keras implementieren

Keras ist eine High-Level-Deep-Learning-API, die es Ihnen ermöglicht, alle möglichen Arten von neuronalen Netzen einfach zu bauen, zu trainieren, auszuwerten und auszuführen. Ihre Dokumentation (oder Spezifikation) steht unter <https://keras.io> zur Verfügung. Die Referenzimplementierung (<https://github.com/kerasteam/keras>), ebenfalls als Keras bezeichnet, wurde von François Chollet als Teil eines Forschungsprojekts¹³ entwickelt und als Open-Source-Projekt im März 2015 veröffentlicht. Es fand schnell Verbreitung, weil es so einfach zu nutzen war, Flexibilität bot und ein sehr schönes Design besaß. Um die aufwendigen Berechnungen für die neuronalen Netze durchzuführen, baut diese Referenzimplementierung auf ein Rechen-Backend. Aktuell können Sie aus drei verbreiteten Open-Source Deep-Learning-Bibliotheken wählen: TensorFlow, Microsoft Cognitive Toolkit (CNTK) oder Theano. Um nicht zu sehr zu verwirren, werden wir uns daher auf diese Referenzimplementierung als *Multibackend-Keras* beziehen.

Seit Ende 2016 wurden noch andere Implementierungen veröffentlicht. Sie können Keras jetzt auf Apache MXNet, Apples Core ML, JavaScript oder TypeScript (um Keras-Code in einem Webbrowser ausführen zu können) und PlaidML (das auf allen möglichen GPU-Devices laufen kann, nicht nur auf Nvidia) laufen lassen. Zudem bringt TensorFlow mittlerweile seine eigene Keras-Implementierung `tf.keras` mit. Es unterstützt nur TensorFlow als Backend, bietet dafür aber ein paar sehr nützliche Zusatzfeatures (siehe [Abbildung 10-10](#)): So unterstützt es beispielsweise die Data-API von TensorFlow, wodurch das Laden und Vorverarbeiten von Daten sehr einfach wird. Daher werden wir `tf.keras` in diesem Buch einsetzen. Aber in diesem Kapitel werden wir keines der TensorFlow-spezifischen Features verwenden, daher sollte der Code auch mit anderen Keras-Implementierungen laufen (zumindest in Python), wenn Sie eventuell kleinere Anpassungen vornehmen, zum Beispiel beim Ändern der Imports.

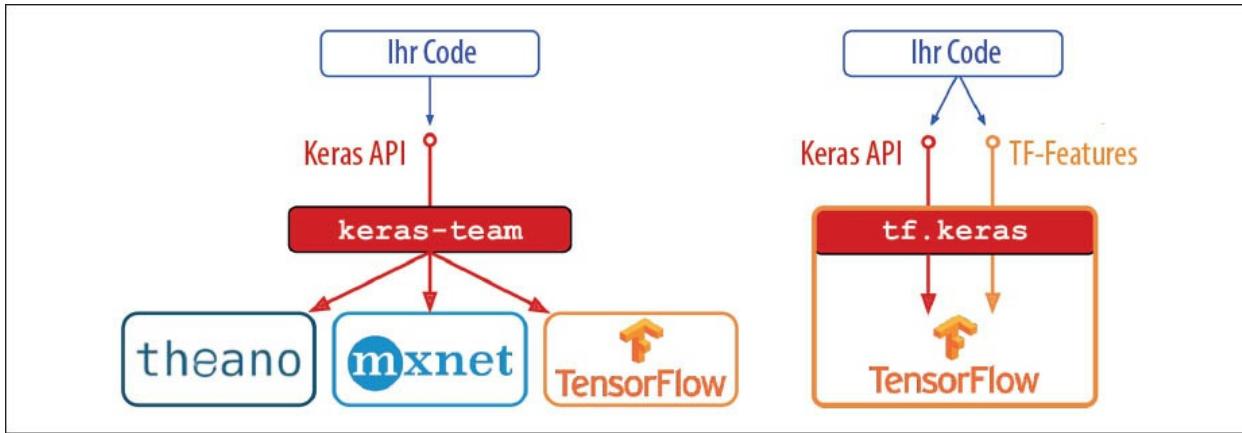


Abbildung 10-10: Zwei Implementierungen der Keras-API: Multibackend-Keras (links) und tf.keras (rechts)

Die nach Keras und TensorFlow beliebteste Deep-Learning-Bibliothek ist PyTorch (<https://pytorch.org>) von Facebook. Das Gute daran ist, dass ihre API der von Keras ähnelt (zum Teil, da beide APIs durch Scikit-Learn und Chainer (<https://chainer.org>) beeinflusst wurden) – kennen Sie also Keras, ist es nicht schwer, zu PyTorch zu wechseln, wenn Sie das jemals wollen. Die Beliebtheit von PyTorch ist 2018 exponentiell gewachsen, vor allem aufgrund der Einfachheit und der ausgezeichneten Dokumentation, die nicht gerade eine Stärke von TensorFlow 1.x war. Aber TensorFlow 2 ist durchaus genauso einfach wie PyTorch, da es Keras als offizielle High-End-API übernommen hat und die Entwickler den Rest der API vereinfacht und aufgeräumt haben. Die Dokumentation wurde ebenfalls komplett neu organisiert, und es ist jetzt viel einfacher, das zu finden, was Sie wissen müssen. Umgekehrt wurden PyTorchs größte Schwächen (beispielsweise die beschränkte Portierbarkeit und keine Analyse von Rechengraphen) in PyTorch 1.0 angegangen. Ein gesunder Wettbewerb ist für jeden von Vorteil.

Okay, Zeit, in den Code einzusteigen! Da tf.keras mit TensorFlow verbunden ist, beginnen wir damit, TensorFlow zu installieren.

TensorFlow 2 installieren

Sofern Sie Jupyter und Scikit-Learn anhand der Schritte in [Kapitel 2](#) installiert haben, nutzen Sie pip zum Installieren von TensorFlow. Haben Sie eine isolierte Umgebung mit virtualenv erstellt, müssen Sie es erst aktivieren:

```
$ cd $ML_PATH                      # Ihr ML-Arbeitsverzeichnis (z. B. $HOME/ml)
$ source my_env/bin/activate        # auf Linux oder macOS
$ .\my_env\Scripts\activate       # auf Windows
```

Als Nächstes installieren Sie TensorFlow 2 (wenn Sie keine virtualenv nutzen, benötigen Sie Administratorrechte, oder Sie ergänzen die Option `--user`):

```
$ python3 -m pip install --upgrade tensorflow
```

Für eine GPU-Unterstützung müssen Sie aktuell tensorflow-gpu statt tensorflow installieren, aber das TensorFlow-Team arbeitet daran, eine einzige Bibliothek zu erstellen, die Systeme mit und ohne GPU unterstützt. Sie müssen zur GPU-Unterstützung trotzdem weitere Bibliotheken installieren (mehr dazu finden Sie unter <https://tensorflow.org/install>). Wir werden uns GPUs genauer in [Kapitel 19](#) anschauen.

Um Ihre Installation zu testen, öffnen Sie eine Python-Shell oder ein Jupyter-Notebook, importieren TensorFlow und tf.keras und geben dann deren Versionen aus:

```
>>> import tensorflow as tf  
>>> from tensorflow import keras  
>>> tf.__version__  
'2.0.0'  
>>> keras.__version__  
'2.2.4-tf'
```

Die zweite Version ist die der Keras-API, die durch tf.keras implementiert wurde. Beachten Sie, dass Sie auf -tf endet und damit hervorhebt, dass tf.keras die Keras-API implementiert (plus ein paar TensorFlow-spezifische Features).

Jetzt können wir tf.keras einsetzen! Beginnen wir damit, einen einfachen Bildklassifikator zu bauen.

Einen Bildklassifikator mit der Sequential API erstellen

Zuerst müssen wir einen Datensatz laden. In diesem Kapitel werden wir Fashion MNIST nutzen, bei dem es sich um einen Drop-in-Ersatz von MNIST (siehe [Kapitel 3](#)) handelt. Es hat genau das gleiche Format wie MNIST (70.000 Graustufenbilder mit jeweils 28×28 Pixeln und 10 Kategorien), aber die Bilder enthalten Modeartikel statt handgeschriebener Ziffern, daher ist jede Kategorie vielfältiger, und das Problem ist deutlich herausfordernder als bei MNIST. So erreicht beispielsweise ein einfaches lineares Modell bei MNIST 92% Genauigkeit, aber nur 83% bei Fashion MNIST.

Den Datensatz mit Keras laden

Keras stellt einige Hilfsfunktionen zum Laden verbreiteter Datensätze bereit, unter anderem MNIST, Fashion MNIST und die kalifornischen Immobiliendaten, die wir in [Kapitel 2](#) genutzt haben. Laden wir Fashion MNIST:

```
fashion_mnist = keras.datasets.fashion_mnist  
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

Laden Sie MNIST oder Fashion MNIST über Keras statt über Scikit-Learn, besteht ein wichtiger Unterschied darin, dass jedes Bild als 28×28 -Array geladen wird und nicht als eindimensionales Array der Größe 784. Zudem werden die Pixel-Intensitäten als Integer (von 0 bis 255) und nicht als Gleitkommazahlen (von 0,0 bis 255,0) repräsentiert. Schauen wir uns die Form und den Datentyp des Trainingsdatensatzes an:

```
>>> X_train_full.shape  
(60000, 28, 28)  
>>> X_train_full.dtype  
dtype('uint8')
```

Der Datensatz ist schon in einen Trainings- und einen Testdatensatz aufgeteilt, aber es gibt keinen Validierungsdatensatz, daher werden wir jetzt einen erzeugen. Und da wir das neuronale Netz mit der Gradientenmethode trainieren werden, müssen wir die Eingabemerkmale skalieren. Aus Gründen der Einfachheit werden wir die Pixel-Intensitäten auf den Bereich von 0 bis 1 herunterrechnen, indem wir sie durch 255,0 dividieren (das konvertiert sie gleichzeitig in Gleitkommazahlen):

```
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0  
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]  
X_test = X_test / 255.0
```

Hat das Label bei MNIST den Wert 5, entspricht das Bild auch einer handgeschriebenen Ziffer 5. Das ist einfach. Für Fashion MNIST benötigen wir aber die Liste mit Kategoriennamen, damit wir wissen, womit wir es zu tun haben:

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",  
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

So zeigt beispielsweise das erste Bild im Trainingsdatensatz eine Jacke:

```
>>> class_names[y_train[0]]  
'Coat'
```

[Abbildung 10-11](#) zeigt ein paar Beispiele aus dem Fashion-MNIST-Datensatz.



Abbildung 10-11: Beispiele aus dem Fashion MNIST

Das Modell mit der Sequential API erstellen

Erstellen wir jetzt das neuronale Netz. Dies ist ein Klassifikations-MLP mit zwei verborgenen Schichten:

```
model = keras.models.Sequential()

model.add(keras.layers.Flatten(input_shape=[28, 28]))

model.add(keras.layers.Dense(300, activation="relu"))

model.add(keras.layers.Dense(100, activation="relu"))

model.add(keras.layers.Dense(10, activation="softmax"))
```

Gehen wir den Code Zeile für Zeile durch:

- In der ersten Zeile wird ein `Sequential`-Modell erzeugt. Das ist die einfachste Form eines Keras-Modells für neuronale Netze, die nur aus einem einzelnen Stack mit Schichten besteht, die sequenziell verbunden sind. Das nennt sich die `Sequential API`.
- Als Nächstes bauen wir die erste Schicht und fügen sie dem Modell hinzu. Es handelt es sich um eine `Flatten`-Schicht, deren Aufgabe es ist, jedes Eingangsbild in ein eindimensionales Array umzuwandeln: Empfängt es Eingangsdaten `X`, berechnet es `X.reshape(-1, 28*28)`. Diese Schicht besitzt keine Parameter – sie dient nur der Vorverarbeitung. Da es sich um die erste Schicht im Modell handelt, sollten Sie `input_shape` angeben, das nicht die Batchgröße enthält, sondern nur die Form der Instanzen. Alternativ könnten Sie einen `keras.layers.InputLayer` als erste Schicht angeben und `input_shape=[28, 28]` setzen.
- Nun fügen wir eine verborgene `Dense`-Schicht mit 300 Neuronen hinzu. Sie nutzt die ReLU-Aktivierungsfunktion. Jede `Dense`-Schicht verwaltet ihre eigene Gewichtsmatrix

mit allen Verbindungsgewichten zwischen den Neuronen und ihren Eingangswerten. Auch verwaltet sie einen Vektor mit Bias-Termen (einem pro Neuron). Empfängt sie Eingangsdaten, berechnet sie [Formel 10-2](#).

- Dann fügen wir eine zweite verborgene Dense-Schicht mit 100 Neuronen hinzu, die ebenfalls die ReLU-Aktivierungsfunktion einsetzt.
- Und schließlich ergänzen wir noch eine Dense-Ausgabeschicht mit 10 Neuronen (eine pro Kategorie), die die Softmax-Aktivierungsfunktion einsetzt (weil es sich um exklusive Kategorien handelt).

☞ Eine Angabe von `activation="relu"` entspricht `activation=keras.activations.relu`. Andere Aktivierungsfunktionen stehen im Paket `keras.activations` zur Verfügung, von denen wir in diesem Buch viele einsetzen werden. Unter <https://keras.io/activations/> finden Sie die vollständige Liste.

Statt die Schichten wie eben eine nach der anderen hinzuzufügen, können Sie auch beim Erstellen des Sequential-Modells eine Liste mit Schichten mitgeben:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Codebeispiele von keras.io verwenden

Codebeispiele, die auf keras.io vorgestellt werden, funktionieren auch mit tf.keras gut, aber Sie müssen die Importanweisungen ändern. Schauen Sie sich zum Beispiel diesen Code von keras.io an:

```
from keras.layers import Dense
output_layer = Dense(10)
```

Sie müssen die Importe wie folgt anpassen:

```
from tensorflow.keras.layers import Dense
output_layer = Dense(10)
```

Oder verwenden Sie die vollständigen Pfade, wenn Ihnen das lieber ist:

```
from tensorflow import keras  
  
output_layer = keras.layers.Dense(10)
```

Das ist zwar mehr Text, aber ich nutze diesen Ansatz in diesem Buch, damit Sie problemlos erkennen können, welche Pakete zu verwenden sind und um Verwechslungen zwischen Standardklassen und eigenen Klassen zu vermeiden. In produktivem Code bevorzuge ich den ersten Code. Viele Menschen nutzen auch `from tensorflow.keras import layers`, gefolgt von `layers.Dense(10)`.

Die Modellmethode `summary()` zeigt alle Schichten des Modells an,¹⁴ einschließlich deren Namen (der automatisch erzeugt wird, sofern Sie ihn nicht beim Erstellen der Schicht setzen), der Ausgabeform (`None` heißt, dass die Batchgröße beliebig sein kann) und der Anzahl an Parametern. Die Zusammenfassung gibt am Ende noch die Gesamtzahl an Parametern aus – inklusive der Menge an trainierbaren und nicht trainierbaren Parametern. Hier haben wir nur trainierbare Parameter (wir werden in [Kapitel 11](#) noch Beispiele für nicht trainierbare Parameter sehen):

```
>>> model.summary()  
  
Model: "sequential"  
  
=====  


| Layer (type)      | Output Shape | Param # |
|-------------------|--------------|---------|
| flatten (Flatten) | (None, 784)  | 0       |
| dense (Dense)     | (None, 300)  | 235500  |
| dense_1 (Dense)   | (None, 100)  | 30100   |
| dense_2 (Dense)   | (None, 10)   | 1010    |

  
=====  
Total params: 266,610  
Trainable params: 266,610  
Non-trainable params: 0
```

Beachten Sie, dass Dense-Schichten oftmals wirklich *viele* Parameter besitzen. So enthält die erste Schicht beispielsweise 784×300 Verbindungsgewichte, dazu 300 Bias-Terme, was sich zu 235.500 Parametern aufsummieren! Damit besitzt das Modell ziemlich viel Flexibilität, um sich an die Trainingsdaten anzupassen, aber es heißt auch, dass das Modell dem Risiko des Overfitting unterliegt, insbesondere wenn Sie nicht viele Trainingsdaten haben. Wir werden darauf später noch zurückkommen.

Sie können leicht eine Liste mit den Schichten eines Modells erhalten, um eine davon über ihren Index anzusprechen, oder Sie gehen über den Namen vor:

```
>>> model.layers  
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,  
<tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,  
<tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,  
<tensorflow.python.keras.layers.core.Dense at 0x13240d240>]  
  
>>> hidden1 = model.layers[1]  
  
>>> hidden1.name  
  
'dense'  
  
>>> model.get_layer('dense') is hidden1  
  
True
```

Alle Parameter einer Schicht können über ihre Methoden `get_weights()` und `set_weights()` angesprochen werden. Für eine Dense-Schicht gehören dazu sowohl die Verbindungsgewichte wie auch die Bias-Terme:

```
>>> weights, biases = hidden1.get_weights()  
  
>>> weights  
  
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,  
        0.03859074, -0.06889391],  
       ...,  
       [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,  
        0.00272203, -0.06793761]], dtype=float32)  
  
>>> weights.shape
```

(784, 300)

```
>>> biases
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., ..., 0., 0., 0.], dtype=float32)
```

```
>>> biases.shape
```

```
(300, )
```

Beachten Sie, dass die Dense-Schicht die Verbindungsgewichte mit Zufallsraten (was wie schon erwähnt erforderlich ist, um die Symmetrie aufzubrechen) und die Biase mit Nullen initialisiert werden, was gut ist. Wollen Sie einmal eine andere Initialisierungsmethode verwenden, können Sie beim Erstellen der Schicht `kernel_initializer` (`Kernel` ist ein anderer Name für die Matrix mit den Verbindungsgewichten) oder `bias_initializer` setzen. Wir werden Initialisierer noch genauer in [Kapitel 11](#) besprechen, aber wenn Sie eine vollständige Liste haben wollen, finden Sie die unter <https://keras.io/initializers/>.



Die Form der Gewichtsmatrix hängt von der Anzahl an Eingangswerten ab. Darum wird empfohlen, `input_shape` anzugeben, wenn Sie die erste Schicht in einem Sequential-Modell erstellen. Aber auch wenn Sie sie nicht angeben, ist das in Ordnung – Keras wird einfach mit dem tatsächlichen Erstellen des Modells warten, bis es die Eingabeform kennt. Das passiert, wenn Sie entweder Daten übergeben (zum Beispiel beim Training) oder wenn Sie seine Methode `build()` aufrufen. Bevor das Modell nicht gebaut ist, haben die Schichten keine Gewichte, und Sie werden bestimmte Dinge nicht tun können (wie zum Beispiel die Zusammenfassung des Modells ausgeben oder es speichern). Wenn Sie also die Eingabeform beim Erstellen des Modells kennen, ist es auch am besten, sie anzugeben.

Das Modell komplizieren

Nachdem ein Modell erstellt wurde, müssen Sie dessen Methode `compile()` aufrufen, um die zu verwendende Verlustfunktion und den Optimizer festzulegen. Optional können Sie eine Liste zusätzlicher Metriken angeben, die während des Trainings und der Auswertung berechnet werden sollen:

```
model.compile(loss="sparse_categorical_crossentropy",
               optimizer="sgd",
               metrics=["accuracy"])
```



Der Einsatz von `loss="sparse_categorical_crossentropy"` entspricht `loss=keras.losses.sparse_categorical_crossentropy`. Genauso können Sie statt `optimizer="sgd"` auch `optimizer=keras.optimizers.SGD()` nutzen, und `metrics=["accuracy"]` entspricht `metrics=[keras.metrics.sparse_categorical_accuracy]` (beim Einsatz dieser Verlustfunktion). Wir werden in diesem Buch auch viele andere Verlustfunktionen, Optimizer und Metriken verwenden – die vollständigen Listen finden Sie unter <https://keras.io/losses>, <https://keras.io/optimizers> und

<https://keras.io/metrics>.

Dieser Code erfordert etwas Erläuterung. Zunächst verwenden wir die Verlustfunktion "sparse_categorical_crossentropy", weil wir es mit Sparse-Labels zu tun haben (das heißt, es gibt für jede Instanz nur einen Index für eine Zielkategorie – in unserem Fall 0 bis 9) und die Kategorien exklusiv sind. Hätten wir stattdessen eine Zielwahrscheinlichkeit pro Kategorie für jede Instanz (zum Beispiel One-Hot-Vektoren wie etwa [0., 0., 0., 1., 0., 0., 0., 0., 0.] für die Kategorie 3), müssten wir stattdessen die Verlustfunktion "categorical_crossentropy" verwenden. Geht es um Binärklassifikation (mit einem oder mehreren binären Labels), würden wir statt "softmax" die Aktivierungsfunktion "sigmoid" (also die logistische Funktion) in der Ausgabeschicht verwenden und die Verlustfunktion "binary_crossentropy" einsetzen.



Wollen Sie Sparse-Labels (also Kategorien-Indizes) in One-Hot-Vektor-Labels umwandeln, nutzen Sie die Funktion `keras.utils.to_categorical()`. In die umgekehrte Richtung kann `np.argmax()` mit `axis=1` zum Einsatz kommen.

Beim Optimierer heißt "sgd", dass wir das Modell mit dem einfachen stochastischen Gradientenverfahren trainieren. Mit anderen Worten: Keras wird den weiter oben beschriebenen Backpropagation-Algorithmus ausführen (also Reverse-Mode-Autodiff plus Gradientenverfahren). Wir werden effizientere Optimierer in [Kapitel 11](#) behandeln (sie verbessern das Gradientenverfahren, aber nicht den Autodiff).



Verwenden wir den SGD-Optimierer, ist es wichtig, die Lernrate anzupassen. Daher werden Sie im Allgemeinen statt `optimizer= SGD` eher `optimizer=keras.optimizer.SGD(lr=???)` verwenden, um die Lernrate zu setzen, da Ersteres standardmäßig `lr=0.01` nutzt.

Und da es sich um einen Klassifikator handelt, ist es schließlich nützlich, dessen "accuracy" während des Trainings und der Auswertung zu messen.

Das Modell trainieren und auswerten

Jetzt ist das Modell dazu bereit, trainiert zu werden. Dazu müssen wir einfach nur dessen Methode `fit()` aufrufen:

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [=====] - 3s 49us/sample - loss: 0.7218 - accuracy: 0.7660
                                         - val_loss: 0.4973 - val_accuracy: 0.8366
```

Epoch 2/30

```
55000/55000 [=====] - 2s 45us/sample - loss: 0.4840 - accuracy: 0.8327  
- val_loss: 0.4456 - val_accuracy: 0.8480
```

[...]

Epoch 30/30

```
55000/55000 [=====] - 3s 53us/sample - loss: 0.2252 - accuracy: 0.9192  
- val_loss: 0.2999 - val_accuracy: 0.8926
```

Wir übergeben die Eingabefeatures (`X_train`) und die Zielkategorien (`y_train`) sowie die Anzahl an Epochen beim Trainieren (ansonsten würde der Standardwert 1 genommen, was definitiv nicht ausreicht, um zu einer guten Lösung zu konvergieren). Auch übergeben wir einen Validierungsdatensatz (das ist optional). Keras wird am Ende jeder Epoche den Verlust und die zusätzlichen Metriken für dieses Set messen, was sehr nützlich ist, um zu sehen, wie gut das Modell wirklich arbeitet. Ist die Performance für den Trainingsdatensatz viel besser als beim Validierungsdatensatz, overfittet Ihr Modell vermutlich den Trainingsdatensatz (oder es ist ein Fehler, wie zum Beispiel eine Dateninkonsistenz zwischen Trainings- und Validierungsdatensatz).

Und das ist es! Das neuronale Netz ist trainiert.¹⁵ Keras zeigt während des Trainings nach jeder Epoche die Anzahl an bisher verarbeiteten Instanzen an (zusammen mit einem Fortschrittsbalken), dazu die durchschnittliche Trainingszeit pro Sample, den Verlust und die Genauigkeit (oder andere zusätzliche Metriken, um die Sie gebeten haben) sowohl für den Trainings- als auch für den Validierungsdatensatz. Sie können sehen, wie der Trainingsverlust abgesunken ist – ein gutes Zeichen – und die Validierungsgenauigkeit nach 30 Epochen einen Wert von 89,26% erreicht hat. Das ist nicht allzu weit von der Trainingsgenauigkeit entfernt, daher scheint es kein Overfitting zu geben.

* Statt einen Validierungsdatensatz über das Argument `validation_data` zu übergeben, können Sie auch mit `validation_split` das Verhältnis angeben, mit dem Keras den Trainingsdatensatz zum Validieren aufteilen soll. So weist beispielsweise `validation_split=0.1` Keras an, die letzten 10% der Daten (vor dem Durchmischen) zum Validieren zu verwenden.

War der Trainingsdatensatz sehr verzerrt – mit über- oder unterrepräsentierten Kategorien –, wäre es sinnvoll, beim Aufruf der Funktion `fit()` das Argument `class_weight` zu setzen und damit unterrepräsentierten Kategorien eine höhere Gewichtung und überrepräsentierten Kategorien eine niedrigere Gewichtung zu verpassen. Diese Gewichte werden dann von Keras beim Berechnen des Verlusts verwendet. Benötigen Sie Gewichte pro Instanz, setzen Sie das Argument `sample_weight` (sind sowohl `class_weight` wie auch `sample_weight` angegeben, multipliziert Keras sie). Gewichte pro Instanz können nützlich sein, wenn manche Instanzen ihre Labels von Experten erhielten, während andere über eine Cloudsourcing-Plattform

gekennzeichnet wurden – so können Sie den Ersteren mehr Gewicht verpassen. Auch können Sie dem Validierungsdatensatz Sample-Gewichte (aber keine Kategoriengewichte) mitgeben, indem Sie sie als drittes Element im Tupel `validation_data` anhängen.

Die Methode `fit()` liefert ein `History`-Objekt zurück, das die Trainingsparameter (`history.params`), die Liste mit durchlaufenen Epochen (`history.epoch`) und vor allem ein Dictionary (`history.history`) mit dem Verlust und zusätzlichen gemessenen Metriken vom Ende jeder Epoche für Trainings- und Validierungsdatensatz enthält (sofern vorhanden). Nutzen Sie dieses Dictionary, um einen pandas-Data Frame zu erstellen und dessen Methode `plot()` aufzurufen, erhalten Sie die Lernkurven, die in [Abbildung 10-12](#) zu sehen sind:

```
import pandas as pd  
  
import matplotlib.pyplot as plt  
  
pd.DataFrame(history.history).plot(figsize=(8, 5))  
  
plt.grid(True)  
  
plt.gca().set_ylim(0, 1) # vertikalen Bereich auf [0-1] setzen  
  
plt.show()
```

Sie sehen, dass sowohl die Trainings- wie auch die Validierungsgenauigkeit während des Trainings stetig steigen, während der Verlust von Training und Validierung sinkt. Sehr gut! Zudem liegen die Kurven von Training und Validierung nahe beieinander, was heißt, dass es nicht zu viel Overfitting gibt. In diesem speziellen Fall sieht das Modell so aus, als funktioniere es zu Beginn des Trainings besser mit dem Validierungs- als mit dem Trainingsdatensatz. Aber das stimmt nicht: Tatsächlich wird der Validierungsfehler am *Ende* jeder Epoche berechnet, während der Trainingsfehler mithilfe eines gleitenden Mittelwerts *während* jeder Epoche ermittelt wird. Daher sollte die Trainingskurve um eine halbe Epoche nach links verschoben sein. Wenn Sie das tun, werden Sie sehen, dass Trainings- und Validierungskurven zu Beginn des Trainings nahezu perfekt übereinanderliegen.

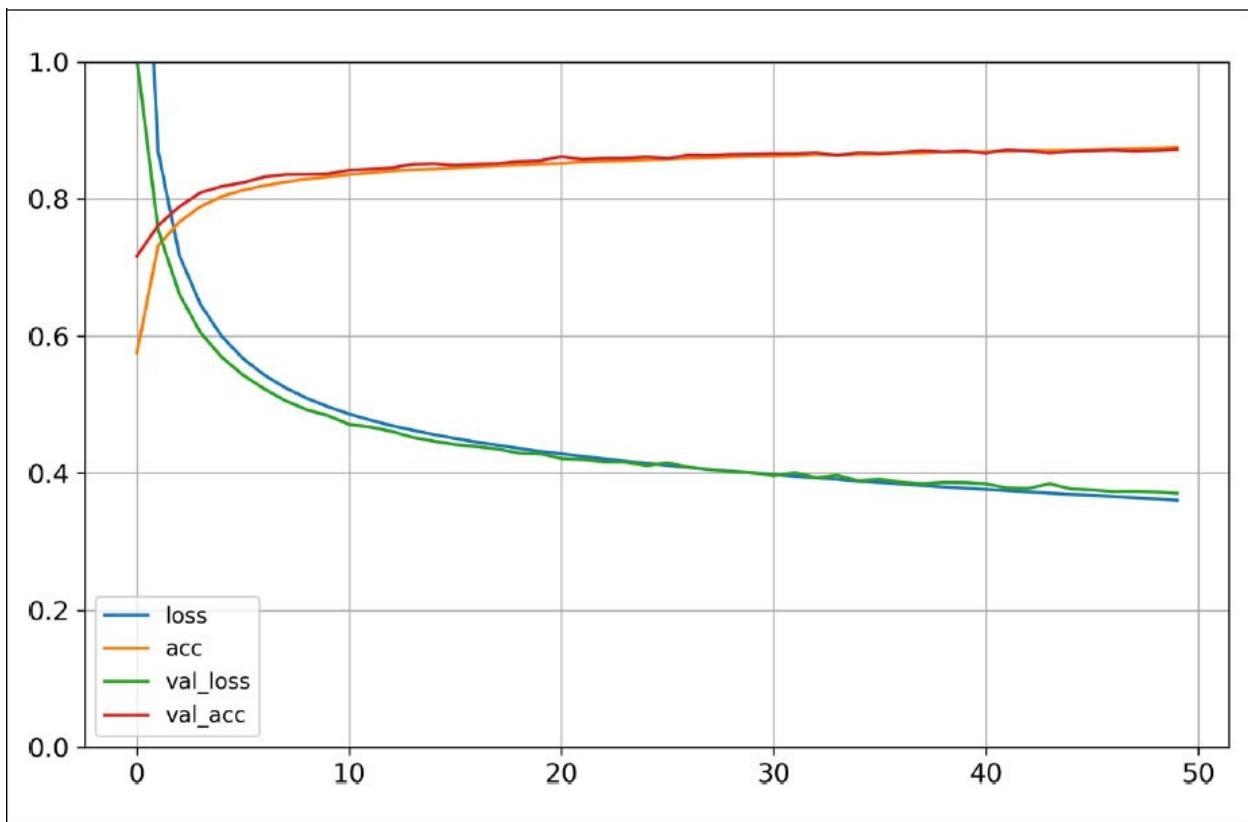


Abbildung 10-12: Lernkurven: durchschnittlicher Verlust und durchschnittliche Genauigkeit beim Training nach jeder Epoche und bei der Validierung nach jeder Epoche



Beim Ausgeben der Trainingskurve sollte diese um eine halbe Epoche nach links verschoben werden.

Die Performance des Trainingsdatensatzes schlägt schließlich die Validierungsperformance, was im Allgemeinen immer passiert, wenn Sie lange genug trainieren. Sie können erkennen, dass das Modell noch nicht ausreichend konvergiert ist, da der Validierungsverlust immer noch sinkt. Daher sollten Sie das Training weiter fortsetzen. Dazu müssen Sie einfach nur nochmals die Methode `fit()` aufrufen, da Keras dann mit dem Training dort weitermacht, wo es zuvor aufgehört hat (Sie sollten schließlich eine Validierungsgenauigkeit nahe an 89% erhalten).

Sind Sie mit der Leistung Ihres Modells nicht zufrieden, sollten Sie einen Schritt zurückgehen und die Hyperparameter anpassen. Zuerst prüfen Sie die Lernrate. Wenn das nicht hilft, probieren Sie es mit einem anderen Optimizer (und passen danach auch immer die Lernrate an). Ist die Leistung immer noch nicht gut, versuchen Sie, die Hyperparameter des Modells anzupassen, wie zum Beispiel die Anzahl an Schichten, die Anzahl an Neuronen pro Schicht oder die Art der Aktivierungsfunktionen für die verschiedenen Schichten. Sie können auch ausprobieren, an anderen Hyperparametern zu drehen, wie beispielsweise an der Batchgröße (sie kann in der Methode `fit()` über das Argument `batch_size` gesetzt werden, das standardmäßig

auf 32 steht). Am Ende dieses Kapitels werden wir uns nochmals mit dem Anpassen der Hyperparameter befassen. Sind Sie dann mit der Validierungsgenauigkeit Ihres Modells zufrieden, sollten Sie es auf den Testdatensatz loslassen, um den Generalisierungsfehler zu schätzen, bevor Sie das Modell produktiv einsetzen. Das können Sie ganz einfach über die Methode `evaluate()` erreichen (sie unterstützt eine Reihe weiterer Argumente, wie zum Beispiel `batch_size` oder `sample_weight` – schauen Sie dafür in die Dokumentation):

```
>>> model.evaluate(X_test, y_test)

10000/10000 [=====] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851

[0.3339798209667206, 0.8851]
```

Wie wir in [Kapitel 2](#) gesehen haben, bekommt man für den Testdatensatz meist eine etwas schlechtere Performance als für den Validierungsdatensatz, weil die Hyperparameter für den Validierungsdatensatz angepasst wurden (in diesem Beispiel haben wir allerdings gar keine Anpassungen an den Hyperparametern vorgenommen, daher ist die niedrigere Genauigkeit reiner Zufall). Widerstehen Sie der Versuchung, die Hyperparameter für den Testdatensatz anzupassen, denn sonst wird Ihre Schätzung des Generalisierungsfehlers zu optimistisch ausfallen.

Mit dem Modell Vorhersagen treffen

Als Nächstes können wir die Methode `predict()` des Modells verwenden, um Vorhersagen für neue Instanzen zu machen. Da wir gar keine echten neuen Instanzen haben, werden wir einfach die ersten drei Instanzen des Testdatensatzes verwenden:

```
>>> X_new = X_test[:3]

>>> y_proba = model.predict(X_new)

>>> y_proba.round(2)

array([[0. , 0. , 0. , 0. , 0. , 0.03, 0. , 0.01, 0. , 0.96],
       [0. , 0. , 0.98, 0. , 0.02, 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],  
      dtype=float32)
```

Wie Sie sehen, schätzt das Modell für jede Instanz eine Wahrscheinlichkeit pro Kategorie – von Kategorie 0 bis Kategorie 9. So schätzt es beispielsweise für das erste Bild, dass die Wahrscheinlichkeit für Kategorie 9 (Ankle Boot) 96% beträgt, für Kategorie 5 (Sandal) liegt sie bei 3%, für Kategorie 7 (Sneaker) bei 1%, und für die anderen Kategorien ist sie vernachlässigbar. Mit anderen Worten: Es »glaubt«, dass das erste Bild Fußbekleidung ist – sehr wahrscheinlich Ankle Boots, aber möglicherweise auch Sandals oder Sneakers. Ist Ihnen nur die Kategorie mit der höchsten geschätzten Wahrscheinlichkeit wichtig (auch wenn diese ziemlich niedrig ist), können Sie stattdessen die Methode `predict_classes()` verwenden:

```
>>> y_pred = model.predict_classes(X_new)  
>>> y_pred  
array([9, 2, 1])  
>>> np.array(class_names)[y_pred]  
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

Hier hat der Klassifikator alle drei Bilder sogar korrekt klassifiziert (Sie sehen das in Abbildung 10-13):

```
>>> y_new = y_test[:3]  
>>> y_new  
array([9, 2, 1])
```

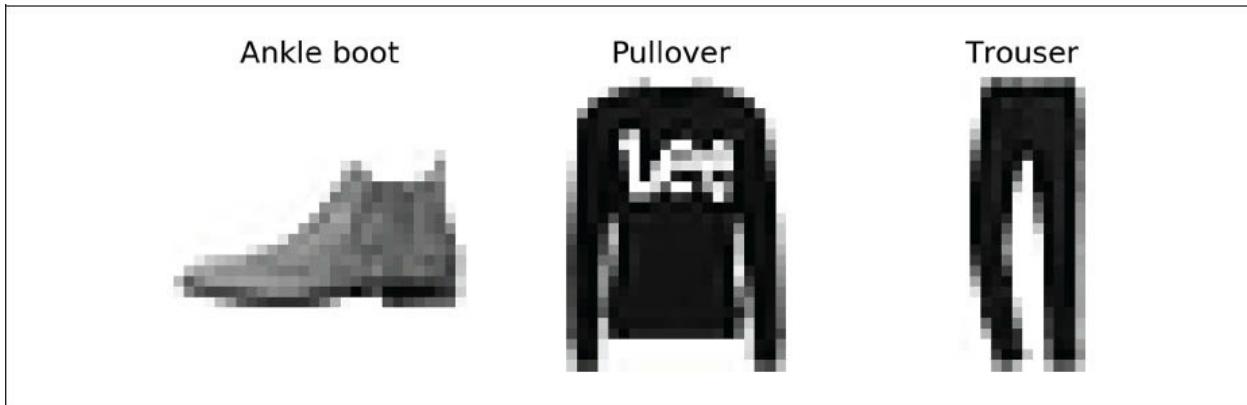


Abbildung 10-13: Korrekt klassifizierte Fashion-MNIST-Bilder

Jetzt wissen Sie, wie Sie mit der Sequential API ein Klassifikations-MLP bauen, trainieren, auswerten und verwenden. Aber wie Sie es mit Regression aus?

Ein Regressions-MLP mit der Sequential API erstellen

Kehren wir zu den kalifornischen Immobilienpreisen zurück und gehen wir das Problem über ein Regressions-KNN an. Aus Gründen der Einfachheit werden wir die Scikit-Learn-Funktion `fetch_california_housing()` zum Laden der Daten nutzen. Dieser Datensatz ist einfacher als der in [Kapitel 2](#) verwendete, da er nur numerische Merkmale enthält (es gibt kein Merkmal `ocean_proximity`), und es gibt keine fehlenden Werte. Nach dem Laden der Daten teilen wir sie in einen Trainings-, einen Validierungs- und einen Testdatensatz auf und skalieren alle Merkmale:

```
from sklearn.datasets import fetch_california_housing

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target)

X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full)

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_valid = scaler.transform(X_valid)

X_test = scaler.transform(X_test)
```

Mit der Sequential API bauen, trainieren, evaluieren und verwenden wir ein Regressions-MLP für Vorhersagen so ähnlich wie bei einer Klassifikation. Der Hauptunterschied liegt darin, dass die Ausgabeschicht aus nur einem einzelnen Neuron besteht (da wir nur einen einzelnen Wert vorhersagen wollen), dort keine Aktivierungsfunktion verwendet wird und für die Verlustfunktion der mittlere quadratische Fehler zum Einsatz kommt. Da der Datensatz ziemlich verrauscht ist, verwenden wir nur eine einzelne verborgene Schicht mit weniger Neuronen als zuvor, um ein Overfitting zu vermeiden:

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
```

```

])
model.compile(loss="mean_squared_error", optimizer="sgd")
history = model.fit(X_train, y_train, epochs=20,
                     validation_data=(X_valid, y_valid))

mse_test = model.evaluate(X_test, y_test)

X_new = X_test[:3] # vorgeben, dass es sich um neue Instanzen handelt
y_pred = model.predict(X_new)

```

Wie Sie sehen, lässt sich die Sequential API ziemlich einfach einsetzen. Aber auch wenn Sequential-Modelle sehr verbreitet sind, ist es manchmal nützlich, neuronale Netze mit komplexeren Topologien oder mit mehreren Eingaben oder Ausgaben zu bauen. Zu diesem Zweck bietet Keras die Functional API an.

Komplexe Modelle mit der Functional API bauen

Ein Beispiel für ein nicht sequenzielles neuronales Netz ist eines vom Typ *Wide & Deep*. Diese Architektur wurde in einem Artikel aus dem Jahr 2016 (<https://homl.info/widedeep>) von Heng-Tze Cheng et al.¹⁶ vorgestellt. Sie verbindet alle Eingaben oder Teile davon direkt mit der Ausgabeschicht, wie Sie in [Abbildung 10-14](#) sehen. Diese Architektur ermöglicht dem neuronalen Netz, sowohl tiefe Muster (über den tiefen Pfad) wie auch einfache Regeln zu erlernen (über den kurzen Pfad).¹⁷ Im Gegensatz dazu müssen bei einem normalen MLP alle Daten durch alle Schichten fließen – damit werden einfache Muster in den Daten eventuell durch die diversen Transformationen zerstört.

Wir bauen ein solches neuronales Netz für die kalifornischen Immobiliendaten:

```

input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])

```

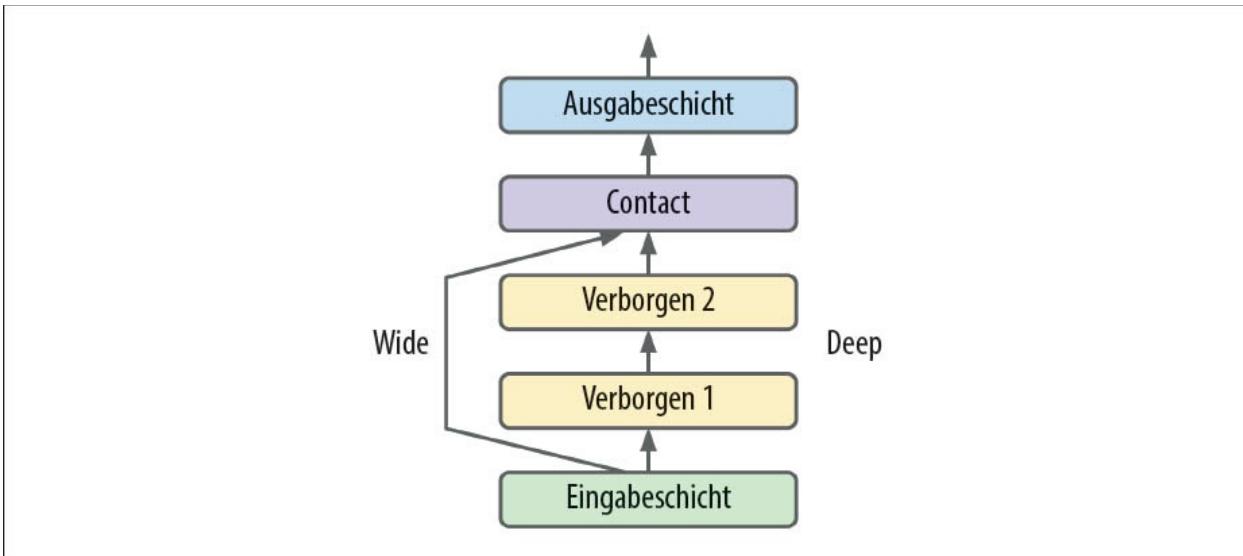


Abbildung 10-14: Neuronales Netz »Wide & Deep«

Gehen wir jede Codezeile durch:

- Zuerst müssen wir ein `Input`-Objekt erstellen.¹⁸ Dabei wird spezifiziert, was für Eingaben in das Modell gegeben werden – einschließlich `shape` und `dtype`. Wie wir noch sehen werden, kann ein Modell auch mehrere Eingaben erhalten.
- Dann erzeugen wir eine `Dense`-Schicht mit 30 Neuronen, die die ReLU-Aktivierungsfunktion verwendet. Sobald sie erstellt ist, können wir sie wie eine Funktion verwenden und ihr die Eingaben übergeben. Das ist der Grund dafür, dass diese API als Functional API bezeichnet wird. Dabei sagen wir Keras so nur, wie die Schichten miteinander verbunden werden sollen – es werden noch keine Daten übertragen.
- Nun erstellen wir eine zweite verborgene Schicht und verwenden sie wieder wie eine Funktion. Diesmal übergeben wir die Ausgabe der ersten verborgenen Schicht.
- Als Nächstes erzeugen wir eine `Concatenate`-Schicht und nutzen sie auch wieder wie eine Funktion, um die Eingabeschicht und die zweite verborgene Schicht miteinander zu verbinden. Sie können auch die Funktion `keras.layers.concatenate()` verwenden, die eine `Concatenate`-Schicht erzeugt und diese direkt mit den übergebenen Eingaben aufruft.
- Dann erstellen wir die Ausgabeschicht, die aus einem einzelnen Neuron ohne Aktivierungsfunktion besteht, und rufen sie wie eine Funktion auf, wobei wir das Ergebnis der vorherigen Verkettung übergeben.
- Zum Schluss erzeugen wir ein Keras-Modell und geben an, welche Eingaben und Ausgaben existieren.

Haben Sie das Keras-Modell gebaut, ist alles genauso wie vorher, daher müssen wir es hier nicht wiederholen: Sie müssen das Modell kompilieren, trainieren, auswerten und dann für die Vorhersage verwenden.

Aber was ist, wenn Sie eine Untermenge der Merkmale über den Wide-Pfad und eine andere

Untermenge (die sich eventuell mit der ersten überlappt) über den Deep-Pfad schicken wollen (siehe [Abbildung 10-15](#))? Dann gibt es die Möglichkeit, mehrere Eingaben zu verwenden. Stellen Sie sich beispielsweise vor, dass wir fünf Merkmale über den Wide-Pfad schicken wollen (Merkmale 0 bis 4) und sechs Merkmale über den Deep-Pfad (Merkmale 2 bis 7):

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="output")(concat)
model = keras.Model(inputs=[input_A, input_B], outputs=[output])
```

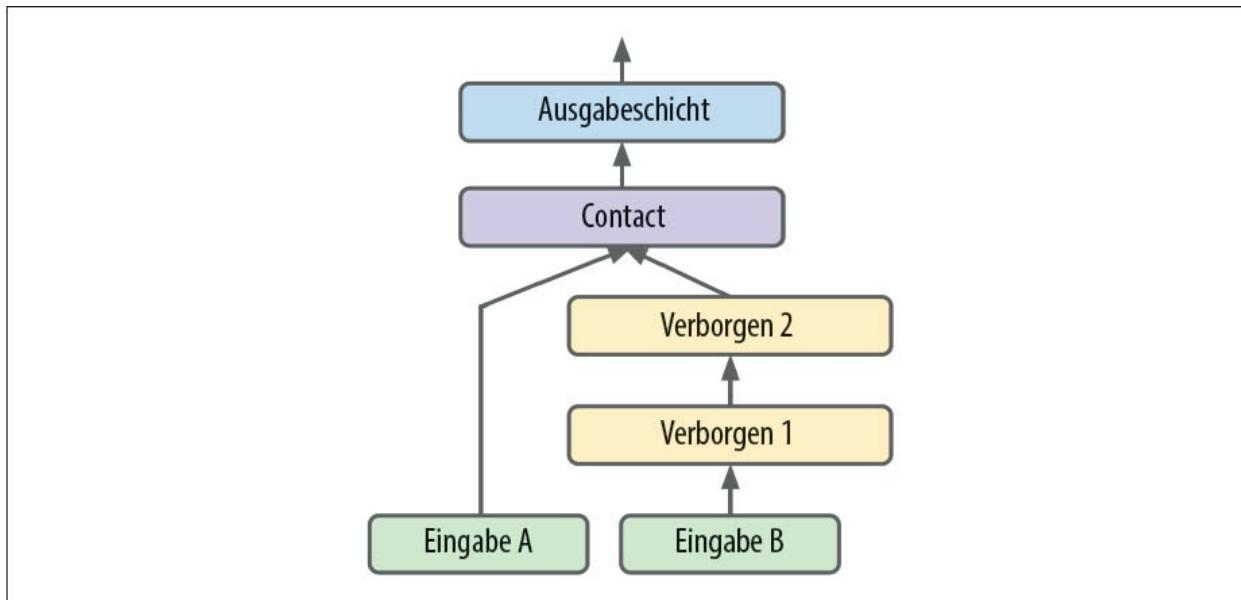


Abbildung 10-15: Verwenden mehrerer Eingaben

Der Code ist selbsterklärend. Sie sollten zumindest die wichtigsten Schichten mit Namen versehen, insbesondere, wenn das Modell komplexer wird. Beachten Sie, dass wir beim Erstellen des Modells `inputs=[input_A, input_B]` angegeben haben. Jetzt können wir das Modell wie üblich kompilieren, aber wenn wir die Methode `fit()` aufrufen, können wir nicht eine einzelne Eingangsmatrix `X_train` übergeben, sondern wir müssen zwei Matrizen (`X_train_A, X_train_B`) übergeben – eine pro Eingang.¹⁹ Das Gleiche gilt für `X_valid` und auch für `X_test` und `X_new`, wenn Sie `evaluate()` oder `predict()` aufrufen:

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
```

```

X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]

X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]

X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]

X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                     validation_data=((X_valid_A, X_valid_B), y_valid))

mse_test = model.evaluate((X_test_A, X_test_B), y_test)

y_pred = model.predict((X_new_A, X_new_B))

```

Es gibt viele Situationen, in denen Sie mehrere Ausgaben haben wollen:

- Die Aufgabe erfordert es. Vielleicht wollen Sie das Hauptobjekt in einem Bild finden und klassifizieren. Dabei handelt es sich um eine Regressionsaufgabe (die Koordinaten des Objektmittelpunkts sowie seine Breite und Höhe finden) und gleichzeitig um eine Klassifikationsaufgabe.
- Genauso können Sie mehrere unabhängige Aufgaben für die gleichen Daten haben. Sicher – Sie könnten ein neuronales Netz pro Aufgabe trainieren, aber in vielen Fällen werden Sie bessere Ergebnisse für alle Aufgaben erhalten, wenn Sie ein einzelnes neuronales Netz mit einer Ausgabe pro Aufgabe trainieren. Das liegt daran, dass das neuronale Netz Merkmale in den Daten lernen kann, die für mehrere Aufgaben nützlich sind. So könnten Sie beispielsweise auf Bilder eine *Multitask-Klassifikation* für Gesichter durchführen und dabei eine Ausgabe zum Klassifizieren des Gesichtsausdrucks nutzen (Lächeln, Überraschung und so weiter) und eine weitere, um herauszufinden, ob jemand eine Brille trägt oder nicht.
- Ein weiterer Anwendungsfall ist der Einsatz als Regularisierungstechnik (also eine Trainingsbeschränkung, deren Ziel es ist, Overfitting zu vermeiden und damit die Fähigkeit des Modells zum Generalisieren zu verbessern). So können Sie beispielsweise zusätzliche unterstützende Ausgaben in die Architektur eines neuronalen Netzes einbauen (siehe [Abbildung 10-16](#)), um sicherzustellen, dass der zugrunde liegende Teil des Netzes für sich etwas Nützliches lernt, ohne vom Rest des Netzes abhängig zu sein.

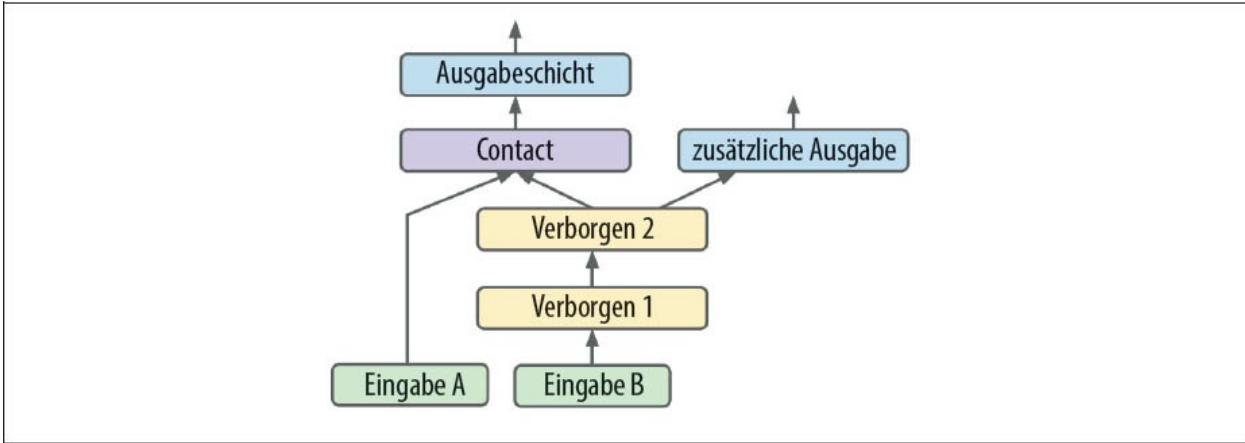


Abbildung 10-16: Mehrere Ausgaben, in diesem Beispiel für eine unterstützende Ausgabe zur Regularisierung

Es ist ziemlich einfach, zusätzliche Ausgaben hinzuzufügen: Verbinden Sie sie mit den passenden Schichten und ergänzen Sie sie in der Liste mit Ausgaben Ihres Modells. So baut beispielsweise der folgende Code das Netz aus Abbildung 10-16:

```
[...] # Wie zuvor bis zur Hauptausgabeschicht
output = keras.layers.Dense(1, name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.Model(inputs=[input_A, input_B], outputs=[output, aux_output])
```

Jede Ausgabe benötigt ihre eigene Verlustfunktion. Daher sollten wir beim Kompilieren des Modells eine Liste mit Verlustfunktionen übergeben²⁰ (übergeben wir eine einzelne Verlustfunktion, wird Keras davon ausgehen, dass sie für alle Ausgaben genutzt werden soll). Standardmäßig berechnet Keras all diese Verluste und addiert sie dann, um den Gesamtverlust für das Training zu erhalten. Uns ist die Hauptausgabe deutlich wichtiger als die zusätzliche Ausgabe (da diese nur der Regulierung dient), daher wollen wir dem Verlust der Hauptausgabe ein viel größeres Gewicht verpassen. Zum Glück ist es möglich, alle Verlustgewichte beim Kompilieren des Modells zu übergeben:

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1], optimizer="sgd")
```

Trainieren wir jetzt das Modell, müssen wir Labels für jede Ausgabe bereitstellen. In diesem Beispiel sollten die Hauptausgabe und die zusätzliche Ausgabe versuchen, das Gleiche vorherzusagen, daher sollten sie auch die gleichen Labels verwenden. Anstatt also `y_train` zu übergeben, müssen wir `(y_train, y_train)` verwenden (das Gleiche gilt für `y_valid` und `y_test`):

```
history = model.fit(
```

```
[X_train_A, X_train_B], [y_train, y_train], epochs=20,
validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))
```

Evaluieren wir das Modell, wird Keras den Gesamtverlust und die einzelnen Verluste zurückgeben:

```
total_loss, main_loss, aux_loss = model.evaluate(
    [X_test_A, X_test_B], [y_test, y_test])
```

Genauso liefert die Methode `predict()` die Vorhersagen für jede Ausgabe:

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

Wie Sie sehen, können Sie mit der Functional API recht einfach beliebige Architekturen bauen. Schauen wir uns noch eine weitere Möglichkeit an, wie sich Keras-Modelle erzeugen lassen.

Dynamische Modelle mit der Subclassing API bauen

Sowohl die Sequential API wie auch die Functional API sind deklarativ: Sie beginnen mit dem Deklarieren, welche Schichten Sie verwenden und wie diese verbunden sein sollen – erst dann können Sie das Modell mit Daten zum Trainieren und Vorhersagen füttern. Das hat viele Vorteile: Das Modell kann einfach gesichert, dupliziert und geteilt werden, seine Struktur lässt sich anzeigen und analysieren, und das Framework kann Formen ableiten und Typen prüfen, um Fehler frühzeitig abzufangen (also bevor überhaupt Daten durch das Modell laufen). Auch das Debugging ist recht einfach, da es sich beim gesamten Modell um einen statischen Graphen aus Schichten handelt. Aber genau das ist auch der Nachteil: Das Modell ist statisch. Zu manchen Modellen gehören Schleifen, unterschiedliche Formen, bedingte Verzweigungen und anderes dynamisches Verhalten. Für solche Fälle – oder auch nur, wenn Sie einen imperativeren Programmierstil bevorzugen – ist die Subclassing API für Sie da.

Bilden Sie einfach Unterklassen der Klasse `Model`, erzeugen Sie die benötigten Schichten im Konstruktor und nutzen Sie diese für die gewünschten Berechnungen in der Methode `call()`. Erzeugen Sie beispielsweise eine Instanz der folgenden Klasse `WideAndDeepModel`, erhalten Sie ein Modell, das dem oben mit der Functional API erzeugten entspricht. Sie können es dann kompilieren, auswerten und für Vorhersagen verwenden – so wie zuvor:

```
class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # handles standard args (e.g., name)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
```

```

        self.main_output = keras.layers.Dense(1)

        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):

        input_A, input_B = inputs

        hidden1 = self.hidden1(input_B)

        hidden2 = self.hidden2(hidden1)

        concat = keras.layers.concatenate([input_A, hidden2])

        main_output = self.main_output(concat)

        aux_output = self.aux_output(hidden2)

        return main_output, aux_output

model = WideAndDeepModel()

```

Das Modell gleicht sehr dem der Functional API, nur dass wir die Eingaben nicht erzeugen müssen – wir verwenden einfach das Argument `input` der Methode `call()` und trennen das Erstellen der Schichten²¹ im Konstruktor von deren Einsatz in der Methode `call()`. Der große Unterschied liegt darin, dass Sie in `call()` so gut wie alles tun können: `for`-Schleifen, `if`-Anweisungen, TensorFlow-Operationen unter der Motorhaube – lassen Sie Ihrer Fantasie freien Lauf (siehe [Kapitel 12](#))! Damit erhalten Forscher, die mit neuen Ideen experimentieren, eine großartige API.

Diese zusätzliche Flexibilität gibt es aber nicht umsonst – die Architektur Ihres Modells ist in der Methode `call()` verborgen, daher kann Keras sie nicht einfach inspizieren, sichern oder duplizieren. Und wenn Sie die Methode `summary()` aufrufen, erhalten Sie nur eine Liste mit Schichten ohne zusätzliche Informationen darüber, wie sie miteinander verbunden sind. Zudem kann Keras keine Typen und Formen im Voraus prüfen, und Fehler können so schneller geschehen. Sofern Sie also diese zusätzliche Flexibilität nicht benötigen, sollten Sie wohl besser bei der Sequential API oder der Functional API bleiben.



Keras-Modelle können wie normale Schichten verwendet werden, daher können Sie sie problemlos kombinieren, um komplexe Architekturen zu erstellen.

Nachdem Sie jetzt wissen, wie Sie neuronale Netze mit Keras aufbauen und trainieren, werden Sie sie auch sichern wollen!

Ein Modell sichern und wiederherstellen

Beim Einsatz der Sequential API oder der Functional API ist das Sichern eines trainierten Keras-Modells wirklich einfach:

```
model = keras.models.Sequential([...]) # oder keras.Model(...)

model.compile(...)

model.fit(...)

model.save("my_keras_model.h5")
```

Keras nutzt das HDF5-Format, um sowohl die Modellarchitektur (einschließlich der Hyperparameter aller Schichten) wie auch die Werte aller Modellparameter für jede Schicht (zum Beispiel Verbindungsgewichte und Biase) zu sichern. Zudem speichert es den Optimizer ab (einschließlich dessen Hyperparameter und jedes Status, den er eventuell besitzt).

Sie werden meist ein Skript nutzen, das ein Modell trainiert und abspeichert, und eines oder mehrere Skripten (oder Webservices), die das Modell laden und zum Vorhersagen einsetzen. Das Laden des Modells geschieht genauso einfach:

```
model = keras.models.load_model("my_keras_model.h5")
```



Das funktioniert mit der Sequential API oder der Functional API, aber leider nicht beim Subclassing von Modellen. Sie können `save_weights()` und `load_weights()` verwenden, um zumindest die Modellparameter zu sichern und wieder zu laden, aber für alles andere sind Sie selbst verantwortlich.

Aber was ist, wenn das Trainieren viele Stunden dauert? Das ist nicht ungewöhnlich, insbesondere beim Trainieren mit großen Datensätzen. In diesem Fall sollten Sie nicht nur Ihr Modell am Ende des Trainings sichern, sondern auch schon während des Trainings in regelmäßigen Abständen Checkpoints sichern, um zu verhindern, dass Sie alles verlieren, wenn Ihr Computer abstürzt. Wie können Sie aber der Methode `fit()` mitteilen, dass sie Checkpoints sichern soll? Dafür verwenden Sie Callbacks.

Callbacks

Die Methode `fit()` kann ein Argument `callbacks` übernehmen, mit dem Sie eine Liste von Objekten angeben können, die Keras am Anfang und am Ende des Trainings, nach jeder Epoche und sogar vor und nach dem Verarbeiten jedes Batchs aufruft. So sichert beispielsweise der Callback `ModelCheckpoint` Checkpoints Ihres Modells in regelmäßigen Abständen während des Trainings – standardmäßig am Ende jeder Epoche:

```
[...] # das Modell bauen und kompilieren

checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")

history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

Verwenden Sie eine Validierung während des Trainings, können Sie beim Erstellen des

`ModelCheckpoint` zusätzlich `save_best_only=True` setzen. Dadurch wird Ihr Modell nur dann gesichert, wenn seine Leistung für den Validierungsdatensatz die bisher beste ist. So müssen Sie sich nicht darum sorgen, dass Ihr Training zu lange dauert und es zu einem Overfitting an dem Trainingsdatensatz kommt: Stellen Sie nach dem Training einfach das letzte gesicherte Modell wieder her, erhalten Sie das für den Validierungsdatensatz beste Modell. Der folgende Code zeigt einen einfachen Weg auf, ein Early Stopping zu implementieren (siehe [Kapitel 4](#)):

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)

history = model.fit(X_train, y_train, epochs=10,
                     validation_data=(X_valid, y_valid),
                     callbacks=[checkpoint_cb])

model = keras.models.load_model("my_keras_model.h5") # zum besten Modell zurück
```

Eine andere Möglichkeit, ein Early Stopping zu implementieren, ist einfach der Einsatz des Callbacks `EarlyStopping`. Er unterbricht das Training, wenn er eine gewisse Anzahl von Epochen lang keinen Fortschritt mit dem Validierungsdatensatz mehr messen kann (definiert durch das Argument `patience`) und kehrt dann optional zum besten Modell zurück. Sie können beide Callbacks kombinieren, um Checkpoints Ihres Modells zu sichern (falls Ihr Computer abstürzt) und das Training abzubrechen, wenn es keinen Fortschritt mehr gibt (um nicht mehr Zeit und Ressourcen zu verschwenden):

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                   restore_best_weights=True)

history = model.fit(X_train, y_train, epochs=100,
                     validation_data=(X_valid, y_valid),
                     callbacks=[checkpoint_cb, early_stopping_cb])
```

Die Anzahl der Epochen kann auf einen hohen Wert gesetzt werden, da das Training automatisch endet, wenn es keinen Fortschritt mehr gibt. In diesem Fall muss das beste Modell nicht wieder zurückgeholt werden, weil sich der Callback `Early Stopping` die besten Gewichte automatisch merkt und sie am Ende des Trainings für Sie wiederherstellt.



Im Paket `keras.callbacks` (<https://keras.io/callbacks/>) finden Sie eine ganze Reihe weiterer Callbacks.

Brauchen Sie mehr Kontrollmöglichkeiten, können Sie sich Ihre eigenen Callbacks schreiben. Als Beispiel dafür zeigt der folgende Callback das Verhältnis zwischen Validierungsverlust und Trainingsverlust während des Trainings an (um zum Beispiel ein Overfitting zu erkennen):

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):  
  
    def on_epoch_end(self, epoch, logs):  
  
        print("\nval/train: {:.2f}".format(logs["val_loss"] / logs["loss"]))
```

Wie zu erwarten, können Sie `on_train_begin()`, `on_train_end()`, `on_epoch_begin()`, `on_epoch_end()`, `on_batch_begin()` und `on_batch_end()` implementieren. Callbacks können auch während der Evaluierung und beim Voraussagen genutzt werden, wenn sie dort benötigt werden (beispielsweise zum Debuggen). Beim Evaluieren sollten Sie `on_test_begin()`, `on_test_end()`, `on_test_batch_begin()` oder `on_test_batch_end()` implementieren (aufgerufen durch `evaluate()`), bei der Voraussage `on_predict_begin()`, `on_predict_end()`, `on_predict_batch_begin()` oder `on_predict_batch_end()` (aufgerufen durch `predict()`).

Schauen wir uns nun noch ein weiteres Werkzeug an, das Sie bei der Arbeit mit tf. keras auf jeden Fall nutzen sollten: TensorBoard.

TensorBoard zur Visualisierung verwenden

TensorBoard ist ein ausgezeichnetes interaktives Visualisierungswerkzeug, mit dem Sie die Lernkurven während des Trainings betrachten, Lernkurven zwischen mehreren Läufen vergleichen, den Rechengraphen darstellen, Trainingsstatistiken analysieren, durch Ihr Modell erzeugte Bilder anzeigen, komplexe mehrdimensionale Daten auf drei Dimensionen herunterprojiziert und für Sie automatisch geclustert ausgeben lassen können und noch vieles mehr. Dieses Tool wird automatisch installiert, wenn Sie TensorFlow installieren, daher haben Sie es schon an Bord.

Um es zu verwenden, müssen Sie Ihr Programm so anpassen, dass es die zu visualisierenden Daten in speziellen binären Logdateien namens *Event Files* ausgibt. Jeder binäre Datensatz wird als *Summary* bezeichnet. Der TensorBoard-Server überwacht das Log-Verzeichnis, holt sich automatisch geänderte Daten und aktualisiert die Visualisierungen: So können Sie Live-Daten darstellen (mit einer kleinen Verzögerung), zum Beispiel Lernkurven während des Trainings. Meist lassen Sie den TensorBoard-Server auf ein Root-Log-Verzeichnis zeigen und konfigurieren Ihr Programm so, dass es bei jedem Lauf in ein anderes Unterverzeichnis schreibt. So kann die gleiche TensorBoard-Server-Instanz Daten aus mehreren Läufen Ihres Programms visualisieren und vergleichen, ohne alles durcheinanderzubringen.

Beginnen wir damit, das Root-Log-Verzeichnis zu definieren, das wir für unsere TensorBoard-Logs verwenden. Zusätzlich schreiben wir noch eine kleine Funktion, die ausgehend vom aktuellen Datum und der Uhrzeit einen Unterverzeichnispfad erzeugt, sodass dieser bei jedem Lauf anders ist. Sie können in den Namen des Log-Verzeichnisses noch zusätzliche Informationen mit aufnehmen, wie zum Beispiel Werte von Hyperparametern, die Sie testen, um

dann in TensorBoard einfacher zu erkennen, was Sie sich da gerade anschauen:

```
import os

root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():

    import time

    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir() # z. B. './my_logs/run_2019_06_07-15_15_22'
```

Netterweise bietet Keras einen Callback TensorBoard() an:

```
[...] # Das Modell bauen und kompilieren

tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)

history = model.fit(X_train, y_train, epochs=30,
                     validation_data=(X_valid, y_valid),
                     callbacks=[tensorboard_cb])
```

Und das ist alles, was Sie zu tun haben! Sehr viel einfacher kann es eigentlich nicht mehr werden. Führen Sie diesen Code aus, kümmert sich der TensorBoard()-Callback darum, das Log-Verzeichnis für Sie anzulegen (zusammen mit seinen übergeordneten Verzeichnissen, wenn das erforderlich ist), und während des Trainings erzeugt es die Event Files und schreibt die Summaries dort hinein. Nachdem das Programm ein zweites Mal lief (vielleicht, weil ein paar Werte von Hyperparametern angepasst wurden), erhalten Sie eine Verzeichnisstruktur wie die folgende:

```
my_logs/
|— run_2019_06_07-15_15_22
|   |— train
|   |   |— events.out.tfevents.1559891732.mycomputer.local.38511.694049.v2
|   |   |— events.out.tfevents.1559891732.mycomputer.local.profile-empty
|   |   |— plugins/profile/2019-06-07_15-15-32
|   |       |— local.trace
```

```
|   └── validation  
|       └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2  
└── run_2019_06_07-15_15_49  
    └── [...]
```

Es gibt ein Verzeichnis pro Durchlauf mit jeweils einem Unterverzeichnis für die Trainingslogs und die Validierungslogs. Beide enthalten Event Files, aber in den Trainingslogs finden sich auch Profiling Traces: Damit kann Ihnen TensorBoard genau zeigen, wie viel Zeit das Modell in jedem seiner Teile und auf all seinen Devices verbracht hat, was zum Finden von Performanceflaschenhälzen sehr hilfreich ist.

Als Nächstes müssen Sie den TensorBoard-Server starten. Das können Sie durch einen Befehl in einem Terminal erreichen. Installieren Sie TensorFlow innerhalb einer virtualenv, sollten Sie es aktivieren. Dann führen Sie den folgenden Befehl im Root-Verzeichnis Ihres Projekts aus (oder von einem anderen Ort, solange Sie auf das richtige Log-Verzeichnis verweisen):

```
$ tensorboard --logdir=./my_logs --port=6006
```

```
TensorBoard 2.0.0 at http://mycomputer.local:6006/ (Press CTRL+C to quit)
```

Kann Ihre Shell das Skript *tensorboard* nicht finden, müssen Sie Ihre Umgebungsvariable PATH so erweitern, dass sie das Verzeichnis mit dem Skript enthält (oder Sie ersetzen *tensorboard* in der Befehlszeile durch `python3 -m tensorboard.main`). Läuft der Server, können Sie einen Webbrower öffnen und <http://localhost:6006> aufrufen.

Stattdessen können Sie TensorBoard auch direkt innerhalb von Jupyter verwenden, indem Sie die folgenden Befehle ausführen. Die erste Zeile lädt die TensorBoard-Erweiterung, die zweite startet einen TensorBoard-Server an Port 6006 (sofern er nicht schon gestartet wurde) und verbindet sich mit ihm:

```
%load_ext tensorboard  
%tensorboard --logdir=./my_logs --port=6006
```

Nun sollten Sie die Weboberfläche von TensorBoard sehen. Klicken Sie auf den Tab **SCALARS**, um die Lernkurven anzuzeigen (siehe [Abbildung 10-17](#)). Unten links wählen Sie die gewünschten Logs aus (zum Beispiel die Trainingslogs des ersten und zweiten Laufs) und klicken auf den Skalar `epoch_loss`. Sie sehen, wie der Trainingsverlust in beiden Läufen geringer wurde, aber beim zweiten Mal ging es viel schneller. Und tatsächlich haben wir eine Lernrate von 0,05 (`optimizer=keras.optimizers.SGD(lr=0.05)`) statt 0,001 genutzt.

Sie können auch den gesamten Graphen, die erlernten Gewichte (projiziert in 3-D) oder die Profiling Traces darstellen. Der Callback `TensorBoard()` bietet zudem Optionen zum Protokollieren zusätzlicher Daten, wie zum Beispiel Embeddings (siehe [Kapitel 13](#)).

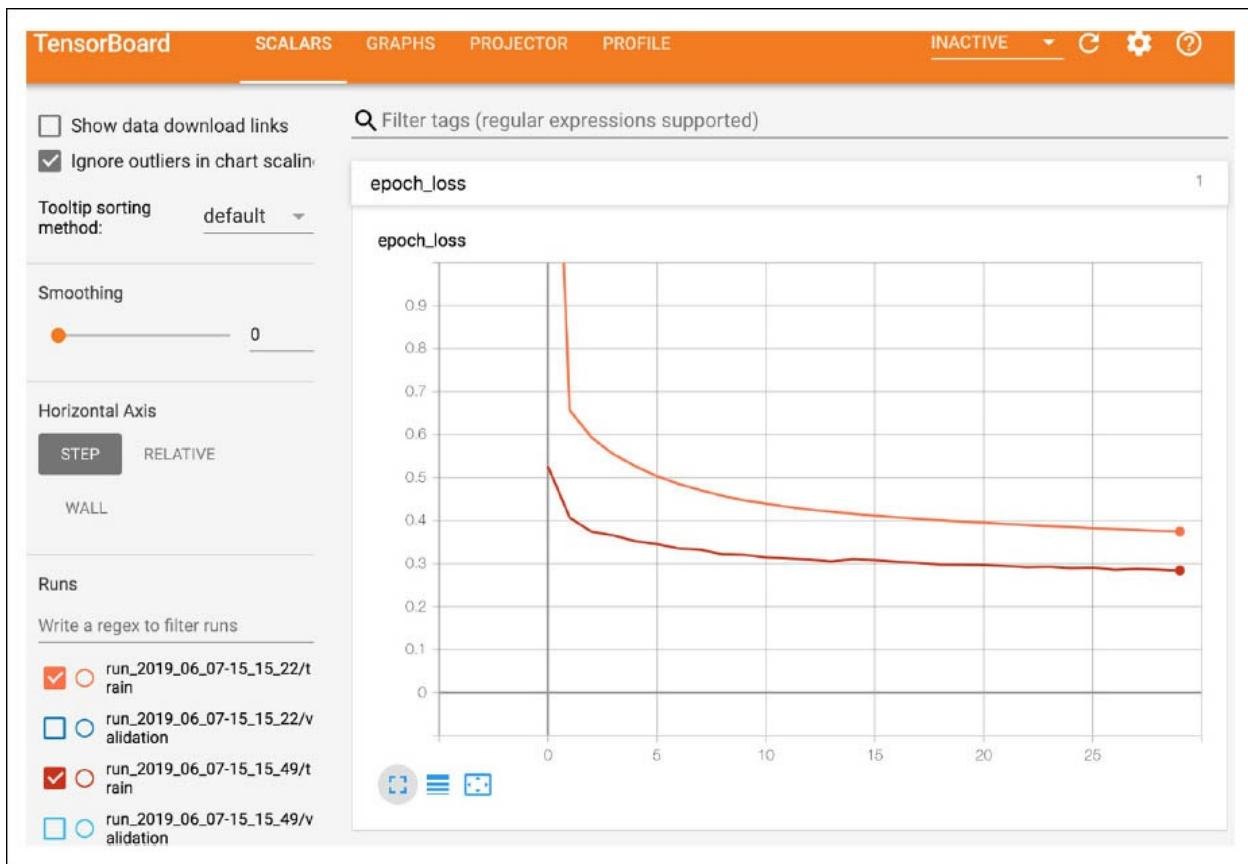


Abbildung 10-17: Visualisieren von Lernkurven mit TensorBoard

Außerdem enthält TensorFlow im Paket `tf.summary` eine API auf niedrigerer Ebene. Der folgende Code erzeugt einen `SummaryWriter` mithilfe der Funktion `create_file_writer()` und nutzt diesen als Kontext, um Skalare, Histogramme, Bilder, Audio und Text zu protokollieren, was sich dann alles mithilfe von TensorBoard darstellen lässt (probieren Sie es einmal aus!):

```
test_logdir = get_run_logdir()

writer = tf.summary.create_file_writer(test_logdir)

with writer.as_default():

    for step in range(1, 1000 + 1):

        tf.summary.scalar("my_scalar", np.sin(step / 10), step=step)

        data = (np.random.randn(100) + 2) * step / 100 # Zufallsdaten

        tf.summary.histogram("my_hist", data, buckets=50, step=step)

        images = np.random.rand(2, 32, 32, 3) # random 32x32 RGB images

        tf.summary.image("my_images", images * step / 1000, step=step)
```

```

texts = ["Schritt: " + str(step), " Quadrat: " + str(step**2)]

tf.summary.text("my_text", texts, step=step)

sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)

audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])

tf.summary.audio("my_audio", audio, sample_rate=48000, step=step)

```

Das ist ein sehr nützliches Visualisierungstool, sogar über TensorFlow oder Deep Learning hinaus.

Fassen wir zusammen, was Sie in diesem Kapitel bisher gelernt haben: Sie haben gesehen, woher neuronale Netze stammen, was ein MLP ist und wie Sie es zur Klassifikation und Regression verwenden können, wie Sie die Sequential API von `tf.keras` zum Bauen von MLPs einsetzen und wie die Functional API und die Subclassing API für das Bauen komplexerer Modellarchitekturen zum Einsatz kommen. Sie haben gelernt, wie Sie ein Modell sichern und wiederherstellen und wie Sie mit Callbacks Checkpoints, Early Stopping und anderes verwirklichen. Und schließlich haben Sie erfahren, wie Sie mit TensorBoard Visualisierungen ermöglichen. Damit können Sie jetzt schon viele Probleme mithilfe neuronaler Netze angehen. Aber Sie fragen sich vielleicht, wie Sie die richtige Anzahl an verborgenen Schichten, die Anzahl an Neuronen im Netz und all die anderen Hyperparameter auswählen. Schauen wir uns das jetzt an.

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 Netzwerkarchitektur 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?

Eine Option ist, einfach viele Kombinationen aus Hyperparametern auszuprobieren und zu sehen, welche davon am besten für den Validierungsdatensatz funktionieren (oder eine k-fache Kreuzvalidierung zu verwenden). Wir können beispielsweise `GridSearchCV` oder `RandomizedSearchCV` verwenden, um den Raum der Hyperparameter zu durchforsten, so wie wir das in [Kapitel 2](#) getan haben. Dazu müssen wir unser Keras-Modell in Objekten verpacken, die normale Scikit-Learn-Regressoren simulieren. Als ersten Schritt erstellen wir eine Funktion, die bei einem gegebenen Satz von Hyperparametern ein Keras-Modell baut und kompiliert:

```

def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):

    model = keras.models.Sequential()

    model.add(keras.layers.InputLayer(input_shape=input_shape))

```

```

for layer in range(n_hidden):
    model.add(keras.layers.Dense(n_neurons, activation="relu"))

model.add(keras.layers.Dense(1))

optimizer = keras.optimizers.SGD(lr=learning_rate)

model.compile(loss="mse", optimizer=optimizer)

return model

```

Diese Funktion erzeugt aus der Eingabeform und der Anzahl an verborgenen Schichten und Neuronen ein einfaches Sequential-Modell für die univariate Regression (nur ein Ausgabeneuron) und kompiliert es mit einem SGD-Optimierer, der mit der angegebenen Lernrate konfiguriert ist. Es ist gute Praxis, vernünftige Standardwerte für so viele Hyperparameter wie möglich mitzugeben – so wie es Scikit-Learn auch tut.

Als Nächstes erzeugen wir einen KerasRegressor, der auf dieser Funktion `build_model()` basiert:

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

Das Objekt vom Typ `KerasRegressor` ist eine dünne Hülle um das Keras-Modell, das mit `build_model()` gebaut wurde. Da wir beim Erstellen keine Hyperparameter angegeben haben, werden die Standardwerte genutzt, die wir in `build_model()` definiert haben. Jetzt können wir dieses Objekt wie einen normalen Regressor in Scikit-Learn nutzen: Wir können ihn mit dessen Methode `fit()` trainieren, ihn dann mit seiner Methode `score()` auswerten und über `predict()` Vorhersagen treffen:

```

keras_reg.fit(X_train, y_train, epochs=100,
               validation_data=(X_valid, y_valid),
               callbacks=[keras.callbacks.EarlyStopping(patience=10)])

mse_test = keras_reg.score(X_test, y_test)

y_pred = keras_reg.predict(X_new)

```

Beachten Sie, dass alle zusätzlichen Parameter, die Sie an die Methode `fit()` übergeben, auch an das zugrunde liegende Keras-Modell weitergereicht werden. Und der Score ist das Gegenteil der MSE, weil Scikit-Learn Scores statt Verluste haben möchte (höher ist also besser).

Wir wollen aber nicht nur solch ein einzelnes Modell trainieren und auswerten, sondern Hunderte von Varianten durchprobieren, um herauszufinden, welche mit dem Validierungsdatensatz am besten funktioniert. Da es viele Hyperparameter gibt, ist eine zufällige Suche besser als eine Gittersuche (wie in [Kapitel 2](#) besprochen). Versuchen wir, die Anzahl an verborgenen Schichten, die Menge an Neuronen und die Lernrate zu ermitteln:

```

from scipy.stats import reciprocal

from sklearn.model_selection import RandomizedSearchCV

param_distrib = {

    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distrib, n_iter=10, cv=3)

rnd_search_cv.fit(X_train, y_train, epochs=100,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])

```

Das entspricht dem, was wir in [Kapitel 2](#) gemacht haben, nur dass wir hier zusätzliche Parameter an die Methode `fit()` übergeben und diese an das zugrunde liegende Keras-Modell weitergegeben werden. Beachten Sie, dass `RandomizedSearchCV` eine k-fache Kreuzvalidierung verwendet, daher greift es nicht auf `X_valid` und `y_valid` zurück, die nur für ein Early Stopping genutzt werden.

Die Untersuchung kann abhängig von der Hardware, der Größe des Datensatzes, der Modellkomplexität und der Werte von `n_iter` und `cv` viele Stunden dauern. Ist sie abgeschlossen, können Sie auf die gefundenen besten Parameter, den besten Score und das trainierte Keras-Modell wie folgt zugreifen:

```

>>> rnd_search_cv.best_params_
{'learning_rate': 0.0033625641252688094, 'n_hidden': 2, 'n_neurons': 42}
>>> rnd_search_cv.best_score_
-0.3189529188278931
>>> model = rnd_search_cv.best_estimator_.model

```

Sie können nun dieses Modell sichern, es mit dem Testdatensatz evaluieren und – wenn Sie mit dessen Leistung zufrieden sind – in die Produktivumgebung deployen. Der Einsatz einer Zufallssuche ist nicht sehr schwierig, und für halbwegs einfache Probleme funktioniert sie ziemlich gut. Ist das Training aber langsam (zum Beispiel bei komplexeren Problemen mit größeren Datensätzen), wird dieses Vorgehen nur einen kleinen Bereich des Hyperparameter-

Raums durchforsten können. Sie können dieses Problem teilweise abmildern, indem Sie dem Suchprozess manuell unter die Arme greifen: Führen Sie zuerst eine schnelle Zufallssuche mit einem weiten Bereich an Hyperparameter-Werten durch, dann lassen Sie darauf eine weitere Suche mit kleineren Wertebereichen rund um die beim ersten Lauf gefundenen besten Werte folgen und so weiter. Dieses Vorgehen wird hoffentlich zu einem guten Satz von Hyperparametern führen. Aber es ist sehr zeitaufwendig, und vermutlich haben Sie eigentlich Besseres zu tun.

Zum Glück gibt es eine Reihe von Techniken, um einen Suchraum effizienter zu durchforsten als mit einer Zufallssuche. Die zentrale Idee ist einfach: Stellt sich eine Region des Raums als gut heraus, sollte sie genauer betrachtet werden. Solche Techniken kümmern sich für Sie um das »Zoomen« und führen in viel kürzerer Zeit zu viel besseren Lösungen. Hier ein paar Python-Bibliotheken, die Sie zum Optimieren der Hyperparameter nutzen können:

Hyperopt (<https://github.com/hyperopt/hyperopt>)

Eine beliebte Bibliothek zum Optimieren für alle möglichen komplexen Suchräume (einschließlich reeller Werte, wie zum Beispiel der Lernrate, und diskreter Werte, wie der Anzahl an Schichten).

Hyperas (<https://github.com/maxpumperla/hyperas>), *kopt* (<https://github.com/Avsecz/kopt>) oder *Talos* (<https://github.com/autonomio/talos>)

Nützliche Bibliotheken zum Optimieren von Hyperparametern für Keras-Modelle (die ersten beiden basieren auf Hyperopt).

Keras Tuner (<https://homl.info/kerastuner>)

Eine einfach zu verwendende Bibliothek von Google zum Optimieren von Hyperparametern für Keras-Modelle mit einem gehosteten Service zur Visualisierung und Analyse.

Scikit-Optimize (skopt) (<https://scikit-optimize.github.io/>)

Eine allgemein einsetzbare Optimierungsbibliothek. Die Klasse BayesSearchCV führt eine bayessche Optimierung durch und besitzt dabei eine Schnittstelle, die der von GridSearchCV ähnelt.

Spearmint (<https://github.com/JasperSnoek/spearmint>)

Eine Bibliothek zum bayesschen Optimieren.

Hyperband (<https://github.com/zygmuntz/hyperband>)

Eine schnelle Bibliothek zum Tunen von Hyperparametern, die auf dem aktuellen Hyperband-Artikel (<https://homl.info/hyperband>) von Lisha Li et al.²² beruht.

Sklearn-Deap (<https://github.com/rsteca/sklearn-deap>)

Eine Bibliothek zum Optimieren von Hyperparametern, die auf evolutionären Algorithmen basiert und eine GridSearchCV-ähnliche Schnittstelle besitzt.

Auch viele Firmen bieten Services zum Optimieren von Hyperparametern an. Wir werden den Hyperparameter Tuning Service (<https://homl.info/googletuning>) von Googles AI Platform in Kapitel 19 besprechen. Es gibt aber zum Beispiel auch Arimo (<https://arimo.com/>), SigOpt (<https://sigopt.com/>) oder Oscar (<http://oscar.calldesk.ai/>).

Das Optimieren von Hyperparametern ist immer noch Gegenstand aktueller Forschung, und

evolutionäre Algorithmen sind gerade wieder im Kommen. Schauen Sie sich beispielsweise den ausgezeichneten Artikel aus dem Jahr 2017 (<https://homl.info/pbt>) von DeepMind²³ an, in dem die Autoren gemeinsam eine Population an Modellen und ihre Hyperparameter optimiert haben. Google hat ebenfalls einen evolutionären Ansatz genutzt – nicht nur zum Suchen von Hyperparametern, sondern auch, um die beste Architektur eines neuronalen Netzes für das Problem zu finden. Ihre AutoML-Suite steht bereits als Cloud-Service (<https://cloud.google.com/automl/>) bereit. Vielleicht sind die Zeiten der manuell gebauten neuronalen Netze bald vorbei? Lesen Sie sich dazu auch den Google-Post (<https://homl.info/automlpost>) dazu durch. Tatsächlich wurden evolutionäre Algorithmen ziemlich erfolgreich eingesetzt, um individuelle neuronale Netze zu trainieren und damit die allgegenwärtige Gradientenmethode zu ersetzen. Schauen Sie sich beispielsweise das Posting (<https://homl.info/neuroevol>) aus dem Jahr 2017 von Uber an, in dem die Autoren ihre Technik *Deep Neuroevolution* vorstellen.

Aber trotz all dieser erstaunlichen Fortschritte und der ganzen Tools und Services ist es immer noch hilfreich, eine grobe Vorstellung davon zu haben, welche Werte für jeden Hyperparameter sinnvoll sind, sodass Sie schnell einen Prototyp bauen und den Suchraum verkleinern können. Die folgenden Abschnitte liefern Richtlinien für die Wahl der Anzahl der verborgenen Schichten und Neuronen in einem MLP und für die Wahl guter Werte einiger der wichtigsten Hyperparameter.

Anzahl verborgener Schichten

Bei vielen Aufgaben können Sie mit einer einzelnen verborgenen Schicht beginnen und passable Ergebnisse erhalten. Ein MLP mit nur einer verborgenen Schicht kann theoretisch auch die komplexesten Funktionen modellieren, genug Neuronen vorausgesetzt. Aber tiefere Netze besitzen eine weitaus höhere *Parametereffizienz* als flache: Sie können komplexe Funktionen mit exponentiell weniger Neuronen als flache Netze modellieren, wodurch sie mit der gleichen Menge an Trainingsdaten eine deutlich bessere Performance erreichen können.

Um dies nachzuvollziehen, nehmen wir einmal an, Sie sollten mit einem Zeichenprogramm einen Wald zeichnen, ohne jedoch Copy-and-paste zu verwenden. Es würde unglaublich viel Zeit kosten: Sie müssten jeden Baum einzeln zeichnen, Ast für Ast, Blatt für Blatt. Wenn Sie stattdessen ein Blatt zeichnen, es mehrfach kopieren und damit 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 einfachere 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). Das nennt sich *Transfer Learning*.

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 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 in [Kapitel 14](#) noch 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

Die Anzahl der Neuronen in den Eingabe- und Ausgabeschichten wird durch die Art von Ein- und Ausgabe bestimmt, die für Ihre Aufgabe erforderlich sind. So sind beispielsweise für MNIST $28 \times 28 = 784$ Eingabeneuronen und 10 Ausgabeneuronen notwendig.

Bei den verborgenen Schichten war es üblich, diese pyramidenförmig aufzubauen, sodass jede folgende Schicht immer weniger Neuronen enthielt – der Grund hierfür ist, dass viele einfache Merkmale sich zu deutlich weniger komplexen Merkmalen entwickeln können. Ein typisches neuronales Netz für MNIST könnte aus drei verborgenen Schichten mit 300, 200 und 100 Neuronen bestehen. Allerdings wird dies in der Praxis meist nicht umgesetzt, denn es scheint, dass in den meisten Fällen eine gleiche Anzahl von Neuronen in allen verborgenen Schichten genauso gut funktioniert oder sogar besser – damit müssen Sie zudem nur einen Hyperparameter anstatt einen pro Schicht optimieren. Aber abhängig vom Datensatz kann es manchmal helfen, die erste verborgene Schicht größer als die anderen zu machen.

Wie die Anzahl der Schichten können Sie auch die Anzahl Neuronen nach und nach erhöhen, bis das Netz overfittet. Aber in der Praxis ist es oft einfacher und effizienter, ein Modell mit mehr Schichten und Neuronen zu wählen, als Sie tatsächlich benötigen, und durch Early Stopping und anderen Regularisierungstechniken die Gefahr von Overfitting zu verhindern. Vincent Vanhoucke, ein Wissenschaftler bei Google, hat diesen Ansatz »Stretch Pants« getauft: 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. Damit vermeiden Sie Schichten, die sich als Flaschenhälse herausstellen und die Ihr Modell ruinieren können. Hat eine Schicht zu wenig Neuronen, besitzt sie nicht ausreichend repräsentierende Leistung, um all

die nützlichen Informationen aus den Eingangsdaten zu sichern (zum Beispiel kann eine Schicht mit zwei Neuronen nur zweidimensionale Daten ausgeben – verarbeitet sie also dreidimensionale Daten, werden manche Informationen verloren gehen). Egal wie groß und leistungsfähig der Rest des Netzes ist – diese Information lässt sich nie wieder rekonstruieren.



Im Allgemeinen entwickelt das Erhöhen der Anzahl Schichten mehr Durchschlagskraft als das Erhöhen der Anzahl Neuronen pro Schicht.

Lernrate, Batchgröße und andere Hyperparameter

Die Anzahl der verborgenen Schichten und der Neuronen sind nicht die einzigen Hyperparameter, an denen Sie in einem MLP drehen können. Hier gehen wir auf ein paar der wichtigsten ein und geben Tipps dazu, wie man sie setzt:

Lernrate

Die Lernrate ist der wohl wichtigste Hyperparameter. Im Allgemeinen beträgt die optimale Lernrate die Hälfte der maximalen Lernrate (also der Lernrate, über der der Trainingsalgorithmus divergiert, wie wir in [Kapitel 4](#) gesehen haben). Eine Möglichkeit zum Finden einer guten Lernrate ist, das Modell für ein paar Hundert Iterationen zu trainieren, beginnend mit einer sehr niedrigen Lernrate (beispielsweise 10^{-5}), und diese nach und nach bis zu einem sehr hohen Wert zu steigern (zum Beispiel 10). Das geschieht durch ein Multiplizieren der Lernrate mit einem konstanten Faktor bei jeder Iteration (zum Beispiel um $\exp(\log(10^6)/500)$, um von 10^{-5} in 500 Schritten zu 10 zu gelangen). Tragen Sie den Verlust als Funktion der Lernrate auf (mit einer logarithmischen Skala für die Lernrate), sollten Sie sehen, wie sie zuerst fällt. Aber nach einer Weile wird sie zu groß sein, und der Verlust sollte schnell steigen: Die optimale Lernrate wird ein bisschen unterhalb des Punkts liegen, ab dem der Verlust wieder wächst (meist etwa 10 Mal niedriger als der Wendepunkt). Sie können dann Ihr Modell neu initialisieren und es normal mit der guten Lernrate trainieren. Wir werden uns mehr Techniken zur Lernrate in [Kapitel 11](#) anschauen.

Batchgröße

Die Batchgröße kann eine deutliche Auswirkung auf die Leistung und die Trainingsdauer Ihres Modells haben. Der Hauptvorteil einer großen Batchgröße ist, dass Hardwarebeschleuniger wie zum Beispiel GPUs sie effizient verarbeiten können (siehe [Kapitel 19](#)), sodass der Trainingsalgorithmus mehr Instanzen pro Sekunde zu Gesicht bekommt. Daher empfehlen viele Forscher und Praktiker, die größte Batchgröße zu verwenden, die noch in den GPU-Speicher passt. Aber es gibt ein Problem: In der Praxis führen große Batchgrößen häufig zu Instabilitäten beim Training, insbesondere zu Beginn, und das daraus entstehende Modell kann eventuell nicht so gut generalisieren wie ein Modell mit einer kleinen Batchgröße. Im April 2018 hat Yann LeCun sogar getweetet: »Echte Freunde lassen dich keine Mini-Batches größer als 32 verwenden.«, und er zitiert damit einen Artikel (<https://homl.info/smallbatch>) aus dem gleichen Jahr von Dominic Masters und Carlo Luschi²⁴, in dem beschrieben ist, dass der Einsatz kleiner Batches (von 2 bis 32) zu bevorzugen sei, weil das zu besseren Modellen in kürzerer Trainingszeit führt. Andere Artikel weisen allerdings in die entgegengesetzte Richtung – 2017 haben Artikel von

Elad Hoffer et al. (<https://homl.info/largebatch>)²⁵ und Priya Goyal et al. (<https://homl.info/largebatch2>)²⁶ gezeigt, dass es möglich war, sehr große Batchgrößen (bis zu 8192) mithilfe verschiedener Techniken zu verwenden, wie zum Beispiel dem »Aufwärmen« der Lernrate (also dem Beginn mit einer kleinen Lernrate, die dann gesteigert wird, wie wir in [Kapitel 11](#) sehen werden). Das führte zu einer sehr kurzen Trainingszeit ohne eine Generalisierungslücke. Eine Strategie besteht also darin, eine große Batchgröße zu verwenden und die Lernrate zu steigern. Wird das Training dadurch instabil oder ist die resultierende Performance unbefriedigend, versuchen Sie es stattdessen mit einer kleineren Lernrate.

Aktivierungsfunktion

Wir haben weiter oben im Kapitel schon besprochen, wie man die Aktivierungsfunktion wählt: Im Allgemeinen ist die ReLU-Funktion eine gute Wahl für alle verborgenen Schichten. Für die Ausgabeschicht hängt sie komplett von Ihrer Aufgabe ab.

Anzahl der Iterationen

In den meisten Fällen muss die Anzahl der Trainingsiterationen nicht angepasst werden – nutzen Sie stattdessen einfach Early Stopping.



Die optimale Lernrate hängt von den anderen Hyperparametern ab – insbesondere von der Batchgröße. Verändern Sie also irgendeinen Hyperparameter, achten Sie darauf, dass Sie auch die Lernrate anpassen.

Mehr Best Practices zum Anpassen von Hyperparametern in neuronalen Netzen finden Sie in dem ausgezeichneten Artikel (<https://homl.info/1cycle>)²⁷ aus dem Jahr 2018 von Leslie Smith.

Damit beenden wir unsere Einführung in künstliche neuronale Netze und ihre Umsetzung mit Keras. In den folgenden Kapiteln werden wir Techniken zum Trainieren sehr tiefer Netze besprechen. Auch werden wir uns anschauen, wie wir Modelle mithilfe der Lower-Level-API von TensorFlow anpassen und wie wir Daten mit der Data-API effizient laden und vorverarbeiten. Und wir werden einige weitere beliebte Architekturen neuronaler Netze kennenlernen: Convolutional Neural Networks zur Bildverarbeitung, rekurrente neuronale Netze für sequenzielle Daten, Autoencoder für Representation Learning und Generative Adversarial Networks zum Modellieren und Erzeugen von Daten.²⁸

Übungen

1. Der TensorFlow Playground (<https://playground.tensorflow.org/>) ist ein nützlicher Simulator für neuronale Netze, der vom TensorFlow-Team gebaut wurde. In dieser Übung werden Sie mit nur wenigen Klicks eine Reihe binärer Klassifikatoren bauen und die Modellarchitektur und deren Hyperparameter anpassen, um mit der Funktionsweise neuronaler Netze und der Auswirkungen ihrer Hyperparameter vertraut zu werden. Nehmen Sie sich Zeit, um die folgenden Aufgaben abzuarbeiten:
 - a. Die von einem neuronalen Netz erlernten Muster. Versuchen Sie, das neuronale Standardnetz zu trainieren, indem Sie auf den Run-Button klicken (oben links). Beachten Sie, wie schnell eine gute Lösung für die Klassifikationsaufgabe gefunden

wird. Die Neuronen in der ersten verborgenen Schicht haben einfache Muster erlernt, während die Neuronen in der zweiten verborgenen Schicht gelernt haben, die einfachen Muster aus der ersten verborgenen Schicht zu komplexeren Mustern zu kombinieren. Im Allgemeinen gilt: Je mehr Schichten es gibt, desto komplexer können die Muster sein.

- b. Aktivierungsfunktionen. Probieren Sie, die tanh-Aktivierungsfunktion durch eine ReLU-Aktivierungsfunktion zu ersetzen, und trainieren Sie das Netz erneut. Sie sehen, dass noch schneller eine Lösung gefunden wird, aber die Grenzen dieses Mal linear sind. Das liegt an der Form der ReLU-Funktion.
 - c. Das Risiko lokaler Minima. Passen Sie die Netzarchitektur so an, dass es nur eine verborgene Schicht mit drei Neuronen gibt. Trainieren Sie das Netz mehrfach (zum Zurücksetzen der Gewichte klicken Sie auf den Reset-Button neben dem Play-Button). Sie sehen, dass die Trainingszeit sehr unterschiedlich ist und das Training manchmal sogar bei einem lokalen Minimum hängen bleibt.
 - d. Was passiert, wenn neuronale Netze zu klein sind? Entfernen Sie ein Neuron, sodass nur zwei übrig bleiben. Beachten Sie, dass das neuronale Netz nun keine gute Lösung mehr finden kann, auch wenn Sie es mehrfach versuchen. Das Modell hat zu wenig Parameter und underfittet die Trainingsdaten systematisch.
 - e. Was passiert, wenn neuronale Netze groß genug sind? Setzen Sie die Anzahl der Neuronen auf acht und trainieren Sie das Netz mehrere Male. Sie sehen, dass es jetzt durchgehend schnell ist und niemals stecken bleibt. Das zeigt eine wichtige Erkenntnis aus der Theorie neuronaler Netze: Große neuronale Netze bleiben so gut wie niemals in lokalen Minima stecken, und selbst wenn sie das tun, sind diese lokalen Optima fast so gut wie das globale Optimum. Allerdings können sie lange Zeit auf langen Plateaus hängen bleiben.
 - f. Das Risiko verschwindender Gradienten in tiefen Netzen. Wählen Sie den spiralförmigen Datensatz (der Datensatz unten rechts unter »DATA«) und passen Sie die Netzarchitektur so an, dass es vier verborgene Schichten mit jeweils acht Neuronen gibt. Sie sehen, dass das Training viel länger braucht und oft lange auf Plateaus verweilt. Auch tendieren die Neuronen in den höchsten Schichten (rechts) dazu, sich schneller zu entwickeln als die Neuronen in den niedrigsten Schichten (links). Dieses Problem der »verschwindenden Gradienten« lässt sich mit einer besseren Initialisierung der Gewichte und anderen Techniken, besseren Optimierern (wie AdaGrad oder Adam) oder Batch Normalization abmildern (siehe [Kapitel 11](#)).
 - g. Experimentieren Sie. Nehmen Sie sich beispielsweise mal eine Stunde Zeit, spielen Sie mit anderen Parametern herum und bekommen Sie so ein Gefühl dafür, was sie bewirken, um ein intuitives Verständnis für neuronale Netze zu erhalten.
2. 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: $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$.
 3. Warum ist eine logistische Regression dem klassischen Perzeptron (einer einzelnen Schicht von Threshold Logic 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?

4. Warum war die logistische Aktivierungsfunktion eine Schlüsselkomponente beim Trainieren der ersten MLPs?
5. Nennen Sie drei verbreitete Aktivierungsfunktionen. Können Sie diese zeichnen?
6. Angenommen, Sie hätten 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 .
7. 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](#).
8. Was ist das Backpropagation-Verfahren, und wie funktioniert es? Was ist der Unterschied zwischen Backpropagation und Autodiff im Reverse-Modus?
9. Können Sie alle Hyperparameter aufzählen, die Sie in einem einfachen MLP verändern können? Wenn das MLP die Trainingsdaten overfittet, wie könnten Sie diese Hyperparameter verändern, um das Problem zu beheben?
10. Trainieren Sie ein Deep-Learning-MLP auf dem MNIST-Datensatz (Sie können ihn mithilfe von `keras.datasets.mnist.load_data()` laden). Schauen Sie, ob Sie eine Relevanz von mehr als 98% erzielen. Versuchen Sie, die optimale Lernrate zu finden, indem Sie dem in diesem Kapitel vorgestellten Ansatz folgen (die Lernrate exponentiell wachsen zu lassen, den Verlust auszugeben und den Punkt zu finden, an dem der Verlust durch die Decke geht). Probieren Sie, alles Vorgestellte dort hineinzupacken – Checkpoints, Early Stopping und die Ausgabe von Lernkurven mithilfe von TensorBoard.

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 wenigen verborgenen Schichten. Wie lässt sich eine komplexe Aufgabe wie das Erkennen Hunderter Gegenstände in hochauflösenden Bildern angehen? Dazu müssen Sie ein weitaus tieferes DNN mit zehn oder noch viel mehr Schichten aus jeweils Hunderten Neuronen und Hunderttausenden Verbindungen trainieren. Das wird kein Spaziergang, und die folgenden sind nur ein paar der Probleme, denen Sie sich gegenübersehen können:

- Bei Deep-Learning-Netzen tritt eventuell das Problem der *schwindenden Gradienten* oder das verwandte Problem der *explodierenden Gradienten* auf. Dabei werden die Gradienten immer kleiner oder immer größer, während man sich beim Training rückwärts durch das DNN bewegt. Beide Probleme erschweren das Trainieren der ersten Schichten erheblich.
- Sie haben eventuell nicht ausreichend Trainingsdaten für solch ein großes Netz, oder das Labeln ist zu teuer.
- Das Trainieren bei einem derart großen Netz ist extrem langsam.
- Bei einem Modell mit Millionen Parametern besteht eine ernste Gefahr, die Trainingsdaten zu overfitten, insbesondere wenn es nicht ausreichend Trainingsinstanzen gibt oder diese zu verrauscht sind.

In diesem Kapitel wenden wir uns nacheinander jedem dieser Probleme zu und stellen Techniken zu deren Lösung vor. Wir beginnen mit dem Problem schwindender und explodierender Gradienten und probieren einige der beliebtesten Lösungsstrategien aus. Als Nächstes werden wir uns das Transfer Learning und unüberwachtes Pretraining anschauen, das Ihnen bei komplexen Aufgaben auch dann helfen kann, wenn Sie nur wenige gelabelte Daten haben. Anschließend werden wir unterschiedliche Optimierer betrachten, die bei großen Modellen den Trainingsvorgang 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 Kostenfunktion 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. Das bezeichnet man als das Problem der *explodierenden Gradienten*, das vor allem in rekurrenten neuronalen Netzen auftritt (siehe [Kapitel 15](#)). Allgemeiner ausgedrückt, sind die Gradienten in Deep-Learning-Netzen instabil; die Lerngeschwindigkeiten unterschiedlicher Schichten weichen stark voneinander ab.

Dieses ungünstige Verhalten wurde schon vor einer Weile beobachtet und war einer der Gründe, aus dem man Deep-Learning-Netze bis in die frühen 2000er-Jahre beiseitegelegt hatte. Es war nicht klar, was die Gradienten dazu bewegte, beim Trainieren eines DNN so instabil zu sein, aber ein Artikel (<https://homl.info/47>) aus dem Jahr 2010 von Xavier Glorot und Yoshua Bengio¹ fand einige Hauptverdächtige, darunter die Kombination der beliebten logistischen Aktivierungsfunktion mit der damals verbreiteten zufälligen 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.

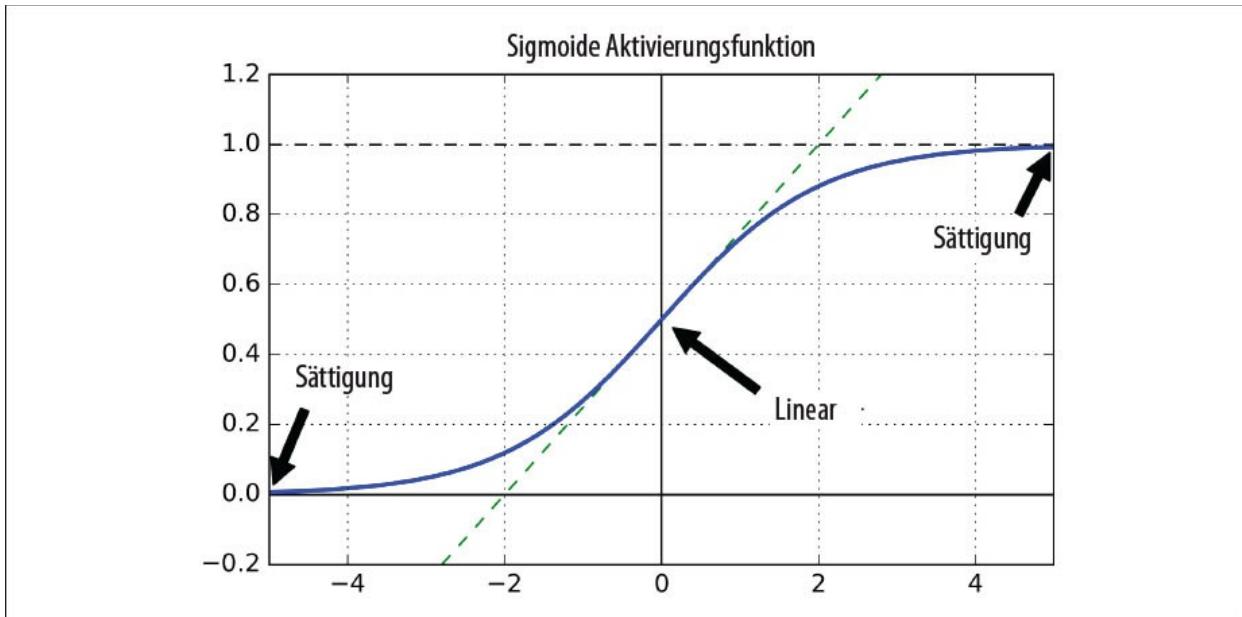


Abbildung 11-1: Sättigung der logistischen Aktivierungsfunktion

Initialisierung nach Glorot und He

In ihrem Artikel empfehlen Glorot und Bengio einen Weg, das Problem der instabilen Gradienten deutlich zu mildern. Sie weisen darauf hin, dass das Signal in beide Richtungen fließen können muss: 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 Verbindungen und Neuronen (diese Zahlen werden als *Fan-in* und *Fan-out* der Schicht bezeichnet), aber die Autoren schlügen einen praktisch gut funktionierenden Kompromiss vor: Die Gewichte der Verbindungen müssen zufällig gesetzt werden, wie in [Formel 11-1](#) beschrieben, wobei gilt $\text{fan}_{\text{avg}} = (\text{fan}_{\text{in}} + \text{fan}_{\text{out}})/2$. Diese Initialisierungsstrategie wird *Initialisierung nach Xavier* oder *Initialisierung nach Glorot* genannt – nach dem ersten Autor des Artikels.

Formel 11-1: Initialisierung nach Glorot (beim Verwenden der logistischen Aktivierungsfunktion)

$$\text{Normalverteilung mit Mittelwert } 0 \text{ und Standardabweichung } \sigma^2 = \sqrt{\frac{1}{\text{fan}_{\text{avg}}}}$$

Oder eine Gleichverteilung zwischen $-r$ und $+r$, mit $r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$

Ersetzen Sie in [Formel 11-1](#) fan_{avg} durch fan_{in} , erhalten Sie eine Initialisierungsstrategie, die Yann LeCun in den 1990ern vorgeschlagen hat. Er nannte sie *Initialisierung nach LeCun*.

Genevieve Orr und Klaus-Robert Müller empfahlen sie sogar in ihrem Buch *Neural Networks: Tricks of the Trade* (Springer) aus dem Jahr 1998. Die LeCun-Initialisierung ist äquivalent zur Glorot-Initialisierung, wenn $fan_{in} = fan_{out}$. Es dauerte über ein Jahrzehnt, bis die Forscher erkannten, wie wichtig dieser Trick ist. Der Einsatz der Glorot-Initialisierung kann das Training deutlich beschleunigen, und sie ist einer der Tricks, die zum Erfolg von Deep Learning beigetragen haben.

Es wurden ähnliche Strategien für andere Aktivierungsfunktionen vorgeschlagen.³ Diese unterscheiden sich nur in der Größe der Varianz und dem Einsatz von fan_{out} oder fan_{in} , wie Sie in [Tabelle 11-1](#) sehen (für die gleichförmige Verteilung berechnen Sie einfach $r = \sqrt{3\sigma^2}$). Die Initialisierungsstrategie (<https://homl.info/48>) für die ReLU-Aktivierungsfunktion (und ihre Varianten, einschließlich der kurz beschriebenen ELU-Aktivierung) wird manchmal *Initialisierung nach He* genannt – nach dem ersten Autor des Artikels. Die SELU-Aktivierungsfunktion wird weiter unten noch erläutert. Sie sollte zusammen mit der LeCun-Initialisierung eingesetzt werden (vorzugsweise mit einer Normalverteilung, wie wir sehen werden).

Tabelle 11-1: Initialisierungsparameter für jede Art von Aktivierungsfunktion

Initialisierung	Aktivierungsfunktionen	σ^2 Normal
Glorot	keine, tanh, logistisch, Softmax	$1 / fan_{avg}$
He	ReLU und Varianten	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Standardmäßig nutzt Keras die Glorot-Initialisierung mit einer Gleichverteilung. Beim Erstellen einer Schicht können Sie die He-Initialisierung einsetzen, indem Sie `kernel_initializer="he_uniform"` oder `kernel_initializer="he_normal"` verwenden, zum Beispiel so:

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Wollen Sie die He-Initialisierung mit einer gleichförmigen Verteilung nutzen, die aber auf fan_{avg} statt auf fan_{in} basiert, können Sie den `VarianceScaling`-Initialisierer wie folgt einsetzen:

```
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                                    distribution='uniform')

keras.layers.Dense(10, activation="sigmoid", kernel_initializer=he_avg_init)
```

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 instabiler Gradienten war. Bis dahin hatten die meisten Menschen 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, das bedeutet, 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. Ein Neuron stirbt, wenn seine Gewichte so angepasst werden, dass die gewichtete Summe seiner Eingaben für alle Instanzen im Trainingsdatensatz negativ ist. Geschieht das, wird immer nur 0 ausgegeben, und die Gradientenmethode hat keine Auswirkungen mehr, weil der Gradient der ReLU-Funktion bei negativen Werten null ist.⁴

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 im Jahr 2015 erschienener Artikel (<https://homl.info/49>)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. Diese Funktion schnitt ebenfalls recht gut ab und schien als Regularisierung zu fungieren (also das Risiko für Overfitting der Trainingsdaten zu senken). Schließlich wurde auch die *parametrisierte 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.

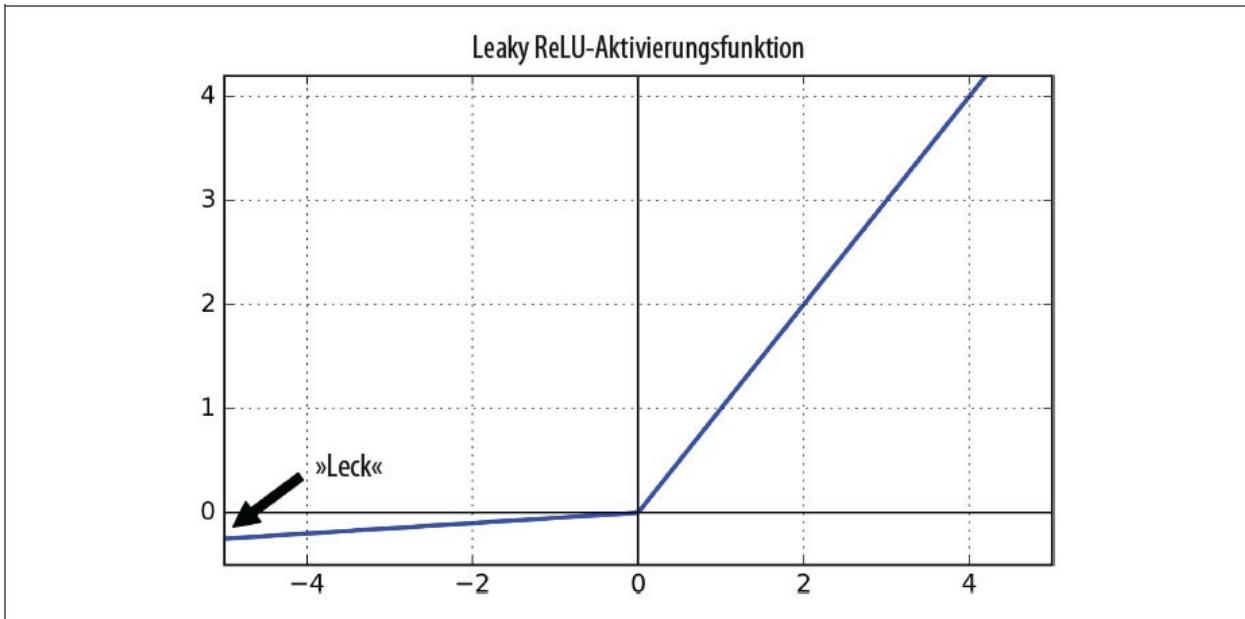


Abbildung 11-2: Leaky ReLU: wie ReLU, aber mit einer kleinen Steigung für negative Werte

Schließlich schlug ein Artikel (<https://homl.info/50>) aus dem Jahr 2015 von Djork-Arné Clevert et al.⁶ eine neue Aktivierungsfunktion namens *Exponential Linear Unit* (ELU) vor, die im Experiment sämtliche ReLU-Varianten aus dem Feld 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

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{wenn } z < 0 \\ z & \text{wenn } z \geq 0 \end{cases}$$

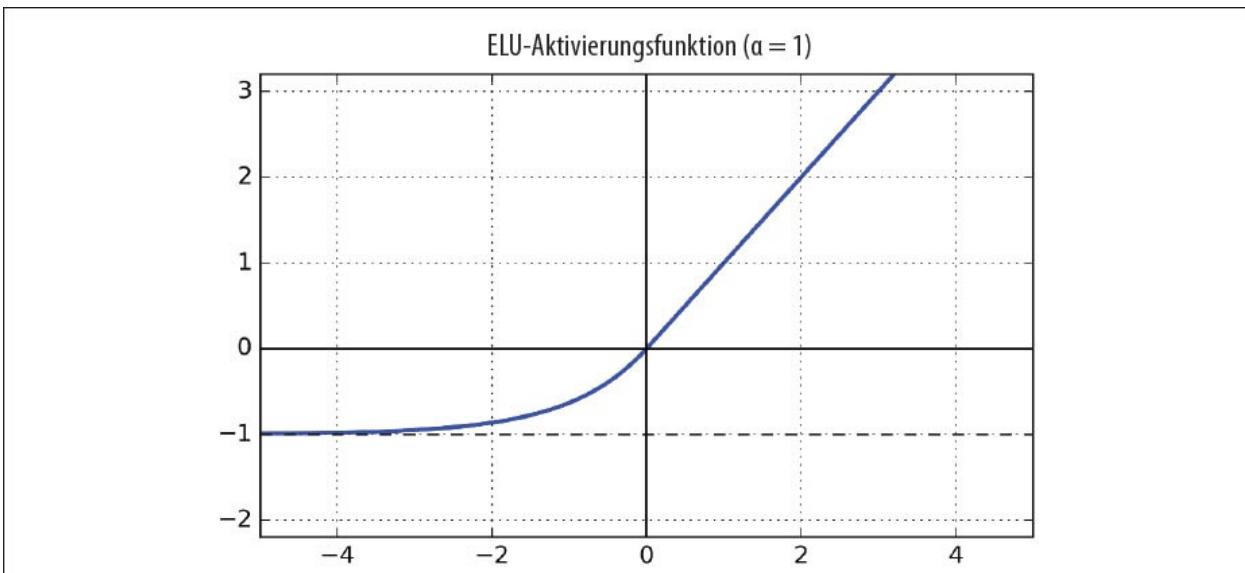


Abbildung 11-3: ELU-Aktivierungsfunktion

Sie sieht der ReLU-Funktion sehr ähnlich, weist aber einige große Unterschiede auf:

- Sie nimmt bei $z < 0$ negative Werte an, wodurch das Neuron eine durchschnittliche Ausgabe um 0 haben kann und das Problem schwindender Gradienten bekämpft. 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.
- Die Ableitung für $z < 0$ ist ungleich 0, was das Problem toter Neuronen vermeidet.
- Ist $\alpha = 1$, 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.

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.

Im Jahr 2017 hat dann ein Artikel (<https://homl.info/selu>) von Günter Klambauer et al.⁷ die Scaled-ELU-Aktivierungsfunktion (SELU) vorgestellt: Wie ihr Name schon andeutet, handelt es sich bei ihr um eine skalierte Variante der ELU-Aktivierungsfunktion. Die Autoren haben gezeigt, dass das Netz *selbstnormalisierend* sein wird, wenn Sie ein neuronales Netz nur aus dichten Schichten aufbauen und alle verborgenen Schichten die SELU-Aktivierungsfunktion nutzen: Die Ausgabe jeder Schicht tendiert dann dazu, während des Trainings einen Mittelwert von 0 und eine Standardabweichung von 1 zu erreichen, was das Problem der verschwindenden oder explodierenden Gradienten löst. Im Ergebnis funktioniert die SELU-Aktivierungsfunktion häufig signifikant besser für solche neuronalen Netze (insbesondere Deep-Learning-Netze). Es gibt allerdings ein paar Bedingungen, damit die Selbstnormalisierung auch geschehen kann (im Artikel finden Sie die mathematischen Begründungen dafür):

- Die Eingabemerkmale müssen standardisiert sein (Mittelwert 0 und Standardabweichung 1).
- Die Gewichte jeder verborgenen Schicht müssen mit der LeCun-Normal-Initialisierung belegt werden. In Keras müssen Sie dafür `kernel_initializer= "lecun_normal"` setzen.
- Die Architektur des Netzes muss sequenziell sein. Versuchen Sie, SELU in nicht sequenziellen Architekturen einzusetzen, wie zum Beispiel bei rekurrenten Netzen (siehe [Kapitel 15](#)) oder in Netzen mit Skip-Verbindungen (also Verbindungen, die Schichten überspringen, wie zum Beispiel in Wide-&-Deep-Netzen), ist die Selbstnormalisierung leider nicht garantiert, daher wird SELU nicht unbedingt besser funktionieren als andere Aktivierungsfunktionen.
- Der Artikel garantiert eine Selbstnormalisierung nur, wenn alle Schichten dicht sind, aber manche Forscher haben angemerkt, dass die SELU-Aktivierungsfunktion auch die Leistung in Convolutional Neural Networks verbessern kann (siehe [Kapitel 14](#)).

Welche Aktivierungsfunktion sollten Sie also in den verborgenen Schichten Ihrer Deep-

Learning-Netze verwenden? Auch wenn es im Einzelfall Unterschiede gibt, gilt als Faustregel SELU > ELU > Leaky ReLU (und Varianten) > ReLU > tanh > logistisch. Verhindert die Netzarchitektur eine Selbstdnormalisierung, kann ELU besser funktionieren als SELU (da SELU bei $z = 0$ nicht stetig ist). 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 (zum Beispiel 0,3 für Leaky ReLU). 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. Und weil ReLU die (bisher) am häufigsten verwendete Aktivierungsfunktion ist, enthalten viele Bibliotheken und Hardwarebeschleuniger ReLU-spezifische Optimierungen – ist Ihnen also Geschwindigkeit am wichtigsten, kann ReLU weiterhin die beste Wahl sein.

Um die Leaky-ReLU-Aktivierungsfunktion zu verwenden, erzeugen Sie eine Leaky ReLU-Schicht und fügen sie zu Ihrem Modell direkt nach der Schicht hinzu, auf die Sie sie anwenden wollen:

```
model = keras.models.Sequential([
    [...],
    keras.layers.Dense(10, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(alpha=0.2),
    [...]
])
```

Für PReLU ersetzen Sie `LeakyReLU(alpha=0.2)` durch `PReLU()`. Es gibt in Keras aktuell keine offizielle Implementierung von RReLU, aber Sie können sie ziemlich einfach selbst implementieren (um zu lernen, wie das geht, schauen Sie sich die Übungen am Ende von [Kapitel 12](#) an).

Zur SELU-Aktivierung setzen Sie beim Erstellen einer Schicht `activation="selu"` und `kernel_initializer="lecun_normal"`:

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

Batchnormalisierung

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 (<https://homl.info/51>) aus dem Jahr 2015⁸ schlagen Sergey Ioffe und Christian

Szegedy eine Technik namens *Batchnormalisierung* (BN) vor, um diese Probleme anzugehen. Die Technik besteht aus dem Hinzufügen einer Operation kurz vor der Aktivierungsfunktion in jeder verborgenen Schicht des Modells. Dabei werden einfach die Eingaben auf null zentriert und normalisiert, und das Ergebnis wird anschließend mit zwei neuen Parametern pro Schicht skaliert und verschoben (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. In vielen Fällen brauchen Sie Ihren Trainingsdatensatz nicht zu standardisieren (zum Beispiel mit einem `StandardScaler`), wenn Sie eine BN-Schicht als allererste Schicht in Ihr neuronales Netz einfügen – die BN-Schicht erledigt das schon für Sie (nun, zumindest näherungsweise, da sie sich immer nur einen Batch gleichzeitig anschaut, zudem kann sie jedes Eingabemerkmal umskalieren und verschieben).

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 »Batchnormalisierung«). Die gesamte Prozedur ist in [Formel 11-3](#) zusammengefasst.

Formel 11-3: Algorithmus zur Batchnormalisierung

1. $\boldsymbol{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_B)^2$
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\sigma_B^2 + \epsilon}}$
4. $\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$

Dabei gilt:

- $\boldsymbol{\mu}_B$ ist der Vektor der Eingabe-Mittelwerte für den gesamten Mini-Batch B (er enthält einen Mittelwert pro Eingabe).
- σ_B ist der Vektor mit den Eingabe-Standardabweichungen, ebenfalls für den gesamten Mini-Batch bestimmt (er enthält eine Standardabweichung pro Eingabe).
- m_B ist die Anzahl Datenpunkte im Mini-Batch.
- $\hat{\mathbf{x}}^{(i)}$ ist der Vektor mit auf null zentrierten und normalisierten Eingaben für Instanz i .
- γ ist der Vektor mit den Ausgabe-Skalierungsparametern für die Schicht (er enthält einen Skalierungsparameter pro Eingabe).
- \otimes steht für eine elementweise Multiplikation (jeder Eingabewert wird mit seinem zugehörigen Ausgabe-Skalierungsfaktor multipliziert).
- β ist der Vektor mit den Parametern zum Verschieben der Schicht (Offset). Er enthält einen Offset-Parameter pro Eingabe. Jede Eingabe wird durch ihren zugehörigen Verschiebeparameter verschoben.
- 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.

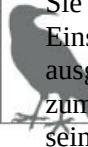
Während des Trainings standardisiert BN also seine Eingaben, um sie dann umzuskalieren und zu verschieben. Gut! Wie sieht es mit dem Testen aus? Nun, das ist nicht so einfach. Tatsächlich müssen wir eventuell Vorhersagen für einzelne Instanzen statt für ganze Batches treffen. Und selbst wenn wir einen Batch haben, kann er zu klein sein, oder die Instanzen sind nicht unabhängig und gleichförmig verteilt, sodass das Berechnen von Statistiken über die Batchinstanzen unzuverlässig wäre. Eine Lösung bestünde darin, bis zum Ende des Trainings zu warten, dann den gesamten Trainingsdatensatz durch das neuronale Netz zu schicken und Mittelwert und Standardabweichung jeder Eingabe der BN-Schicht zu berechnen. Diese »finalen« Eingabe-Mittelwerte und -Standardabweichungen könnten dann statt der Batch-Eingabe-Mittelwerte und -Standardabweichungen während der Vorhersagen zum Einsatz kommen. Aber die meisten Implementierungen der Batchnormalisierung schätzen diese abschließenden Statistiken während des Trainings mithilfe eines gleitenden Durchschnitts der Mittelwerte und Standardabweichungen der Schichteingaben. Das macht Keras automatisch, wenn Sie die `BatchNormalization`-Schicht einsetzen. Fassen wir zusammen: Vier Parametervektoren werden in jeder batchnormalisierten Schicht gelernt – γ (der Ausgabe-Skalierungs-Vektor) und β (der Ausgabe-Offset-Vektor) werden während der normalen Backpropagation gelernt, während μ (der finale Eingabe-Mittelwert-Vektor) und σ (der finale Eingabe-Standardabweichungs-Vektor) mithilfe eines exponentiellen gleitenden Mittelwerts entstehen. Beachten Sie, dass μ und σ während des Trainings geschätzt werden, aber nur danach zum Einsatz kommen (um die Batch-Eingabe-Mittelwerte und -Standardabweichungen in [Formel 11-3](#) zu ersetzen).

Ioffe und Szegedy haben gezeigt, dass die Batchnormalisierung alle Deep-Learning-Netze, mit denen sie experimentiert haben, deutlich verbessert hat, was zu einer großen Verbesserung bei der ImageNet-Klassifikationsaufgabe führte (ImageNet ist eine große Datenbank mit Bildern, die in vielen Kategorien klassifiziert sind und oft zum Überprüfen von Bilderkennungssystemen dient). Das Problem der verschwindenden Gradienten wurde so deutlich reduziert, dass sogar gesättigte Aktivierungsfunktionen wie `tanh` oder die logistische Aktivierungsfunktion zum Einsatz kommen konnten. Die Netze reagierten zudem deutlich unempfindlicher auf die Gewichtsinitialisierung. Die Autoren konnten viel höhere Lernraten nutzen und den Lernprozess signifikant beschleunigen. Insbesondere merken sie an:

Angewendet auf ein aktuelles Bildklassifikationsmodell erreicht Batchnormalisierung die gleiche Genauigkeit mit 14 Mal weniger Trainingsschritten und schlägt das Ursprungsmodell deutlich. [...] Bei einem Ensemble batchnormalisierter Netze verbessern wir das beste veröffentlichte Ergebnis bei der ImageNet-Klassifikation: Wir erreichen 4,9% Top-5-Validierungsfehler (und 4,8% Testfehler) und sind damit besser als menschliche Klassifikatoren.

Und schließlich – wie ein Geschenk, das immer mehr liefert – verhält sich die Batchnormalisierung wie ein Regulierer und verringert die Notwendigkeit für andere Regulierungstechniken (wie das später in diesem Kapitel beschriebene Drop-out).

Aber durch Batchnormalisierung wird das Modell komplexer (auch wenn – wie schon besprochen – dadurch die Notwendigkeit des Normalisierens der Eingabedaten wegfallen kann). Zudem gibt es Zusatzkosten zur Laufzeit: Das neuronale Netz macht langsamere Vorhersagen aufgrund der zusätzlichen Berechnungen, die auf jeder Schicht erforderlich sind. Zum Glück ist es oft möglich, die BN-Schicht nach dem Training mit der vorherigen Schicht zu verschmelzen und damit die Zusatzkosten zu vermeiden. Das geschieht durch das Aktualisieren der Gewichte und Biase der vorherigen Schicht, sodass diese direkt die Ausgaben mit passender Skalierung und Offset ermittelt. Berechnet beispielsweise die vorherige Schicht $\mathbf{XW} + \mathbf{b}$, wird die BN-Schicht $\gamma \otimes (\mathbf{XW} + \mathbf{b} - \mu) / \sigma + \beta$ berechnen (wenn wir mal den Smoothing-Term ϵ im Nenner ignorieren). Definieren wir $\mathbf{W}' = \gamma \otimes \mathbf{W} / \sigma$ und $\mathbf{b}' = \gamma \otimes (\mathbf{b} - \mu) / \sigma + \beta$, vereinfacht sich die Gleichung zu $\mathbf{XW}' + \mathbf{b}'$. Ersetzen wir also in der vorherigen Schicht Gewichte und Biase (\mathbf{W} und \mathbf{b}) durch die aktualisierten Gewichte und Biase (\mathbf{W}' und \mathbf{b}'), können wir die BN-Schicht loswerden (der Optimizer von TFLite tut das automatisch, siehe [Kapitel 19](#)).



Sie werden eventuell feststellen, dass das Training eher langsam abläuft, weil jede Epoche beim Einsatz der Batchnormalisierung viel mehr Zeit verbraucht. Das wird im Allgemeinen dadurch ausgeglichen, dass die Konvergenz mit BN viel schneller geschieht und damit weniger Epochen zum Erreichen der gleichen Leistung notwendig sind. Insgesamt wird die *Wall Time* meist kürzer sein (das ist die Zeit, die von Ihrer Uhr an der Wand gemessen wird).

Implementieren der Batchnormalisierung mit Keras

Wie das meiste bei Keras ist auch das Implementieren der Batchnormalisierung einfach und intuitiv. Fügen Sie einfach eine `BatchNormalization`-Schicht vor oder nach jeder Aktivierungsfunktion verborgener Schichten ein und ergänzen Sie optional eine BN-Schicht als erste in Ihrem Modell. Das folgende Modell wendet beispielsweise BN nach jeder verborgenen Schicht und als erste Schicht an (nach dem Verflachen der Eingabebilder):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

Das ist alles! In diesem kleinen Beispiel mit nur zwei verborgenen Schichten ist es unwahrscheinlich, dass die Batchnormalisierung eine sehr positive Auswirkung haben wird – aber für tiefere Netze kann es einen deutlichen Unterschied ausmachen.

Schauen wir uns die Modellzusammenfassung an:

```
>>> model.summary()
```

```
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
<hr/>		
flatten_3 (Flatten)	(None, 784)	0
<hr/>		
batch_normalization_v2 (Batch Normalization)	(None, 784)	3136
<hr/>		
dense_50 (Dense)	(None, 300)	235500
<hr/>		
batch_normalization_v2_1 (Batch Normalization)	(None, 300)	1200
<hr/>		
dense_51 (Dense)	(None, 100)	30100
<hr/>		
batch_normalization_v2_2 (Batch Normalization)	(None, 100)	400
<hr/>		
dense_52 (Dense)	(None, 10)	1010
<hr/>		
Total params: 271,346		
Trainable params: 268,978		
Non-trainable params: 2,368		

Wie Sie sehen, fügt jede BN-Schicht vier Parameter pro Eingabe hinzu: γ , β , μ und σ (so ergänzt beispielsweise die erste BN-Schicht 3.136 Parameter, also 4×784). Die letzten beiden Parameter μ und σ sind die gleitenden Mittelwerte – sie sind nicht von der Backpropagation betroffen, daher nennt Keras sie »nicht trainierbar«⁹ (wenn Sie die Gesamtsumme der BN-Parameter zusammenzählen [3136 + 1200 + 400] und durch 2 teilen, erhalten Sie 2.368, was der Gesamtzahl nicht trainierbarer Parameter in diesem Modell entspricht).

Schauen wir uns die Parameter der ersten BN-Schicht an. Zwei sind trainierbar (durch

Backpropagation), zwei nicht:

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization_v2/gamma:0', True),
 ('batch_normalization_v2/beta:0', True),
 ('batch_normalization_v2/moving_mean:0', False),
 ('batch_normalization_v2/moving_variance:0', False)]
```

Erstellen Sie nun eine BN-Schicht in Keras, werden auch zwei Operationen angelegt, die von Keras bei jeder Iteration während des Trainings aufgerufen werden. Diese Operationen aktualisieren die gleitenden Mittelwerte. Da wir das TensorFlow-Backend verwenden, handelt es sich bei diesen Operationen um TensorFlow-Operationen (wir werden diese in [Kapitel 12](#) behandeln):

```
>>> model.layers[1].updates
[<tf.Operation 'cond_2/Identity' type=Identity>,
 <tf.Operation 'cond_3/Identity' type=Identity>]
```

Die Autoren des BN-Artikels plädieren dafür, die BN-Schichten vor den Aktivierungsfunktionen hinzuzufügen und nicht erst dahinter (wie wir es getan haben). Darüber gibt es Diskussionen, aber es hängt auch von der Aufgabe ab – Sie können damit experimentieren, um zu sehen, welche Option mit Ihrem Datensatz am besten funktioniert. Um die BN-Schichten vor die Aktivierungsfunktionen zu setzen, müssen Sie die Funktionen aus den verborgenen Schichten entfernen und als eigene Schichten hinter den BN-Schichten einfügen. Und da eine Batchnormalisierungsschicht einen Offset-Parameter pro Eingabe besitzt, können Sie den Bias-Term aus der vorherigen Schicht entfernen (übergeben Sie beim Erstellen einfach `use_bias=False`):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
```

```

    keras.layers.Activation("elu"),
    keras.layers.Dense(10, activation="softmax")
])

```

Die Klasse `BatchNormalization` besitzt eine Reihe von Hyperparametern, an denen Sie drehen können. Die Standardwerte sind im Allgemeinen schon gut, aber gelegentlich müssen Sie das `momentum` anpassen. Dieser Hyperparameter wird von der `BatchNormalization`-Schicht genutzt, wenn sie die exponentiellen gleitenden Mittelwerte aktualisiert – bei einem neuen Wert \mathbf{v} (also einem neuen Vektor mit Eingabe-Mittelwerten oder -Standardabweichungen, der mit dem aktuellen Batch berechnet wurde) aktualisiert die Schicht den gleitenden Mittelwert $\hat{\mathbf{v}}$ über diese Gleichung:

$$\hat{\mathbf{v}} \leftarrow \hat{\mathbf{v}} \times \text{momentum} + \mathbf{v} \times (1 - \text{momentum})$$

Ein gutes Momentum liegt meist nahe an 1 – zum Beispiel 0,9, 0,99 oder 0,999 (Sie wollen mehr Neunen für größere Datensätze und kleinere Mini-Batches haben).

Ein weiterer wichtiger Hyperparameter ist `axis`: Er bestimmt, welche Achse normalisiert werden soll. Standardwert ist -1 , womit die letzte Achse normalisiert wird (mit Mittelwerten und Standardabweichungen, die aus den *anderen* Achsen berechnet wurden). Ist der Eingabebatch zweidimensional (dann ist die Batchform `[Batchgröße, Features]`), wird jedes Eingabemerkmal basierend auf dem Mittelwert und der Standardabweichung normalisiert, die über alle Instanzen im Batch berechnet wurden. So wird beispielsweise die erste BN-Schicht im vorherigen Codebeispiel jedes der 784 Eingangsmerkmale unabhängig voneinander normalisieren (sowie umskalieren und verschieben). Verschieben wir die erste BN-Schicht vor die `Flatten`-Schicht, wird der Eingabebatch dreidimensional sein, und die Form `[Batchgröße, Höhe, Breite]` haben – die BN-Schicht wird daher 28 Mittelwerte und 28 Standardabweichungen berechnen (eine pro Pixelspalte, berechnet über alle Instanzen im Batch und über alle Zeilen in den Spalten) und alle Pixel in einer gegebenen Spalte mit dem gleichen Mittelwert und der gleichen Standardabweichung normalisieren. Es wird auch nur 28 Skalierungsparameter und 28 Offset-Parameter geben. Wollen Sie stattdessen jedes einzelne der 784 Pixel unabhängig behandeln, sollten Sie `axis=[1, 2]` setzen.

Beachten Sie, dass die BN-Schicht nicht die gleichen Berechnungen während des und nach dem Training durchführt: Sie nutzt Batchstatistik während des Trainings und die »finale« Statistik nach dem Training (also die abschließenden Werte der gleitenden Mittelwerte). Werfen wir einen Blick in diese Klasse, um zu sehen, wie das geschieht:

```

class BatchNormalization(keras.layers.Layer):
    [...]
    def call(self, inputs, training=None):
        [...]

```

In der Methode `call()` werden die Berechnungen durchgeführt – wie Sie sehen, besitzt sie ein zusätzliches Argument `training`, das standardmäßig auf `None` gesetzt ist. Die Methode `fit()` setzt sie aber während des Trainings auf `1`. Müssen Sie eine eigene Schicht schreiben und muss sich diese im Training und beim Testen unterschiedlich verhalten, fügen Sie deren Methode `call()` ebenfalls ein Argument `training` hinzu und nutzen dieses in der Methode, um zu entscheiden, was zu berechnen ist¹⁰ (wir werden eigene Schichten in [Kapitel 12](#) behandeln).

BatchNormalization gehört in Deep-Learning-Netzen inzwischen zu den am häufigsten eingesetzten Schichten – sie wird in den Diagrammen sogar häufig weggelassen, da davon ausgegangen wird, dass sie hinter jeder Schicht eingefügt ist. Aber ein aktueller Artikel (<https://homl.info/fixup>) von Honyi Zhang et al.¹¹ kann diese Annahme eventuell ändern: Durch eine neue *Fixed-Update-* (Fixup-)Gewichtsinitialisierungstechnik haben es die Autoren geschafft, ein sehr tiefes neuronales Netz (10. 000 Schichten!) ohne BN zu trainieren und dabei beste Leistungen bei komplexen Bildklassifikationsaufgaben zu erreichen. Da dies aktuellste Forschung ist, sollten Sie aber vielleicht auf zusätzliche Untersuchungen warten, die diese Erkenntnisse untermauern, bevor Sie die Batchnormalisierung wegwerfen.

Gradient Clipping

Eine weitere beliebte Technik, um das Problem der explodierenden Gradienten zu entschärfen, ist, die Gradienten während der Backpropagation zu kappen, sodass sie niemals einen festgelegten Schwellenwert überschreiten (hilfreich vor allem bei rekurrenten neuronalen Netzen, siehe [Kapitel 15](#)). Man nennt diese Technik *Gradient Clipping* (<https://homl.info/52>).¹² Diese wird meist in Recurrent Neural Networks eingesetzt, da sich dort die Batchnormalisierung nur schwer verwenden lässt (wie wir in [Kapitel 15](#) sehen werden). Für andere Arten von Netzen ist die BN meist ausreichend.

In Keras müssen Sie zum Implementieren des Gradient Clipping nur beim Erstellen eines Optimierers das Argument `clipvalue` oder `clipnorm` setzen, zum Beispiel:

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)  
model.compile(loss="mse", optimizer=optimizer)
```

Dieser Optimierer beschneidet jede Komponente des Gradientenvektors auf einen Wert zwischen $-1,0$ und $1,0$. Damit werden alle partiellen Ableitungen des Verlusts (bezüglich jedes trainierbaren Parameters) auf den Bereich zwischen $-1,0$ und $1,0$ beschränkt. Der Grenzwert ist ein Hyperparameter, den Sie anpassen können. Beachten Sie, dass er die Richtung des Gradientenvektors verändern kann. Hat beispielsweise der ursprüngliche Gradientenvektor den Wert $[0,9; 100,0]$, weist er nahezu in die Richtung der zweiten Achse; nach dem Beschneiden erhalten Sie aber $[0,9; 1,0]$, was ungefähr diagonal zwischen den beiden Achsen steht. In der Praxis funktioniert dieser Ansatz gut. Wollen Sie sichergehen, dass das Gradient Clipping die Richtung des Gradientenvektors nicht ändert, sollten Sie die Norm begrenzen, indem Sie `clipnorm` statt `clipvalue` setzen. So wird der gesamte Gradient beschnitten, wenn seine ℓ_2 -Norm größer als der ausgewählte Grenzwert ist. Setzen Sie beispielsweise `clipnorm=1.0`, wird

der Vektor $[0,9; 100,0]$ auf $[0,00899964; 0,9999595]$ begrenzt. Damit wird seine Richtung beibehalten, aber die erste Komponente fast ausgelöscht. Beobachten Sie, dass die Gradienten während des Trainings explodieren (sie können die Größe der Gradienten über TensorBoard verfolgen), ist es einen Versuch wert, die Werte oder die Norm mit unterschiedlichen Grenzwerten zu beschränken und zu sehen, welche Option mit dem Validierungsdatensatz am besten funktioniert.

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 (wir werden in [Kapitel 14](#) besprechen, wie Sie solche finden). 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](#)).

- Wenn die Bilder bei Ihrer neuen Aufgabe nicht die gleiche Größe haben wie die in der ursprünglichen Aufgabe verwendeten, müssen Sie meist einen Vorverarbeitungsschritt einbauen, 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.

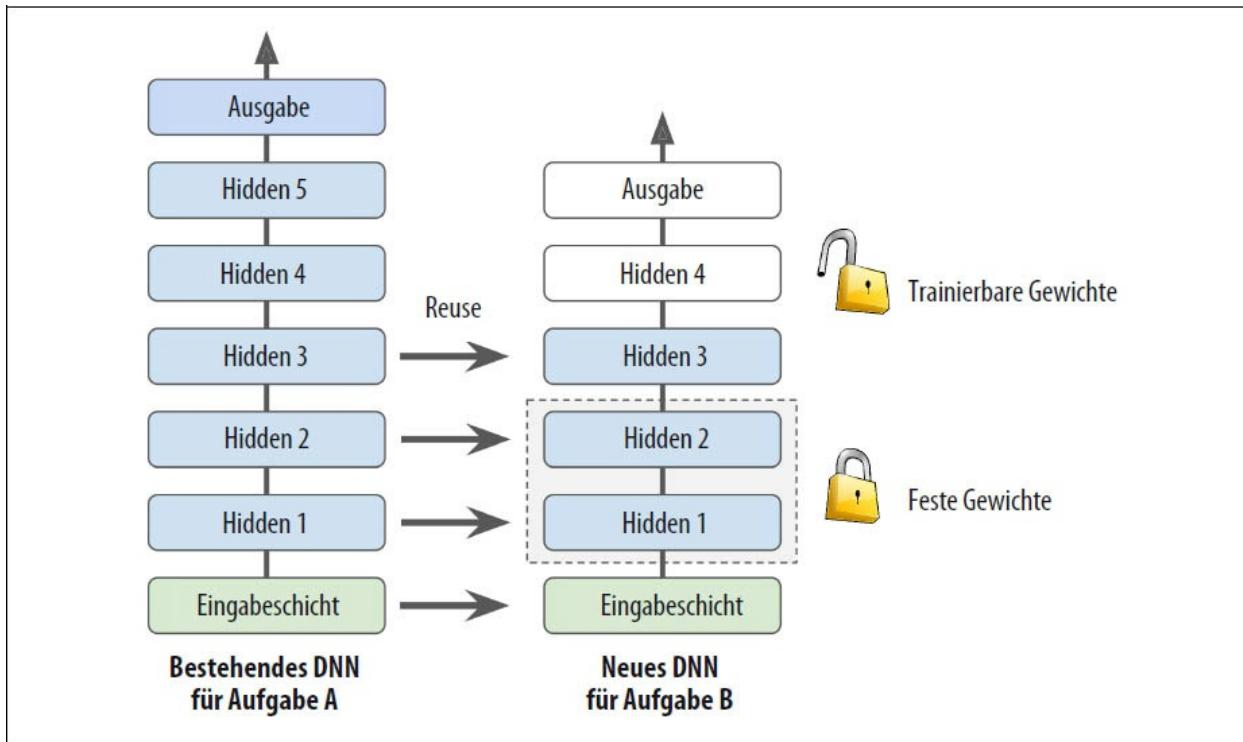


Abbildung 11-4: Wiederverwenden vortrainierter Schichten

Die Ausgabeschicht des ursprünglichen Modells sollte normalerweise ersetzt werden, da sie für die neue Aufgabe meist nicht nützlich ist und es eventuell nicht einmal die richtige Anzahl an Ausgaben gibt.

Genauso sind die oberen verborgenen Schichten des ursprünglichen Modells eher nicht so hilfreich wie die unteren Schichten, da sich die High-Level-Eigenschaften, die für die neue Aufgabe am nützlichsten sind, deutlich von denen unterscheiden können, die für die Ursprungsaufgabe hilfreich waren. Sie müssen eben die richtige Zahl an Schichten finden, die sich wiederverwenden lassen.



Je ähnlicher sich die Aufgaben sind, desto mehr Schichten können Sie wiederverwenden (beginnend mit den unteren Schichten). Bei sehr ähnlichen Aufgaben können Sie versuchen, alle verborgenen Schichten beizubehalten und die Ausgabeschicht zu ersetzen.

Versuchen Sie, alle wiederverwendeten Schichten zunächst einzufrieren (also ihre Gewichte nicht trainierbar zu machen, sodass die Gradientenmethode sie nicht verändert), um dann Ihr Modell zu trainieren und zu sehen, wie gut es funktioniert. Probieren Sie dann, eine oder zwei der obersten verborgenen Schichten wieder »aufzutauen«, damit die Backpropagation sie anpassen kann – prüfen Sie, ob sich die Leistung dadurch verbessert. Je mehr Trainingsdaten Sie haben, desto mehr Schichten können Sie auftauen. Es ist auch nützlich, dabei die Lernrate zu verringern – so vermeiden Sie, Ihre sorgsam angepassten Gewichte durcheinanderzubringen.

Erreichen Sie immer noch keine gute Performance und haben Sie nur wenige Trainingsdaten, versuchen Sie, die oberste verborgene Schicht (oder mehrere davon) zu verwerfen und die

verbleibenden verborgenen Schichten wieder einzufrieren. Sie können das so lange ausprobieren, bis Sie die richtige Zahl von wiederzuverwendenden Schichten gefunden haben. Stehen Ihnen sehr viele Trainingsdaten zur Verfügung, können Sie versuchen, die obersten verborgenen Schichten zu ersetzen, statt sie wegzwerfen, oder sogar noch mehr Schichten hinzuzufügen.

Transfer Learning mit Keras

Schauen wir uns ein Beispiel an. Stellen Sie sich vor, der Fashion-MNIST-Datensatz würde nur acht Kategorien enthalten – beispielsweise alle außer Sandale und Shirt. Jemand hat ein Keras-Modell für diesen Datensatz gebaut und trainiert und eine ausreichend gute Leistung erreicht (>90% Genauigkeit). Nennen wir es Modell A. Sie wollen nun eine andere Aufgabe angehen: Sie haben Bilder von Sandalen und Shirts und wollen einen binären Klassifikator trainieren (positiv = Shirt, negativ = Sandale). Ihr Datensatz ist ziemlich klein, Sie haben nur 200 gelabelte Bilder. Trainieren Sie ein neues Modell für diese Aufgabe (nennen wir es Modell B) mit der gleichen Architektur wie Modell A, wird sich dieses ziemlich gut schlagen (97,2% Genauigkeit). Aber da es sich um eine viel einfachere Aufgabe handelt (es sind nur zwei Kategorien), haben Sie auf mehr gehofft. Während Sie Ihren ersten Kaffee trinken, geht Ihnen auf, dass Ihre Aufgabe der alten Aufgabe ziemlich ähnelt, also kann vielleicht Transfer Learning helfen? Finden wir es heraus.

Zuerst müssen Sie Modell A laden und basierend auf dessen Schichten ein neues Modell erstellen. Wir verwenden dazu alle Schichten außer der Ausgabeschicht:

```
model_A = keras.models.load_model("my_model_A.h5")  
  
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])  
  
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Beachten Sie, dass sich `model_A` und `model_B_on_A` nun ein paar gemeinsame Schichten teilen. Trainieren Sie `model_B_on_A`, beeinflusst das auch `model_A`. Wollen Sie das vermeiden, müssen Sie `model_A klonen`, bevor Sie dessen Schichten wiederverwenden. Dazu übertragen Sie die Architektur von Modell A mit `clone_model()` und kopieren dann die Gewichte (da `clone_model()` das nicht tut):

```
model_A_clone = keras.models.clone_model(model_A)  
  
model_A_clone.set_weights(model_A.get_weights())
```

Jetzt können Sie `model_B_on_A` für Aufgabe B trainieren, aber da die neue Ausgabeschicht mit Zufallswerten initialisiert wurde, wird das zu großen Fehlern führen (zumindest während der ersten paar Epochen), und Sie erhalten große Fehlergradienten, die die wiederverwendeten Gradienten zerstören können. Um das zu vermeiden, können Sie die wiederverwendeten Schichten während der ersten Epochen einfrieren und der neuen Schicht etwas Zeit zum Erlernen vernünftiger Gewichte geben. Dazu setzen Sie das Attribut `trainable` jeder Schicht auf `False` und kompilieren das Modell:

```

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                      metrics=["accuracy"])

```



Sie müssen Ihr Modell immer kompilieren, nachdem Sie Schichten eingefroren oder aufgetaut haben.

Jetzt können Sie das Modell für ein paar Epochen trainieren, dann die wiederverwendeten Schichten auftauen (wodurch das Modell erneut kompiliert werden muss) und mit dem Training fortfahren, um die wiederverwendeten Schichten für Aufgabe B genauer abzustimmen. Nach dem Auftauen der Schichten ist es meist eine gute Idee, die Lernrate zu verringern, um erneut das Beschädigen der wiederverwendeten Gewichte zu vermeiden:

```

history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=1e-4) # Standard für lr ist 1e-2
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                      metrics=["accuracy"])

history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))

```

Und was ist dabei herausgekommen? Nun, die Testgenauigkeit des Modells liegt bei 99,25%, was heißtt, dass das Transfer Learning die Fehlerrate von 2,8% auf nahezu 0,7% verringert hat! Das ist ein Faktor von vier.

```

>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.06887910133600235, 0.9925]

```

Hat Sie das überzeugt? Sollte es aber nicht – ich habe geschummelt! Ich habe viele

Konfigurationen ausprobiert, bis ich eine fand, die eine starke Verbesserung brachte. Versuchen Sie, die Kategorien oder die Zufallswerte zu verändern, werden Sie feststellen, dass die Verbesserung im Allgemeinen klein ist, verschwindet oder sich sogar ins Negative verkehrt. Ich habe die Daten so lange gefoltert, bis sie gestanden haben. Sieht ein Artikel zu gut aus, sollten Sie misstrauisch werden: Eventuell hilft diese neue, schicke Technik gar nicht so viel (oder vielleicht macht sie die Ergebnisse sogar schlechter), aber die Autoren haben sehr viele Varianten ausprobiert und nur über die besten Ergebnisse berichtet (die möglicherweise durch pures Glück entstanden sind), ohne zu erwähnen, wie viele Fehlerversuche sie vorher hatten. Die meiste Zeit geschieht das gar nicht aus Böswilligkeit, aber es ist mit ein Grund dafür, dass so viele Ergebnisse in der Wissenschaft nie reproduziert werden können.

Warum habe ich geschummelt? Es zeigt sich, dass Transfer Learning mit kleinen dichten Netzen nicht gut funktioniert, vermutlich weil kleine Netze auch nur wenige Muster erlernen und dichte Netze sehr spezifische Muster lernen, die in anderen Aufgaben nicht unbedingt nützlich sind. Transfer Learning funktioniert am besten mit tiefen Convolutional Neural Networks, die eher Feature-Detektoren erlernen, die allgemeiner arbeiten (vor allem in den unteren Schichten). Wir werden uns das Transfer Learning in [Kapitel 14](#) nochmals anschauen und dabei die gerade behandelten Techniken einsetzen (und dann auch nicht schummeln, das verspreche ich!).

Unüberwachtes Vortrainieren

Nehmen wir 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 nicht möglich ist, können Sie immer noch ein *unüberwachtes Vortraining* durchführen (siehe [Abbildung 11-5](#)). Natürlich ist es oft günstig, ungelabelte Trainingsbeispiele zu erhalten, aber es ist teuer, sie mit Labels zu versehen. Wenn Ihnen also reichlich ungelabelte Trainingsdaten zur Verfügung stehen, können Sie versuchen, damit ein unüberwachtes Modell zu trainieren, wie zum Beispiel einen Autoencoder oder ein Generative Adversarial Network (siehe [Kapitel 17](#)). Dann können Sie die unteren Schichten des Autoencoder oder des GAN-Diskriminators wiederverwenden, die Ausgabeschicht für Ihre Aufgabe daraufsetzen und das so entstandene Netz mithilfe überwachten Lernens feinjustieren (also mit den gelabelten Trainingsbeispielen).

Diese Technik haben Geoffrey Hinton und sein Team 2006 genutzt, und sie hat zum Revival neuronaler Netze und zum Erfolg von Deep Learning geführt. Bis 2010 war unüberwachtes Vortrainieren – typischerweise mit Restricted Boltzmann Machines (RBMs, siehe [Anhang E](#)) – die Norm für tiefe Netze, und erst nachdem das Problem der verschwindenden Gradienten eingedämmt werden konnte, fand das Trainieren von DNNs rein mit überwachtem Lernen weitere Verbreitung. Unüberwachtes Vortrainieren (heutzutage meist nicht mehr mit RBMs, sondern eher mit Autoencodern oder GANs) ist immer noch eine gute Option, wenn Sie eine komplexe Aufgabe lösen müssen, kein ähnliches Modell wiederverwenden können und nur wenige gelabelte, aber viele ungelabelte Trainingsdaten haben.

In den frühen Tagen des Deep Learning war es schwierig, tiefe Modelle zu trainieren, daher wurde eine Technik namens *Greedy Layer-Wise Pretraining* (siehe [Abbildung 11-5](#)) genutzt.

Erst wurde ein unüberwachtes Modell mit einer einzelnen Schicht trainiert – meist ein RBM –, dann wurde diese Schicht eingefroren und eine weitere daraufgesetzt. Dann wurde das Modell wieder trainiert (eigentlich nur die neue Schicht), diese neue Schicht wieder eingefroren und so weiter. Heutzutage ist es viel einfacher: Die Leute trainieren im Allgemeinen das vollständige unüberwachte Modell in einem Durchlauf (in [Abbildung 11-5](#) beginnen Sie einfach bei Schritt 3) und verwenden dann Autoencoder oder GANs statt RBMs.

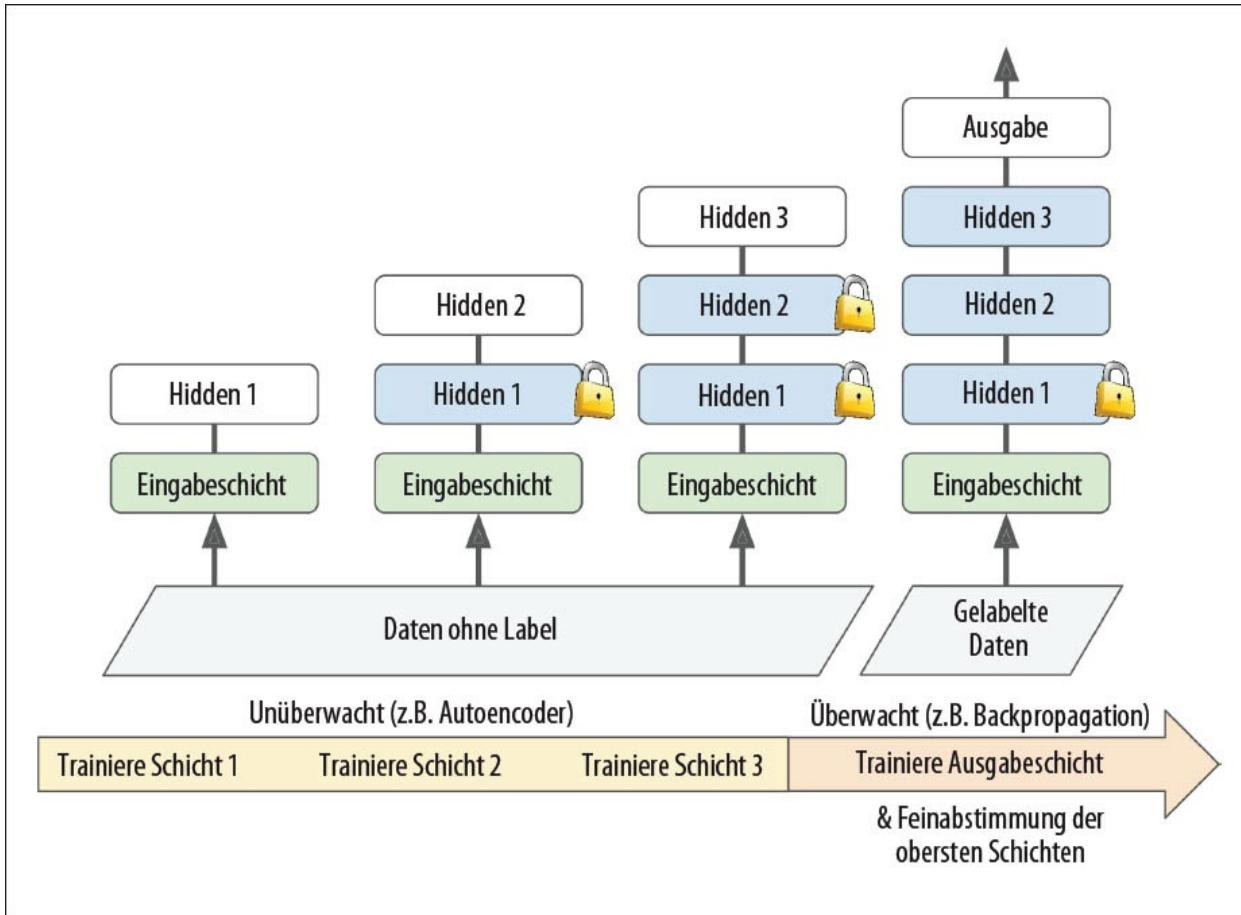


Abbildung 11-5: Beim unüberwachten Vortrainieren wird ein Modell mit den ungelabelten Daten (oder allen Daten) mithilfe einer unüberwachten Lerntechnik trainiert und dann für die eigentliche Aufgabe mit den gelabelten Aufgaben mit einer überwachten Lerntechnik im Detail angepasst – der unüberwachte Teil kann wie hier gezeigt eine Schicht nach der anderen oder direkt das ganze Modell trainieren.

Vortrainieren anhand einer Hilfsaufgabe

Haben Sie nicht ausreichend gelabelte Daten, stellen wir Ihnen abschließend noch 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 Netzes 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 Web 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.

Für die *natürliche Sprachverarbeitung* (Natural Language Processing, NLP) können Sie einen Korpus mit Millionen Textdokumenten herunterladen und daraus automatisch gelabelte Daten erzeugen. Sie könnten beispielsweise zufällig einzelne Wörter ausblenden und ein Modell trainieren, das vorhersagt, welches Wort fehlt (so sollte es zum Beispiel vorhersagen, dass das fehlende Wort im Satz »What ____ you saying?« wahrscheinlich »are« oder »were« ist). Können Sie ein Modell darin trainieren, bei dieser Aufgabe eine gute Leistung zu erreichen, wird es schon ziemlich viel über Sprache wissen, und Sie können es sicherlich für Ihre eigentliche Aufgabe wiederverwenden und es mit Ihren gelabelten Daten optimieren (mehr zu Vortrainingsaufgaben besprechen wir in [Kapitel 15](#)).

- ❖ *Selbstüberwachtes Lernen* heißt, dass Sie automatisch die Labels aus den Daten selbst erzeugen und dann ein Modell mit dem entstandenen »gelabelten« Datensatz mithilfe überwachter Lerntechniken trainieren. Da für diesen Ansatz kein Labeling durch Menschen erforderlich ist, lässt es sich am besten als Form des unüberwachten Lernens klassifizieren.

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 nutzen, die Aktivierungsfunktion sinnvoll zu wählen, Batchnormalisierung zu verwenden und Teile eines vortrainierten Netzes einzusetzen (eventuell gebaut für eine künstliche Aufgabe oder per unüberwachtes Lernen). 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- und Nadam-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 der von Boris Polyak im Jahr 1964 vorgeschlagenen *Momentum Optimization* (<https://homl.info/54>).¹³ Im Gegensatz dazu führt das Gradientenverfahren auf einer schiefen 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, der Gradient wird 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*, das zwischen 0 (hohe Reibung) und 1 (keine Reibung) liegen muss. Ein typisches Moment beträgt 0,9.

Formel 11-4: Moment-Algorithmus

1. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla J(\theta)$
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 (wir ignorieren dabei das Vorzeichen). Wenn beispielsweise $\beta = 0,9$ gilt,

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 Batchnormalisierung haben die oberen Schichten häufig Eingaben mit sehr unterschiedlichen Wertebereichen. In dieser Situation ist die Momentum Optimization sehr wertvoll. 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 Keras ist ein Klacks: Nutzen Sie einfach den SGD-Optimierer und setzen Sie dessen Hyperparameter `momentum`, dann lehnen Sie sich anschließend zurück und genießen!

```
optimizer = keras.optimizers.SGD(lr=0.001, 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 vorgestellt wurde (<https://homl.info/55>),¹⁴ 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 θ zu bestimmen, sondern ein wenig weiter vorwärts in Richtung des Moments $\theta + \beta m$ (siehe [Formel 11-5](#)).

Formel 11-5: Algorithmus des beschleunigten Gradienten nach Nesterov

1. $m \leftarrow \beta m - \eta \nabla_{\theta} J(\theta + \beta m)$
2. $\theta \leftarrow \theta + m$

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 $\theta + \beta m$).

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 wird beschleunigt.

NAG beschleunigt das Trainieren im Allgemeinen besser als die gewöhnliche Momentum Optimization. Um sie zu verwenden, setzen Sie einfach beim Erstellen eines SGD-Optimierers den Parameter `nesterov=True`:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

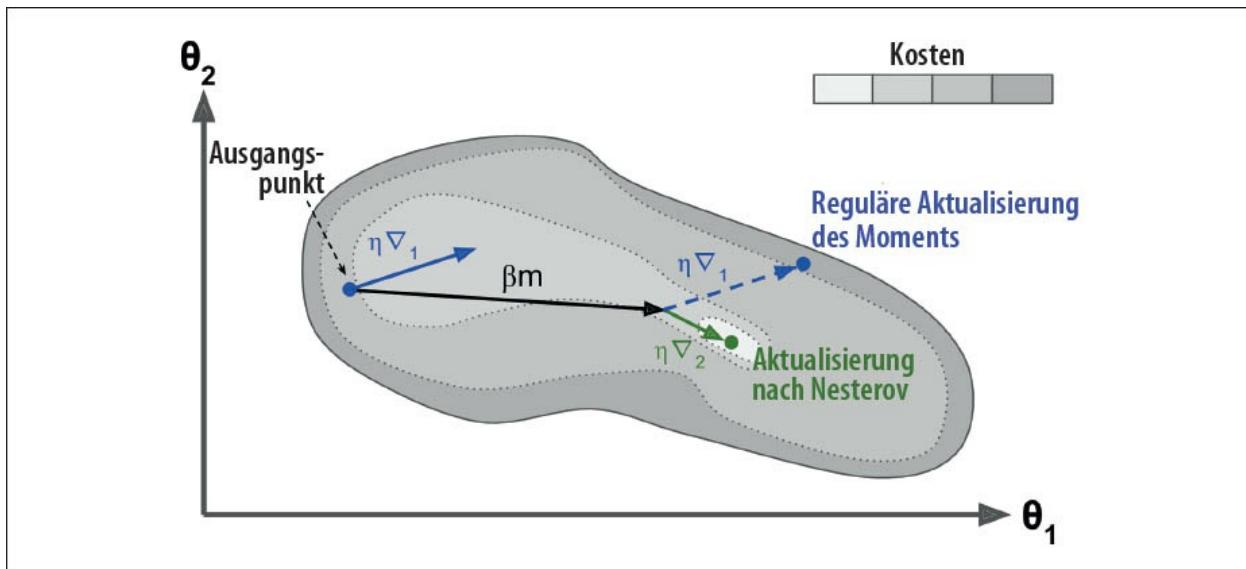


Abbildung 11-6: Gewöhnliche Momentum Optimization im Vergleich zur Variante nach Nesterov: Die normale wendet die berechneten Gradienten vor dem Momentum-Schritt an, die Variante erst danach.

AdaGrad

Betrachten wir noch einmal das Problem der länglichen Schüssel: Das Gradientenverfahren bewegt sich schnell entlang der steilsten Wand, zeigt dabei aber nicht direkt auf das globale Optimum 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 (<https://homl.info/56>)¹⁵ arbeitet genau so, er skaliert den Gradientenvektor entlang der steilsten Dimensionen herunter (siehe Formel 11-6):

Formel 11-6: AdaGrad-Algorithmus

1. $s \leftarrow s + \nabla_{\theta}J(\theta) \otimes \nabla_{\theta}J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta}J(\theta) \oslash \sqrt{s + \epsilon}$

Der erste Schritt akkumuliert das Quadrat der Gradienten in den Vektor s (das Symbol \otimes steht für die Multiplikation der einzelnen Elemente). Diese vektorisierte Form entspricht der

Berechnung von $s_i \leftarrow s_i + (\partial J(\boldsymbol{\theta}) / \partial \theta_i)^2$ 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 -Dimension steil, wird s_i von Iteration zu Iteration immer größer.

Der zweite Schritt ist beinahe identisch mit dem Gradientenverfahren, es gibt aber einen großen Unterschied: Der Gradientenvektor wird um den Faktor $\sqrt{s + \varepsilon}$ 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 $\theta_i \leftarrow \theta_i - \eta \partial J(\boldsymbol{\theta}) / \partial \theta_i / \sqrt{s_i + \varepsilon}$ 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 η nötig ist.

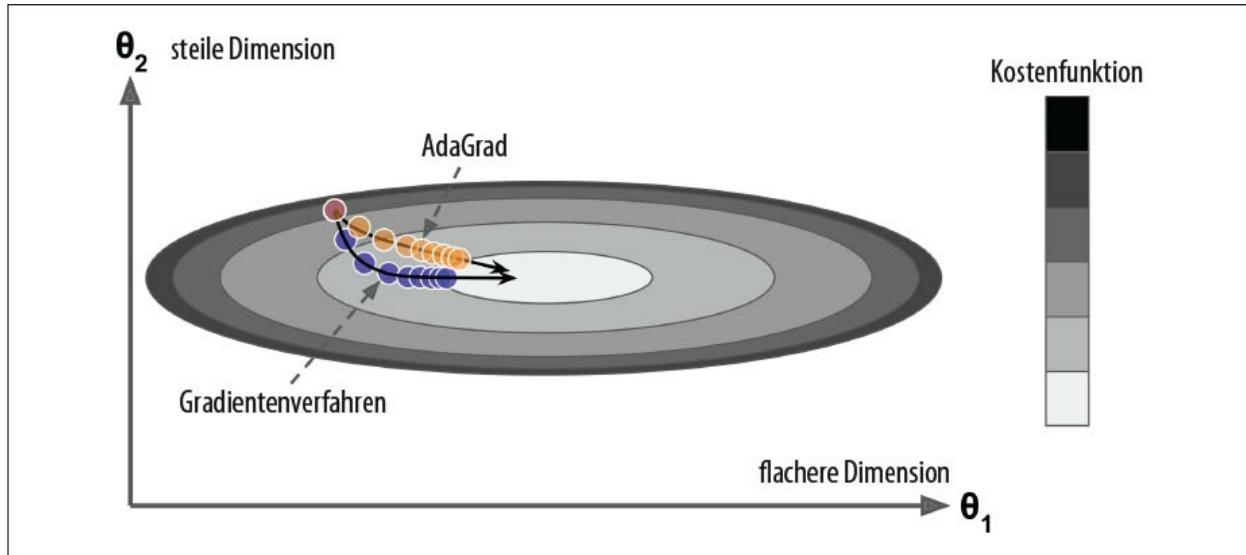


Abbildung 11-7: AdaGrad im Vergleich zum Gradientenverfahren – Ersterer kann frühzeitig auf das Optimum zeigen.

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 Keras einen Adagrad-Optimierer 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). Es ist aber trotzdem hilfreich, AdaGrad zu verstehen, um die anderen Optimierer zum Anpassen der Lernrate zu begreifen.

RMSProp

Wie wir gesehen haben, besteht bei AdaGrad das Risiko, dass er ein wenig zu schnell abbremst

und daher in manchen Fällen das globale Optimum nie erreicht. Der *RMSProp*-Algorithmus¹⁶ behebt 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. $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \varepsilon}$

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 Keras den Optimizer `RMSprop`:

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Außer bei sehr einfachen Aufgaben ist dieser Optimizer AdaGrad so gut wie immer überlegen. Deshalb war dies der von vielen Forschern bevorzugte Algorithmus, bis die Adam-Optimierung auf den Plan kam.

Adam-Optimierung

Adam (<https://homl.info/59>)¹⁷, 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 vorherigen Gradienten, und wie RMSProp merkt es sich auch den Durchschnitt der quadrierten Gradienten, beide unter exponentiellem Zerfall (siehe [Formel 11-8](#)).¹⁸

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. $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4. $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5. $\theta \leftarrow \theta + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$

In dieser Gleichung steht t 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^{-7} initialisiert. Dies sind die voreingestellten Werte der Klasse Adam (um genau zu sein, steht `epsilon` standardmäßig auf `None`, womit Keras `keras.backend.epsilon()` verwendet, was wiederum standardmäßig 10^{-7} liefert – Sie können es mit `keras.backend.set_epsilon()` ändern). So erstellen Sie einen Adam-Optimierer mit Keras:

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

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 zu nutzen ist.



Sollten Sie sich von all diesen verschiedenen Techniken mittlerweile etwas überfordert fühlen und sich fragen, wie Sie die richtige für Ihre Aufgabe auswählen, machen Sie sich keine Sorgen: Am Ende dieses Kapitels liefere ich Ihnen ein paar praktische Ratschläge.

Es lohnt sich noch, auf zwei Varianten von Adam hinzuweisen:

AdaMax

In Schritt 2 von [Formel 11-8](#) summiert Adam die Quadrate der Gradienten in \mathbf{s} auf (mit einer größeren Gewichtung auf die aktuelleren Gradienten). Ignorieren wir in Schritt 5 ϵ sowie die Schritte 3 und 4 (bei denen es sich nur um technische Details handelt), skaliert Adam die Parameteraktualisierungen mit der Quadratwurzel von \mathbf{s} . Kurz gesagt, skaliert Adam die Parameteraktualisierungen mithilfe der ℓ_2 -Norm der altersberücksichtigten Gradienten (die ℓ_2 -Norm ist die Quadratwurzel der Summe der Quadrate). AdaMax – im gleichen Artikel vorgestellt wie Adam – ersetzt die ℓ_2 -Norm durch die ℓ_∞ -Norm (eine schickere Ausdrucksweise für das Maximum). Genauer gesagt, ersetzt es Schritt 2 in [Formel 11-8](#) durch $\mathbf{s} \leftarrow \max(\beta_2 \mathbf{s}, \nabla_{\theta} J(\theta))$, lässt Schritt 4 weg und skaliert in Schritt 5 die Gradientenaktualisierungen um einen Faktor \mathbf{s} herunter, was gerade das Maximum der altersberücksichtigten Gradienten ist. In der Praxis ist AdaMax dadurch möglicherweise stabiler als Adam, aber das hängt vom Datensatz ab, und im Allgemeinen ist Adam schneller. Somit handelt es sich bei AdaMax letztlich um einen weiteren Optimierer, den Sie ausprobieren können, wenn Sie bei einer Aufgabe Probleme mit Adam haben.

Nadam

Beim Nadam-Optimierer handelt es sich um den Adam-Optimierer plus Nesterov-Trick, wodurch er oft etwas schneller als Adam konvergiert. In dem Artikel (<https://homl.info/nadam>), der diese Technik vorstellt,¹⁹ vergleicht der Forscher Timothy Dozat viele verschiedene Optimierer für diverse Aufgaben und stellt fest, dass Nadam im

Allgemeinen schneller als Adam ist, manchmal aber von RMSProp noch übertrumpft wird.

 Adaptive Optimierungsmethoden (einschließlich RMSProp, Adam und Nadam) sind oft wunderbar und konvergieren schnell hin zu einer guten Lösung. Aber ein Artikel aus dem Jahr 2017 (<https://homl.info/60>) von Ashia C. Wilson et al.²⁰ kam zu dem Schluss, dass sie bei manchen Datensätzen zu einer schwachen Verallgemeinerungsleistung führen. Sind Sie also mit der Leistung Ihres Modells unzufrieden, können Sie den klassischen beschleunigten Gradienten nach Nesterov ausprobieren – Ihr Datensatz reagiert eventuell nur allergisch auf adaptive Gradienten. Schauen Sie sich auch die aktuelle Forschung an, denn da passiert ziemlich viel.

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* arbeiten (die *Hessians*, bei denen es sich um die partiellen Ableitungen der Jacobians handelt). 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 2. 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 spärlches Modell in Betracht ziehen.

Eine einfache Möglichkeit hierzu ist, das Modell wie gewohnt zu trainieren und anschließend winzige Gewichte auf null zu setzen. Allerdings wird dies meist nicht zu einem sehr spärlchen Modell führen und dessen Leistung verschlechtern.

Eine andere Möglichkeit besteht darin, während des Trainierens eine starke ℓ_1 -Regularisierung anzuwenden (darauf kommen wir später noch zurück), da diese den Optimierer veranlasst, möglichst viele Gewichte auf null zu setzen (wie im Fall der in [Kapitel 4](#) besprochenen Lasso-Regression).

Erweisen sich diese Techniken als unzureichend, werfen Sie einen Blick auf das TensorFlow Model Optimization Toolkit (TF-MOT) (<https://homl.info/tfmot>), das eine aufgeräumte API hat, mit der Sie während des Trainings iterativ Verbindungen abhängig von deren Magnitude entfernen können.

In [Tabelle 11-2](#) werden alle bisher besprochenen Optimierer verglichen (* ist schlecht, ** ist durchschnittlich, und *** ist gut).

Tabelle 11-2: Vergleich von Optimierern

Klasse	Konvergenzgeschwindigkeit	Konvergenzqualität
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	* (stoppt zu früh)
Adagrad	***	
RMSprop	***	** oder ***
Adam	***	** oder ***
Nadam	***	** oder ***
AdaMax	***	** oder ***

Scheduling der Lernrate

Es ist sehr wichtig, eine gute Lernrate zu finden. 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 zum 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. Ist Ihre Rechenzeit begrenzt, müssen Sie das Trainieren unterbrechen, bevor es anständig konvergiert, und sich mit einer suboptimalen Lösung zufriedengeben (siehe [Abbildung 11-8](#)).

Wie in [Kapitel 10](#) besprochen, finden Sie eventuell eine gute Lernrate, indem Sie das Modell für ein paar Hundert Iterationen trainieren, dabei die Lernrate exponentiell von einem sehr kleinen zu einem sehr großen Wert wachsen lassen, sich dann die Lernkurve anschauen und dann eine Lernrate auswählen, die etwas unter dem Wert liegt, bei dem die Lernkurve wieder hochschießt. Dann können Sie Ihr Modell neu initialisieren und es mit dieser Lernrate trainieren.

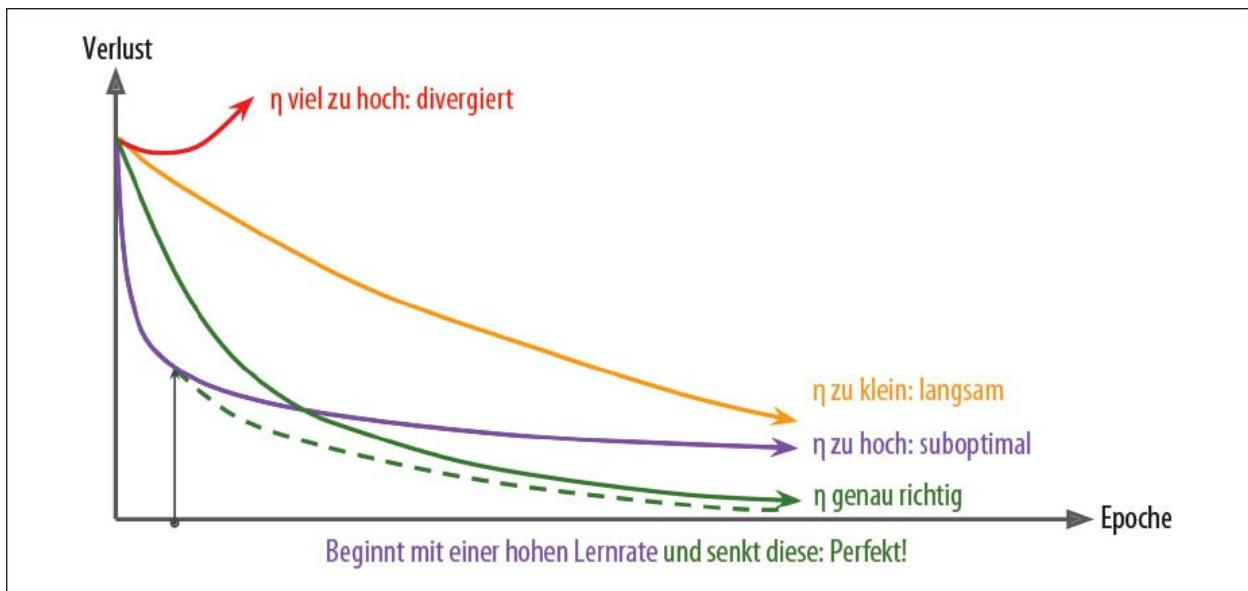


Abbildung 11-8: Lernkurven für unterschiedliche Lernraten η

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. Lohnenswert kann es auch sein, mit einer niedrigen Lernrate zu beginnen, sie zu steigern und dann wieder abfallen zu lassen. Diese Strategien bezeichnet man als *Scheduling der Lernrate* (wir haben den Begriff kurz in [Kapitel 4](#) erwähnt). Die häufigsten Strategien sind:

Power Scheduling

Setzen Sie die Lernrate auf eine Funktion der Iteration t : $\eta(t) = \eta_0 / (1 + t/s)^c$. Die initiale Lernrate η_0 , die Potenz c (meist mit dem Wert 1) und die Schritte s sind Hyperparameter. Nach s Schritten sinkt die Lernrate auf $\eta_0 / 2$. Nach s weiteren Schritten liegt sie bei $\eta_0 / 3$, dann sinkt sie auf $\eta_0 / 4$ und so weiter. Wie Sie sehen, sinkt dieser Schedule erst schnell und dann immer langsamer. Natürlich müssen Sie für das Power Scheduling sowohl η_0 wie auch s (und eventuell c) anpassen.

Exponentielles Scheduling

Legen Sie die Lernrate als Funktion der Iteration t fest: $\eta(t) = \eta_0 \cdot 0,1^{t/s}$. Die Lernrate wird damit nach und nach um einen Faktor von 10 alle s Schritte fallen. Während das Power Scheduling die Lernrate stärker und langsamer verringert, wird sie beim exponentiellen Scheduling weiterhin alle s Schritte um den Faktor 10 reduziert.

Vorgefertigte stückweise konstante Lernrate

Setzen Sie die Lernrate zu Beginn für eine gewisse Anzahl von Epochen auf einen festen Wert (zum Beispiel $\eta_0 = 0,1$ für 5 Epochen), dann für eine weitere Anzahl von Epochen auf einen kleineren Wert (zum Beispiel $\eta_1 = 0,001$ für 50 Epochen) und so weiter. Obwohl diese Lösung sehr gut funktionieren kann, 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.

1cycle Scheduling

Anders als bei den anderen Ansätzen beginnt das *1cycle Scheduling* (vorgestellt in einem Artikel (<https://homl.info/1cycle>) von Leslie Smith aus dem Jahr 2018²¹) damit, die initiale Lernrate η_0 zu erhöhen und bis zur Hälfte des Trainings linear bis zur Lernrate η_1 wachsen zu lassen. Dann reduziert es während der zweiten Hälfte des Trainings die Rate wieder linear bis η_0 , um sie ganz zum Schluss während der letzten paar Epochen um mehrere Größenordnungen abzusenken (immer noch linear). Die maximale Lernrate η_1 wird mit dem gleichen Ansatz ermittelt wie beim Finden der optimalen Lernrate, und die initiale Lernrate η_0 wird ungefähr um einen Faktor 10 kleiner ausgewählt. Beim Einsatz eines Momentum beginnen wir zuerst mit einem hohen Momentum (zum Beispiel 0,95), reduzieren es dann in der ersten Hälfte des Trainings (zum Beispiel linear auf 0,85) und bringen es dann wieder in der zweiten Hälfte zurück auf den maximalen Wert, wobei dieser bei den letzten paar Epochen beibehalten wird. Smith hat viele Experimente durchgeführt, die gezeigt haben, dass dieser Ansatz oft dazu in der Lage war, das Training deutlich zu beschleunigen und

eine bessere Leistung zu erreichen. So gelangte er beispielsweise mit dem beliebten CIFAR10-Bilddatensatz auf 91,9% Validierungsgenauigkeit in nur 100 Epochen – im Gegensatz zu 90,3% Genauigkeit nach 800 Epochen bei einem Standardvorgehen (mit der gleichen Architektur des neuronalen Netzes).

Ein Artikel (<https://homl.info/63>) aus dem Jahr 2013²² 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 abstimmen lässt und ein wenig schneller zur optimalen Lösung gelangt (es wird auch erwähnt, dass es sich leichter als Performance Scheduling implementieren ließ, aber in Keras sind beide Varianten einfach). Allerdings scheint der 1cycle-Ansatz noch schneller zu sein.

Ein Power Scheduling lässt sich in Keras sehr einfach implementieren: Setzen Sie beim Erstellen eines Optimierers den Hyperparameter decay:

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

Der decay ist das Inverse von s (die Anzahl an Schritten, nach denen die Lernrate durch eine weitere Einheit dividiert wurde). Keras geht davon aus, dass $c = 1$ ist.

Exponentielles Scheduling und eine stückweise konstante Lernrate lassen sich ebenfalls einfach umsetzen. Sie müssen zunächst eine Funktion definieren, die die aktuelle Epoche erwartet und die Lernrate zurückgibt. Implementieren wir beispielsweise das exponentielle Scheduling:

```
def exponential_decay_fn(epoch):  
    return 0.01 * 0.1**(epoch / 20)
```

Wollen Sie η_0 und s nicht fest angeben, können Sie auch eine Funktion schreiben, die eine konfigurierte Funktion zurückgibt:

```
def exponential_decay(lr0, s):  
    def exponential_decay_fn(epoch):  
        return lr0 * 0.1**(epoch / s)  
  
    return exponential_decay_fn  
  
exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Als Nächstes erstellen Sie einen Callback LearningRateScheduler, übergeben ihm die Schedule-Funktion und reichen diesen Callback an die Methode fit() weiter:

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
```

```
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

Der `LearningRateScheduler` wird das Attribut `learning_rate` des Optimierers zu Beginn jeder Epoche anpassen. Das reicht normalerweise aus, aber wenn die Lernrate häufiger angepasst werden soll – zum Beispiel bei jedem Schritt –, können Sie Ihren eigenen Callback schreiben (siehe den Abschnitt »Exponential Scheduling« des Notebooks). Das Aktualisieren der Lernrate bei jedem Schritt ist sinnvoll, wenn es viele Schritte pro Epoche gibt. Alternativ können Sie den gleich beschriebenen Ansatz `keras.optimizers.schedules` verfolgen.

Die `Schedule`-Funktion kann optional die aktuelle Lernrate als zweites Argument übernehmen. So multipliziert beispielsweise die folgende `Schedule`-Funktion die vorherige Lernrate mit $0,1^{1/20}$, was zum gleichen exponentiellen Abfall führt (der allerdings mit Epoche 0 statt mit Epoche 1 beginnt):

```
def exponential_decay_fn(epoch, lr):  
    return lr * 0.1**(1 / 20)
```

Diese Implementierung geht von der initialen Lernrate des Optimierers aus (im Gegensatz zur vorherigen Implementierung) – stellen Sie daher sicher, dass Sie sie korrekt setzen.

Sichern Sie ein Modell, werden auch der Optimierer und seine Lernrate mit gesichert. Mit dieser neuen `Schedule`-Funktion können Sie daher einfach ein trainiertes Modell laden und ohne Probleme mit dem Training dort fortfahren, wo Sie aufgehört haben. Nutzt Ihre `Schedule`-Funktion allerdings das Argument `epoch`, wird es komplizierter: Die Epoche wird nicht mitgesichert und bei jedem Aufruf der Methode `fit()` auf 0 zurückgesetzt. Wollen Sie bei einem Modell dort mit dem Training fortfahren, wo Sie aufgehört haben, kann das zu einer sehr großen Lernrate führen, was vermutlich die Gewichte Ihres Modells beschädigt. Eine Lösung ist, das Argument `initial_epoch` der Methode `fit()` manuell zu setzen, sodass `epoch` direkt mit dem richtigen Wert loslegt.

Für eine stückweise konstante Lernrate können Sie eine `Schedule`-Funktion wie die folgende nutzen (wie zuvor können Sie bei Bedarf eine generischere Funktion definieren, siehe den Abschnitt »Piecewise Constant Scheduling« des Notebooks), dann einen `Callback` `LearningRateScheduler` mit dieser Funktion erstellen und ihn an die Methode `fit()` übergeben – so wie wir es beim exponentiellen Scheduling gemacht haben:

```
def piecewise_constant_fn(epoch):  
    if epoch < 5:  
        return 0.01  
    elif epoch < 15:  
        return 0.005
```

```
    else:  
        return 0.001
```

Für ein Performance Scheduling verwenden Sie den Callback ReduceLROnPlateau. Übergeben Sie beispielsweise den folgenden Callback an die Methode `fit()`, wird dieser die Lernrate immer dann mit 0,5 multiplizieren, wenn sich der beste Validierungsverlust fünf aufeinanderfolgende Epochen lang nicht verbessert (es stehen noch andere Optionen zur Verfügung – werfen Sie einen Blick in die Dokumentation):

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

Und schließlich bietet tf.keras noch eine alternative Möglichkeit an, ein Lernraten-Scheduling zu implementieren: Definieren Sie die Lernrate über einen der in `keras.optimizers.schedules` verfügbaren Scheduler und übergeben Sie sie dann an einen Optimizer. Dadurch wird die Lernrate bei jedem Schritt statt nur bei jeder Epoche aktualisiert. So implementieren Sie zum Beispiel den gleichen exponentiellen Schedule wie bei der weiter oben definierten Funktion `exponential_decay_fn()`:

```
s = 20 * len(X_train) // 32 # Anzahl Schritte in 20 Epochen (Batchgröße 32)  
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)  
optimizer = keras.optimizers.SGD(learning_rate)
```

Das ist nett und einfach, und wenn Sie das Modell sichern, werden auch die Lernrate und deren Schedule (einschließlich dessen Status) mit abgespeichert. Dieses Vorgehen ist aber nicht Teil der Keras-API, sondern nur in tf.keras vorhanden.

Bei 1cycle ist die Implementierung auch nicht besonders schwierig: Erstellen Sie einfach einen eigenen Callback, der die Lernrate bei jeder Iteration anpasst (Sie können dafür `self.model.optimizer.lr` ändern). Im Abschnitt »1Cycle Scheduling« des Notebooks finden Sie ein Beispiel dazu.

Zusammengefasst, lässt sich sagen, dass exponentieller Abfall, Performance Scheduling und 1cycle die Konvergenz deutlich beschleunigen können. Probieren Sie es daher einmal aus!

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

Mit Tausenden Parametern können Sie den ganzen Zoo fitten. 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.

Wir haben schon eine der besten Regularisierungstechniken in [Kapitel 10](#) implementiert – das Early Stopping. Und auch wenn die Batchnormalisierung dazu gedacht war, das Problem der instabilen Gradienten zu lösen, verhält sie sich zugleich wie ein ziemlich guter Regularisierer. In diesem Abschnitt werden wir einige der beliebtesten Regularisierungstechniken für neuronale Netze vorstellen: ℓ_1 - und ℓ_2 -Regularisierung, Drop-out und Max-Norm-Regularisierung.

ℓ_1 - und ℓ_2 -Regularisierung

Wie bei den einfachen linearen Modellen in [Kapitel 4](#) können Sie mit der ℓ_2 -Regularisierung den Gewichten der Verbindungen eines neuronalen Netzes (aber nicht den Bias-Termen) Beschränkungen auferlegen und/oder die ℓ_1 -Regularisierung verwenden, wenn Sie ein Sparse-Modell nutzen wollen (mit vielen Gewichten gleich 0). So wenden Sie die ℓ_2 -Regularisierung auf die Verbindungsgewichte einer Keras-Schicht mit einem Regularisierungsfaktor von 0,01 an:

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

Die Funktion `l2()` gibt einen Regularisierer zurück, der während des Trainings bei jedem Schritt aufgerufen wird, um den Regularisierungsverlust zu berechnen. Dieser wird dann dem Gesamtverlust zugeschlagen. Nicht ganz unerwartet können Sie mit `keras.regularizers.l1()` eine ℓ_1 -Regularisierung umsetzen – wollen Sie sowohl ℓ_1 - wie auch ℓ_2 -Regularisierung haben, verwenden Sie `keras.regularizers. l1_l2()` (und geben beide Regularisierungsfaktoren an).

Da Sie normalerweise den gleichen Regularisierer auf alle Schichten Ihres Netzes anwenden wollen – so wie Sie die gleiche Aktivierungsfunktion und Initialisierungsstrategie für alle verborgenen Schichten nutzen –, werden Sie feststellen, dass Sie immer wieder die gleichen Argumente nutzen. Das macht den Code hässlich und fehleranfällig. Um das zu vermeiden, können Sie versuchen, Ihren Code so zu refaktorieren, dass Schleifen zum Einsatz kommen. Eine andere Option ist die Python-Funktion `functools.partial()`, mit der Sie einen dünnen Wrapper für alles Aufrufbare mit Standardargumenten erzeugen können:

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    # ...]
```

```

        keras.layers.Flatten(input_shape=[28, 28]),
        RegularizedDense(300),
        RegularizedDense(100),
        RegularizedDense(10, activation="softmax",
                         kernel_initializer="glorot_uniform")
    )

```

Drop-out

Eine der bei Deep-Learning-Netzen beliebtesten Regularisierungstechniken ist *Drop-out*. Sie wurde von Geoffrey Hinton im Jahr 2012 vorgeschlagen (<https://homl.info/64>)²³ und in einem Artikel (<https://homl.info/65>)²⁴ 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 ein- bis zweiprozentigen 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 10% bis 50% gesetzt: näher an 20 bis 30% in rekurrenten neuronalen Netzen (siehe [Kapitel 15](#)), näher an 40 bis 50% in Convolutional Neural Networks (siehe [Kapitel 14](#)). Nach dem Trainieren werden keine Neuronen mehr ausgelassen. Und das ist auch schon alles (bis auf einige technische Details, denen wir uns gleich widmen).

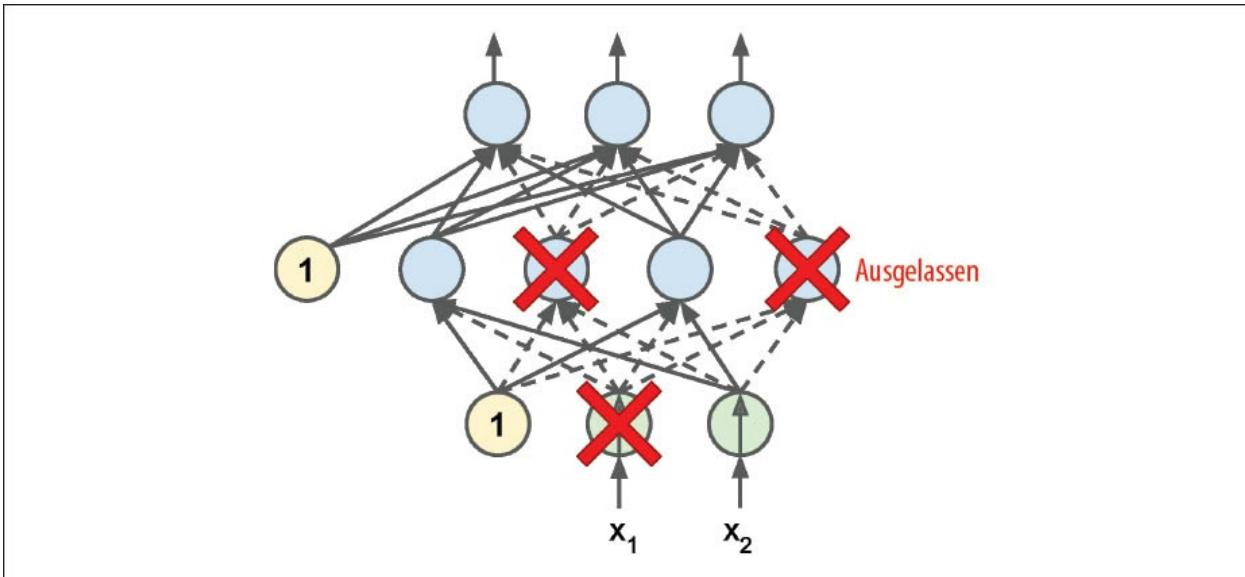


Abbildung 11-9: Bei der Drop-out-Regularisierung wird bei jeder Trainingsiteration eine zufällige Untergruppe aller Neuronen in einer oder mehreren Schichten – außer der Ausgabeschicht – »gedroppt«; diese Neuronen geben in dieser Iteration 0 zurück (dargestellt durch die gestrichelten Pfeile).

Es ist auf den ersten Blick ein wenig überraschend, dass diese destruktive 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 Unternehmen funktionieren würde, aber bei neuronalen Netzen funktioniert sie mit Sicherheit. Mit Dropout 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 10.000 durchgeföhrten Trainingsschritten haben Sie 10.000 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 dennoch alle unterschiedlich. Das am Ende erhaltene neuronale Netz lässt sich als Ensemble aus all diesen

kleineren Netzen ansehen.



In der Praxis können Sie Drop-out normalerweise nur in den obersten einen bis drei Schichten anwenden (abgesehen von der Ausgabeschicht).

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 Keras zu implementieren, können Sie die Schicht `keras.layers.Dropout` verwenden. Beim Trainieren lässt sie zufällig einige Neuronen aus (setzt diese auf 0) und teilt die verbliebenen Eingaben durch die keep-Wahrscheinlichkeit. Nach dem Trainieren tut diese Funktion überhaupt nichts, sie gibt die Eingaben einfach an die nächste Schicht weiter. Der folgende Code wendet die Regularisierung mittels Drop-out vor jeder Dense-Schicht an und nutzt dabei eine Drop-out-Rate von 0,2:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

- ☞ Da der Drop-out nur während des Trainings aktiv ist, kann ein Vergleich von Trainings- und Validierungsverlust in die Irre führen. Insbesondere kann ein Modell den Trainingsdatensatz overfitten und trotzdem gleiche Verluste bei Training und Validierung besitzen. Stellen Sie also sicher, dass Sie den Trainingsverlust ohne Drop-out bestimmen (zum Beispiel nach dem Training).

Wenn Sie Overfitting beobachten, können Sie die Drop-out-Rate erhöhen. Liegt dagegen ein Underfitting der Trainingsdaten vor, sollten Sie die Drop-out-Rate senken. Bei großen Schichten

hilft das Erhöhen der Drop-out-Rate ebenfalls und bei kleinen Schichten das Verringern. Außerdem nutzen viele führende Architekturen Drop-out nach der letzten verborgenen Schicht, was Sie auch ausprobieren können, wenn ein vollständiger Drop-out zu stark ist.

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.

Wollen Sie ein selbstdnormalisierendes Netz, das auf der SELU-Aktivierungsfunktion basiert (wie weiter oben besprochen), regularisieren, sollten Sie *Alpha-Drop-out* verwenden: Dabei handelt es sich um eine Variante des Drop-out, die den Mittelwert und die Standardabweichung der Eingaben beibehält (sie wurde im gleichen Artikel wie SELU vorgestellt, da das reguläre Drop-out die Selbstdnormalisierung zerstören würden).

Monte-Carlo-(MC-)Drop-out

Im Jahr 2016 hat ein Artikel (<https://homl.info/mcdropout>) von Yarin Gal und Zoubin Ghahramani²⁵ ein paar weitere Gründe für Drop-out geliefert:

- Der Artikel zeigte eine stabile Verbindung zwischen Drop-out-Netzwerken (also neuronalen Netzen mit einer Dropout-Schicht vor jeder Gewichtsschicht) und der Approximate Bayesian Inference auf, womit das Drop-out eine solide mathematische Begründung erhält.²⁶
- Die Autoren haben eine leistungsfähige Technik namens *MC-Drop-out* vorgestellt, die die Leistung eines jeden trainierten Drop-out-Modells verbessern kann, ohne es neu trainieren oder überhaupt anpassen zu müssen, die Unsicherheit des Modells verbessert und zudem auch noch ausgesprochen einfach zu implementieren ist.

Wenn Ihnen das alles zu schön vorkommt, schauen Sie sich den folgenden Code an. Dabei handelt es sich um die vollständige Implementierung von MC-Drop-out, die das Drop-out-Modell verbessert, ohne es erneut zu trainieren:

```
y_probas = np.stack([model(X_test_scaled, training=True)
                      for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

Wir machen nur 100 Vorhersagen über den Testdatensatz, setzen `training=True`, um sicherzustellen, dass die Dropout-Schicht aktiv ist, und stacken die Vorhersagen. Da der Drop-out aktiv ist, werden alle Vorhersagen anders sein. Denken Sie daran, dass `predict()` eine Matrix mit einer Zeile pro Instanz und einer Spalte pro Kategorie zurückgibt. Da es 10.000 Instanzen im Testdatensatz und 10 Kategorien gibt, hat diese Matrix die Form [10000, 10]. Wir stacken 100 dieser Matrizen, womit `y_probas` ein Array der Form [100, 10000, 10] wird. Nachdem wir den Mittelwert über die erste Dimension ermittelt haben (`axis=0`), bekommen wir `y_proba` – ein Array der Form [10000, 10] –, wie bei einer einzelnen Vorhersage. Das ist alles! Das Bilden des Mittelwerts über mehrere Vorhersagen mit aktivem Drop-out liefert uns eine

Monte-Carlo-Schätzung, die im Allgemeinen zuverlässiger als das Ergebnis einer einzelnen Schätzung mit abgeschaltetem Drop-out ist. Schauen wir uns beispielsweise die Vorhersage des Modells für die erste Instanz im Testdatensatz mit abgeschaltetem Drop-out an:

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.99]], 
      dtype=float32)
```

Das Modell scheint sich sehr sicher zu sein, dass dieses Bild zu Kategorie 9 (Ankle Boot) gehört. Sollten Sie dem vertrauen? Sind Zweifel angebracht? Vergleichen Sie das mit den Vorhersagen bei aktivem Drop-out:

```
>>> np.round(y_probas[:, :1], 2)
array([[[0. , 0. , 0. , 0. , 0. , 0.14, 0. , 0.17, 0. , 0.68]],
       [[0. , 0. , 0. , 0. , 0. , 0.16, 0. , 0.2 , 0. , 0.64]],
       [[0. , 0. , 0. , 0. , 0. , 0.02, 0. , 0.01, 0. , 0.97]],
       [...]])
```

Das sieht schon anders aus: Anscheinend ist sich das Modell bei aktiviertem Dropout nicht mehr so sicher. Es scheint immer noch Kategorie 9 zu bevorzugen, kann sich manchmal aber auch die Kategorie 5 (Sandal) oder 7 (Sneaker) vorstellen, was nicht so abwegig ist – es gehört alles zur Fußbekleidung. Bilden wir den Mittelwert über die erste Dimension, erhalten wir die folgenden MC-Drop-out-Vorhersagen:

```
>>> np.round(y_proba[:1], 2)
array([[0. , 0. , 0. , 0. , 0. , 0.22, 0. , 0.16, 0. , 0.62]], 
      dtype=float32)
```

Das Modell geht immer noch davon aus, dass dieses Bild zu Kategorie 9 gehört, aber nur mit einer Sicherheit von 62%, was deutlich vernünftiger klingt als 99%. Zudem ist es gut, zu wissen, welche anderen Kategorien noch in Betracht kämen. Sie können auch einen Blick auf die Standardabweichung der Wahrscheinlichkeitsschätzungen (<https://xkcd.com/2110>) werfen:

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:1], 2)
array([[0. , 0. , 0. , 0. , 0. , 0.28, 0. , 0.21, 0.02, 0.32]], 
      dtype=float32)
```

Es scheint eine ziemliche Varianz in den Wahrscheinlichkeitsschätzungen zu geben: Würden Sie ein System in einem Bereich mit echten Risiken bauen (zum Beispiel ein medizinisches oder Finanzsystem), sollten Sie solch einer unsicheren Voraussage besser mit außerordentlicher Vorsicht begegnen – und auf keinen Fall wie eine Voraussage mit 99%. Zudem hat die Modellgenauigkeit einen kleinen Schub von 86,8% auf 86,9% erhalten:

```
>>> accuracy = np.sum(y_pred == y_test) / len(y_test)  
>>> accuracy  
0.8694
```



Die Anzahl der genutzten Monte-Carlo-Samples (in diesem Beispiel 100) ist ein Hyperparameter, den Sie anpassen können. Je höher er ist, desto genauer werden die Vorhersagen und deren Unsicherheitsschätzungen. Aber wenn Sie den Wert verdoppeln, wird sich auch die Inferenzzeit verdoppeln. Und über einer gewissen Anzahl von Samples werden Sie nur wenig weitere Verbesserung erreichen. Ihre Aufgabe ist es daher, abhängig von Ihrer Anwendung das richtige Verhältnis zwischen Latenz und Genauigkeit zu ermitteln.

Enthält Ihr Modell andere Schichten, die sich während des Trainings besonders verhalten (wie zum Beispiel BatchNormalization-Schichten), sollten Sie den Trainingsmodus nicht wie von uns durchgeführt erzwingen. Stattdessen sollten Sie die Dropout-Schichten durch die folgende MCDropout-Klasse ersetzen:²⁷

```
class MCDropout(keras.layers.Dropout):  
  
    def call(self, inputs):  
  
        return super().call(inputs, training=True)
```

Dabei erstellen wir einfach eine Unterklasse der Dropout-Schicht und überschreiben die Methode `call()`, um ihr Argument `training` fest auf `True` zu setzen (siehe [Kapitel 12](#)). Genauso könnten Sie eine Klasse `MCAphaDropout` erzeugen, indem Sie eine Unterklasse von `AlphaDropout` anlegen. Erstellen Sie ein Modell von Grund auf neu, müssen Sie nur `MCDropout` statt `Dropout` verwenden. Haben Sie aber bereits ein Modell, das mit `Dropout` trainiert wurde, müssen Sie ein neues Modell erstellen, das mit dem bestehenden identisch ist mit Ausnahme der Dropout-Schichten, die nun zu `MCDropout`-Schichten werden. Dann kopieren Sie die Gewichte des bestehenden Modells in Ihr neues Modell.

Kurz gesagt, ist MC-Drop-out eine fantastische Technik, die Drop,out-Modelle besser macht und bessere Unsicherheitsvoraussagen ermöglicht. Und da es während des Trainings ein ganz normales Drop,out ist, verhält es sich auch wie ein Regularisierer.

Max-Norm-Regularisierung

Eine weitere Regularisierungstechnik, die bei neuronalen Netzen gerne zur Anwendung kommt, nennt sich *Max-Norm-Regularisierung*: Für jedes Neuron werden die Gewichte w der

eingehenden Verbindungen so beschränkt, dass $\| \mathbf{w} \|_2 \leq r$, wobei r der Max-Norm-Hyperparameter und $\| \mathbf{w} \|_2$ die ℓ_2 -Norm ist.

Die Max-Norm-Regularisierung fügt der Gesamt-Verlustfunktion keinen Regularisierungsverlust hinzu. Stattdessen wird sie meist implementiert, indem $\| \mathbf{w} \|_2$ nach jedem Trainingsschritt berechnet und \mathbf{w} bei Bedarf neu skaliert wird ($\mathbf{w} \leftarrow \frac{\mathbf{w}}{\| \mathbf{w} \|_2}$).

Ein Verringern von r verringert die Regularisierungsstärke und hilft dabei, ein Overfitting zu reduzieren. Die Max-Norm-Regularisierung kann zudem dazu beitragen, die Probleme mit instabilen Gradienten zu dämpfen (wenn Sie keine Batchnormalisierung einsetzen).

Um die Max-Norm-Regularisierung in Keras zu implementieren, setzen Sie das Argument `kernel_constraint` jeder verborgenen Schicht auf einen Constraint `max_norm()` mit dem passenden Max-Wert, zum Beispiel so:

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
                   kernel_constraint=keras.constraints.max_norm(1.))
```

Nach jeder Trainingsiteration wird die Methode `fit()` des Modells das von `max_norm()` zurückgegebene Objekt aufrufen, dabei die Gewichte der Schicht übergeben und die umskalierten Gewichte zurückerhalten, die dann die Gewichte der Schicht ersetzen. Wie Sie in [Kapitel 12](#) sehen werden, können Sie bei Bedarf Ihre eigenen Constraint-Funktionen definieren und als `kernel_constraint` einsetzen. Auch können Sie die Bias-Terme beschränken, indem Sie das Argument `bias_constraint` setzen.

Die Funktion `max_norm()` besitzt ein Argument `axis`, das standardmäßig den Wert 0 hat. Eine Dense-Schicht besitzt im Allgemeinen Gewichte der Form [Anzahl Eingaben, Anzahl Neuronen], daher sorgt `axis=0` dafür, dass der Max-Norm-Constraint unabhängig auf den Gewichtsvektor jedes Neurons angewendet wird. Wollen Sie Max-Norm mit Convolutional-Schichten verwenden (siehe [Kapitel 14](#)), achten Sie darauf, das Argument `axis` des Constraints `max_norm()` passend zu setzen (normalerweise `axis=[0, 1, 2]`).

Zusammenfassung und praktische Tipps

In diesem Kapitel haben wir eine Vielzahl an Techniken besprochen. Sie fragen sich vielleicht, welche davon Sie verwenden sollen. Das hängt von der Aufgabe ab, und es gibt daher noch keinen klaren Konsens, aber ich habe festgestellt, dass die in [Tabelle 11-3](#) aufgeführte Konfiguration in den meisten Fällen gut funktioniert, ohne dass allzu viele Hyperparameter angepasst werden müssen. Betrachten Sie diese Standardwerte jedoch nicht als feste Vorgaben!

Tabelle 11-3: Standardkonfiguration eines DNN

Hyperparameter	Standardwert
Kernelinitialisierer	Initialisierung nach He
Aktivierungsfunktion	ELU
Normalisierung	keine, wenn Shallow; Batchnormalisierung, wenn Deep

Regularisierung	Early Stopping (+ ℓ_2 -Regularisierung bei Bedarf)
Optimierer	Momentum-Optimierer (oder RMSProp oder Nadam)
Scheduler für die Lernrate	1cycle

Handelt es sich beim Netz um eine simple Aufeinanderfolge von dichten Schichten, kann es sich selbst normalisieren, und Sie sollten stattdessen die Konfiguration in [Tabelle 11-4](#) verwenden.

Tabelle 11-4: DNN-Konfiguration für ein selbstnormalisierendes Netz

Hyperparameter	Standardwert
Kernelinitialisierer	LeCun-Initialisierung
Aktivierungsfunktion	SELU
Normalisierung	keine (Selbstnormalisierung)
Regularisierung	Alpha-Drop-out bei Bedarf
Optimierer	Momentum-Optimierer (oder RMSProp oder Nadam)
Scheduler für die Lernrate	1cycle

Vergessen Sie nicht, die Eingangsmerkmale zu normalisieren! Sie sollten auch versuchen, Teile eines vortrainierten neuronalen Netzes zu verwenden, wenn Sie eines finden, das ein ähnliches Problem löst, oder unüberwachtes Vortraining zu nutzen, wenn Sie viele ungelabelte Daten haben, oder ein Vortraining für eine künstliche Aufgabe einzusetzen, wenn Sie viele gelabelte Daten für eine ähnliche Aufgabe besitzen.

Während die obigen Richtlinien die meisten Fälle abdecken sollten, gibt es ein paar Ausnahmen:

- 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 das TensorFlow Model Optimization Toolkit einsetzen. Das wird die Selbstnormalisierung zerstören, daher sollten Sie in diesem Fall die Standardkonfiguration verwenden.
- Benötigen Sie ein Modell mit geringer Latenz (also eines, das blitzschnell Vorhersagen trifft), müssen Sie eventuell die Anzahl der Schichten reduzieren, die Batchnormalisierungsschichten mit den vorherigen Schichten verbinden und eventuell eine schnellere Aktivierungsfunktion wie Leaky ReLU oder einfach ReLU einsetzen. Es hilft auch, um ein spärliches Modell zu nutzen. Und schließlich können Sie die Gleitkommagenaugigkeit von 32 Bit auf 16 oder sogar 8 Bit verringern (siehe »Ein Modell auf ein Mobile oder Embedded Device deployen«). Schauen Sie sich außerdem TF-MOT an.
- Bauen Sie eine Anwendung in einem risikanteren Umfeld oder ist die Inferenzlatenz für Sie nicht sehr wichtig, können Sie MC-Drop-out zum Verbessern der Performance einsetzen, um zuverlässigere Wahrscheinlichkeitsschätzungen und Unsicherheitsschätzungen zu erhalten.

Mit diesen Richtlinien sind Sie bereit, sehr tiefe Netze zu trainieren! Ich hoffe, Sie sind jetzt davon überzeugt, dass Sie mit Keras ziemlich weit kommen. Es kann aber trotzdem passieren, dass Sie noch mehr Kontrolle benötigen – zum Beispiel wenn Sie eine eigene Verlustfunktion schreiben oder den Trainingsalgorithmus anpassen wollen. Für solche Fälle werden Sie auf die auf niedrigerer Ebene arbeitende API von TensorFlow zurückgreifen müssen, die Sie im

nächsten Kapitel kennenlernen.

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 SELU-Aktivierungsfunktion gegenüber ReLU.
4. In welchen Fällen würden Sie die folgenden Aktivierungsfunktionen verwenden: SELU, Leaky ReLU (und Varianten), ReLU, tanh, logistische und Softmax?
5. Was passiert, wenn Sie beim Verwenden des SGD-Optimierers den Hyperparameter `momentum` zu nah an 1 setzen (z.B. 0,99999)?
6. Nennen Sie drei Möglichkeiten, um ein dünn besetztes Modell zu erstellen.
7. Wird das Trainieren durch Drop-out langsamer? Wird die Inferenz langsamer (die Vorhersage auf neuen Datenpunkten)? Was ist mit MC-Drop-out?
8. Üben Sie das Trainieren eines Deep-Learning-Netzwerks mit dem CIFAR-10-Bilddatensatz:
 - a. Erstellen Sie ein DNN mit 20 verborgenen Schichten aus jeweils 100 Neuronen (das ist zu viel, aber darum geht es in dieser Übung). Verwenden Sie die Initialisierung nach He und ELU als Aktivierungsfunktion.
 - b. Trainieren Sie das Netz mit Nadam-Optimierung und Early Stopping auf den CIFAR10-Daten. Sie können sie über `keras.datasets.cifar10.load_data()` laden. Es besteht aus 60.000 Farbbildern mit jeweils 32×32 Pixeln (50.000 zum Trainieren, 10.000 zum Testen) mit 10 Kategorien, daher benötigen Sie eine Softmax-Ausgabeschicht mit 10 Neuronen. Denken Sie daran, jedes Mal nach der richtigen Lernrate zu suchen, wenn Sie die Architektur oder die Hyperparameter des Modells ändern.
 - c. Jetzt versuchen Sie, Batchnormalisierung hinzuzufügen. Vergleichen Sie die Lernkurven: Konvergiert sie schneller als zuvor? Erzeugt sie ein besseres Modell? Wie wirkt sich das auf die Trainingsgeschwindigkeit aus?
 - d. Versuchen Sie, die Batchnormalisierung durch SELU zu ersetzen, und nehmen Sie die notwendigen Anpassungen vor, um sicherzustellen, dass sich das Netz selbst normalisiert (standardisieren Sie also die Eingabefeatures, verwenden Sie die LeCun-Initialisierung, achten Sie darauf, dass das DNN nur eine Folge dichter Schichten enthält, und so weiter).
 - e. Versuchen Sie, das Modell per Alpha-Drop-out zu regulieren. Schauen Sie dann, ob Sie mithilfe von MC-Drop-out ohne ein erneutes Trainieren des Modells eine bessere Genauigkeit erhalten.
 - f. Trainieren Sie Ihr Modell erneut mit dem 1cycle-Scheduling und prüfen Sie, ob das die Trainingsgeschwindigkeit und die Modellgenauigkeit verbessert.

Lösungen zu diesen Übungen finden Sie in [Anhang A](#).

Eigene Modelle und Training mit TensorFlow

Bisher haben wir nur die High-Level-API `tf.keras` von TensorFlow genutzt, aber damit sind wir schon ziemlich weit gekommen: Wir haben neuronale Netze unterschiedlicher Architektur gebaut – unter anderem Regressions- und Klassifikationsnetze, Wide-&-Deep-Netze und selbstnormalisierende Netze –, alle möglichen Techniken angewendet, wie zum Beispiel Batchnormalisierung, Drop-out und Lernraten-Schedules. Tatsächlich werden Sie für 95% der Anwendungsfälle, mit denen Sie zu tun haben, nichts anderes als `tf.keras` benötigen (und `tf.data`, siehe [Kapitel 13](#)). Aber jetzt ist es an der Zeit, tiefer in TensorFlow einzusteigen und sich dessen Low-Level-Python-API (<https://homl.info/tf2api>) anzuschauen. Das wird nützlich sein, wenn Sie mehr Kontrolle brauchen, um eigene Verlustfunktionen, Metriken, Schichten, Modelle, Initialisierer, Regularisierer, Gewichts-Constraints und anderes zu schreiben. Sie benötigen eventuell sogar die vollständige Kontrolle über die Trainingsschleife selbst, zum Beispiel um besondere Transformationen oder Constraints auf die Gradienten anzuwenden (über das Beschneiden hinaus) oder um verschiedene Optimierer für unterschiedliche Teile des Netzes einzusetzen. Wir werden all diese Fälle behandeln und uns auch noch anschauen, wie Sie Ihre eigenen Modelle und Trainingsalgorithmen mithilfe des Automatic-Graph-Generation-Features von TensorFlow verbessern können. Aber zuerst verschaffen wir Ihnen einen kurzen Überblick über TensorFlow.



TensorFlow 2.0 (beta) wurde im Juni 2019 veröffentlicht, und es hat die Verwendung von TensorFlow deutlich vereinfacht. Die erste Auflage dieses Buchs hat TF 1 verwendet, aber diese Auflage setzt auf TF 2.

Ein kurzer Überblick über TensorFlow

Wie Sie wissen, handelt es sich bei TensorFlow um eine leistungsfähige Bibliothek für numerische Berechnungen, die insbesondere besonders gut und sehr anpassbar für Machine Learning im großen Maßstab geeignet ist (aber Sie können sie für alles einsetzen, was umfangreiche Berechnungen erfordert). Sie wurde vom Google-Brain-Team entwickelt und dient als Grundlage für viele der Services von Google, die umfangreiche Rechenanforderungen haben, wie zum Beispiel Google Cloud Speech, Google Photos und Google Search. Im November 2015 wurde sie Open Source, und mittlerweile ist sie die verbreitetste Bibliothek für Deep Learning (in Bezug auf Erwähnungen in Artikeln, die Übernahme in Firmen, Sterne auf GitHub und so weiter). Unzählige Projekte setzen TensorFlow für alle möglichen Aufgaben des Machine Learning ein, wie zum Beispiel für die Bildklassifikation, die Sprachverarbeitung, in Empfehlungssystemen und bei der Zeitserienvorhersage.

Was bietet TensorFlow nun an? Hier eine Zusammenfassung:

- Der Kern ähnelt NumPy sehr stark, aber er bietet GPU-Unterstützung.
- Sie unterstützt verteiltes Rechnen (über mehrere Devices und Server hinweg).
- Sie enthält einen Just-in-Time-(JIT-)Compiler, der es erlaubt, Berechnungen für mehr Geschwindigkeit und besseren Speichereinsatz zu optimieren. Dazu wird der *Rechengraph* einer Python-Funktion ausgelesen, der dann optimiert (zum Beispiel durch das Entfernen ungenutzter Knoten) und schließlich effizient ausgeführt wird (beispielsweise durch das parallele Ausführen voneinander unabhängiger Operationen).
- Rechengraphen können in einem portablen Format exportiert werden, sodass Sie ein TensorFlow-Modell in einer Umgebung trainieren (zum Beispiel mit Python unter Linux) und in einer anderen ausführen können (zum Beispiel mit Java auf einem Android-Gerät).
- Sie implementiert Autodiff (siehe [Kapitel 10](#) und [Kapitel 11](#)) und stellt einige ausgezeichnete Optimierer bereit, wie zum Beispiel RMSProp oder Nadam (siehe [Kapitel 11](#)), sodass Sie alle möglichen Arten von Verlustfunktionen einfach minimieren können.

TensorFlow bietet viele weitere Features an, die darauf aufbauen – das wichtigste ist natürlich `tf.keras`,¹ aber es gibt auch Befehle zum Laden und Vorverarbeiten von Daten (`tf.data`, `tf.io` und so weiter), zur Bildverarbeitung (`tf.image`), zur Signalverarbeitung (`tf.signal`) und vieles mehr (einen Überblick über die Python-API von TensorFlow finden Sie in [Abbildung 12-1](#)).

Auf unterster Ebene ist jede TensorFlow-Operation (oder kurz *Op*) durch sehr effizienten C++-Code implementiert.² Viele Operationen besitzen mehrere Implementierungen, die als *Kernel* bezeichnet werden: Jeder Kernel ist für einen bestimmten Device-Typ gedacht, wie zum Beispiel CPUs, GPUs oder sogar TPUs (*Tensor Processing Units*). Wie Sie vielleicht wissen, können GPUs Berechnungen dramatisch beschleunigen, indem sie sie in viele kleinere Häppchen unterteilen und parallel über mehrere GPU-Threads verteilt ausführen. TPUs sind sogar noch schneller: Es handelt sich um spezielle ASIC-Chips, die für Deep-Learning-Operationen³ gebaut wurden (wir werden den Einsatz von TensorFlow mit GPUs und TPUs in [Kapitel 19](#) behandeln).

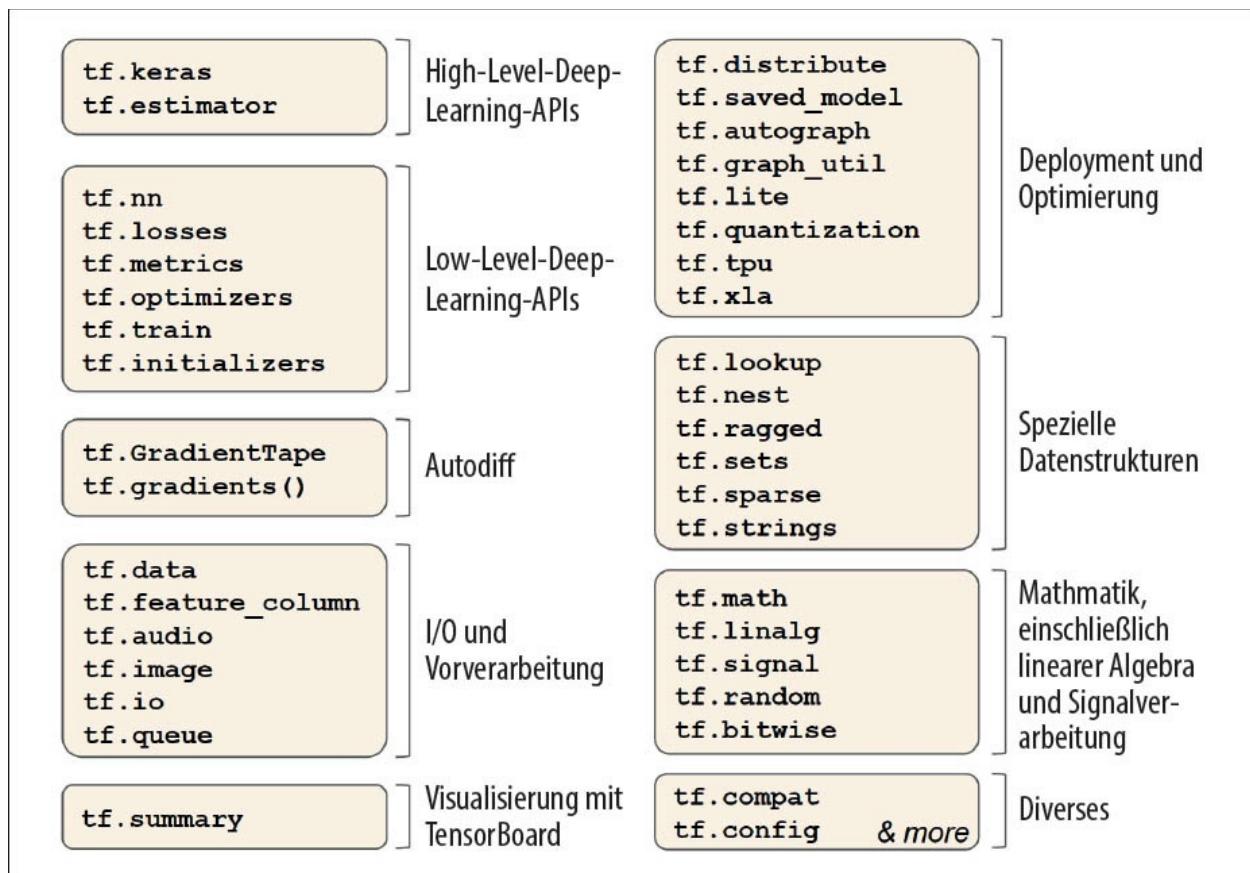


Abbildung 12-1: Die Python-API von TensorFlow

Die Architektur von TensorFlow sehen Sie in Abbildung 12-2. Die meiste Zeit wird Ihr Code die High-Level-APIs einsetzen (insbesondere `tf.keras` und `tf.data`), aber wenn Sie mehr Flexibilität benötigen, werden Sie die Python-API auf niedrigerer Ebene verwenden und direkt mit Tensoren arbeiten. Beachten Sie, dass auch APIs für andere Sprachen zur Verfügung stehen. Auf jeden Fall kümmert sich die Ausführungsengine von TensorFlow darum, die Operationen effizient laufen zu lassen – selbst über mehrere Devices und Maschinen hinweg, wenn Sie das so wollen.

TensorFlow läuft nicht nur unter Windows, Linux und macOS, sondern auch auf mobilen Geräten (mit *TensorFlow Light*), unter anderem auf iOS und Android (siehe Kapitel 19). Wollen Sie die Python-API nicht verwenden, gibt es auch welche für C++, Java, Go und Swift. Es gibt sogar eine Implementierung in JavaScript namens *TensorFlow.js*, die es ermöglicht, Ihre Modelle direkt in Ihrem Browser laufen zu lassen.

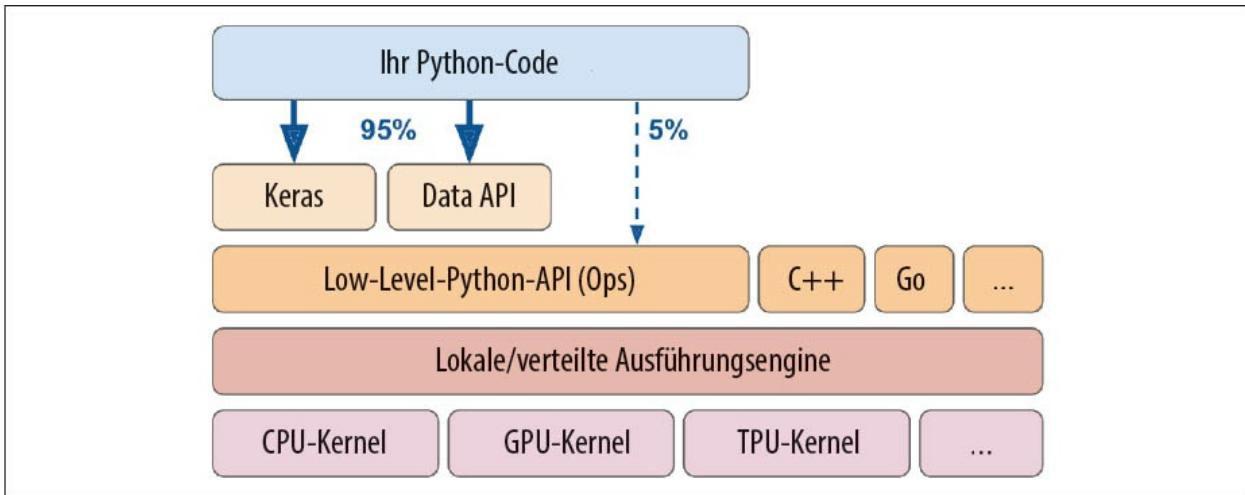


Abbildung 12-2: Die Architektur von TensorFlow

TensorFlow ist aber noch mehr als nur die Bibliothek. Sie befindet sich im Zentrum eines umfassenden Ökosystems aus Bibliotheken. Zunächst einmal gibt es TensorBoard für die Visualisierung (siehe [Kapitel 10](#)). Dann ist da TensorFlow Extended (TFX) (<https://tensorflow.org/tfx>), bei dem es sich um einen Satz an Bibliotheken handelt, die von Google entwickelt wurden, um TensorFlow-Projekte in echte Produkte umzuwandeln; es enthält Tools zur Datenvielfältigkeit, Vorverarbeitung, Modellanalyse und zum Serving (mit TF Serving, siehe [Kapitel 19](#)). Googles *TensorFlow Hub* bietet eine Möglichkeit, vorgebildete neuronale Netze leicht herunterzuladen und wiederzuverwenden. Im Model Garden (<https://github.com/tensorflow/models/>) von TensorFlow erhalten Sie viele Architekturen für neuronale Netze, von denen manche auch vorgebildet sind. In den TensorFlow Resources (<https://www.tensorflow.org/resources>) und unter <https://github.com/jtoy/awesome-tensorflow> finden weitere auf TensorFlow basierende Projekte. GitHub ist ebenfalls eine tolle Quelle für TensorFlow-Projekte, daher ist es oft leicht, für das von Ihnen gewünschte schon bestehende Code zu finden.



Immer mehr ML-Artikel werden zusammen mit ihren Implementierungen veröffentlicht und manchmal sogar mit vorgebildeten Modellen. Schauen Sie bei <https://paperswithcode.com> vorbei, um solche zu finden.

Und schließlich gibt es bei TensorFlow ein Team engagierter und hilfsbereiter Entwickler sowie eine große Community, die Beiträge zur Verbesserung liefert. Um technische Fragen zu stellen, sollten Sie <http://stackoverflow.com/> nutzen und Ihre Frage mit `tensorflow` und `python` taggen. Bugs und Feature Requests können Sie über GitHub (<https://github.com/tensorflow/tensorflow>) erfassen. Allgemeine Diskussionen werden in der Google-Gruppe (<https://groups.google.com/forum/#!topic/homl.info/41>) geführt.

Okay, fangen wir mit dem Programmieren an!

TensorFlow wie NumPy einsetzen

Die TensorFlow-API dreht sich um *Tensoren*, die von Operation zu Operation fließen – daher

der Name *TensorFlow*. Ein Tensor ähnelt stark einem NumPy-ndarray: Er ist normalerweise ein mehrdimensionales Array, er kann aber auch einen Skalar enthalten (ein einfacher Wert wie 42). Diese Tensoren werden wichtig, wenn wir eigene Kostenfunktionen, Metriken, Schichten und anderes erstellen, daher wollen wir sehen, wie wir sie erstellen und bearbeiten.

Tensoren und Operationen

Sie können einen Tensor mit `tf.constant()` erstellen. Dies ist beispielsweise ein Tensor, der eine Matrix mit zwei Zeilen und drei Spalten mit Gleitkommazahlen repräsentiert:

```
>>> tf.constant([[1., 2., 3.], [4., 5., 6.]]) # Matrix  
<tf.Tensor: id=0, shape=(2, 3), dtype=float32, numpy=  
array([[1., 2., 3.],  
       [4., 5., 6.]], dtype=float32)>  
  
>>> tf.constant(42) # Skalar  
<tf.Tensor: id=1, shape=(), dtype=int32, numpy=42>
```

Wie ein ndarray hat ein `tf.Tensor` eine Form und einen Datentyp (`dtype`):

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])  
>>> t.shape  
TensorShape([2, 3])  
>>> t.dtype  
tf.float32
```

Das Indexieren funktioniert wie in NumPy:

```
>>> t[:, 1:]  
<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=  
array([[2., 3.],  
       [5., 6.]], dtype=float32)>  
  
>>> t[..., 1, tf.newaxis]  
<tf.Tensor: id=15, shape=(2, 1), dtype=float32, numpy=  
array([[2.],  
       [5.]], dtype=float32)>
```

Am wichtigsten aber ist, dass alle möglichen Tensor-Operationen zur Verfügung stehen:

```
>>> t + 10  
<tf.Tensor: id=18, shape=(2, 3), dtype=float32, numpy=  
array([[11., 12., 13.],  
       [14., 15., 16.]], dtype=float32)>  
  
>>> tf.square(t)  
<tf.Tensor: id=20, shape=(2, 3), dtype=float32, numpy=  
array([[ 1.,  4.,  9.],  
       [16., 25., 36.]], dtype=float32)>  
  
>>> t @ tf.transpose(t)  
<tf.Tensor: id=24, shape=(2, 2), dtype=float32, numpy=  
array([[14., 32.],  
       [32., 77.]], dtype=float32)>
```

Beachten Sie, dass das Schreiben von `t + 10` äquivalent zum Aufruf von `tf.add(t, 10)` ist (tatsächlich ruft Python die magische Methode `t.__add__(10)` auf, die wiederum einfach `tf.add(t, 10)` aufruft). Andere Operatoren wie `-` und `*` werden ebenfalls unterstützt. Der Operator `@` wurde in Python 3.5 neu hinzugefügt und dient der Matrixmultiplikation: Sein Äquivalent ist der Aufruf der Funktion `tf.matmul()`.

Sie werden alle grundlegenden mathematischen Operationen finden (`tf.add()`, `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()` und so weiter), dazu die meisten Operationen, die auch in NumPy vorhanden sind (zum Beispiel `tf.reshape()`, `tf.squeeze()` oder `tf.tile()`). Manche Funktionen heißen anders als in NumPy – so sind `tf.reduce_mean()`, `tf.reduce_sum()`, `tf.reduce_max()` und `tf.math.log()` die Äquivalente zu `np.mean()`, `np.sum()`, `np.max()` und `np.log()`. Unterscheiden sich die Namen, gibt es oft einen guten Grund dafür. So müssen Sie zum Beispiel in TensorFlow `tf.transpose(t)` schreiben und nicht wie in NumPy einfach `t.T`. Der Grund ist, dass die Funktion `tf.transpose()` nicht genau das Gleiche macht wie das NumPy-Attribut `T`: In TensorFlow wird ein neuer Tensor mit einer eigenen Kopie der transponierten Daten erzeugt, während es sich in NumPy bei `t.T` nur um eine transponierte Sicht auf die gleichen Daten handelt. Genauso heißt die Operation `tf.reduce_sum()` so, weil ihr GPU-Kernel (also die GPU-Implementierung) einen Reduce-Algorithmus verwendet, der nicht garantiert, dass die Reihenfolge durch das Hinzufügen erhalten bleibt: Weil 32-Bit-Gleitkommazahlen eine eingeschränkte Genauigkeit haben, kann sich das Ergebnis bei jedem Aufruf leicht ändern. Das Gleiche gilt für `tf.reduce_mean()` (aber `tf.reduce_max()` ist natürlich deterministisch).

- ☞ Viele Funktionen und Klassen haben Aliase. So handelt es sich beispielsweise bei `tf.add()` und `tf.math.add()` um die gleichen Funktionen. Damit kann TensorFlow kurze Namen für die am häufigsten eingesetzten Operationen nutzen,⁴ während es gleichzeitig seine Pakete sauber organisiert.

Die Low-Level-API von Keras

Die Keras-API besitzt ihre eigene Low-Level-API in `keras.backend`. Sie enthält Funktionen wie `square()`, `exp()` und `sqrt()`. In `tf.keras` rufen diese Funktionen im Allgemeinen einfach die entsprechenden TensorFlow-Operationen auf. Wollen Sie Code schreiben, der auch portabel auf anderen Keras-Implementierungen läuft, sollten Sie diese Keras-Funktionen verwenden. Allerdings decken sie nur eine Untermenge aller in TensorFlow verfügbaren Operationen ab, daher werden wir in diesem Buch die TensorFlow-Operationen direkt verwenden. Hier ein einfaches Beispiel, das `keras.backend` einsetzt, das meist schlicht `K` genannt wird:

```
>>> from tensorflow import keras  
  
>>> K = keras.backend  
  
>>> K.square(K.transpose(t)) + 10  
  
<tf.Tensor: id=39, shape=(3, 2), dtype=float32, numpy=  
array([[11., 26.],  
       [14., 35.],  
       [19., 46.]], dtype=float32)>
```

Tensoren und NumPy

Tensoren sind gut in NumPy integriert – Sie können sie aus einem NumPy-Array erstellen und umgekehrt. Sie können sogar TensorFlow-Operationen auf NumPy-Arrays anwenden und NumPy-Operationen auf Tensoren:

```
>>> a = np.array([2., 4., 5.])  
  
>>> tf.constant(a)  
  
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>  
  
>>> t.numpy() # oder np.array(t)  
  
array([[1., 2., 3.],  
       [4., 5., 6.]], dtype=float32)
```

```

>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=array([4., 16., 25.])>
>>> np.square(t)
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)

```

- § Beachten Sie, dass NumPy standardmäßig eine 64-Bit-Genauigkeit nutzt, während TensorFlow mit 32 Bit arbeitet. Das liegt daran, dass 32-Bit-Genauigkeit für neuronale Netze im Allgemeinen ausreichend ist, zudem laufen die Programme schneller und brauchen weniger RAM. Wenn Sie also einen Tensor aus einem NumPy-Array erstellen, achten Sie darauf, `dtype=tf.float32` zu setzen.

Typumwandlung

Typumwandlungen können die Performance deutlich ausbremsen, und wenn sie automatisch ausgeführt werden, fallen sie gerne überhaupt nicht auf. Um das zu vermeiden, führt TensorFlow keinerlei Typumwandlungen automatisch aus – sie führen nur zu einer Exception, wenn Sie versuchen, eine Operation auf Tensoren mit inkompatiblen Typen anzuwenden. So können Sie beispielsweise keinen Float-Tensor zu einem Integer-Tensor addieren und nicht einmal einen 32-Bit-Float und einen 64-Bit-Float aufsummieren:

```

>>> tf.constant(2.) + tf.constant(40)
Traceback[...]InvalidArgumentError[...]expected to be a float[...]
>>> tf.constant(2.) + tf.constant(40., dtype=tf.float64)
Traceback[...]InvalidArgumentError[...]expected to be a double[...]

```

Das kann zu Beginn etwas nervig sein, aber denken Sie daran, dass es einen guten Grund hat! Und natürlich können Sie `tf.cast()` einsetzen, wenn Sie tatsächlich Typen umwandeln müssen:

```

>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32)
<tf.Tensor: id=136, shape=(), dtype=float32, numpy=42.0>

```

Variablen

Die `tf.Tensor`-Werte, denen wir bisher begegnet sind, sind immutabel: Sie können sie nicht verändern. Wir können also keine normalen Tensoren verwenden, um Gewichte in einem neuronalen Netz zu implementieren, da sie per Backpropagation angepasst werden müssen.

Zudem können sich auch andere Parameter mit der Zeit ändern (zum Beispiel merkt sich ein Momentum-Optimierer die alten Gradienten). Wir brauchen daher eine `tf.Variable`:

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])  
>>> v  
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=  
array([[1., 2., 3.],  
       [4., 5., 6.]], dtype=float32)>
```

Eine `tf.Variable` verhält sich sehr ähnlich wie ein `tf.Tensor`: Sie können die gleichen Operationen darauf ausführen, sie arbeitet gut mit NumPy zusammen, und sie ist genauso wählbar wie bei den Typen. Aber sie kann auch mithilfe der Methode `assign()` verändert werden (oder `assign_add()` bzw. `assign_sub()`), was die Variable um den gegebenen Wert erhöht oder verringert. Sie können auch einzelne Zellen (oder Slices) über die Methode `assign()` der Zelle (oder des Slices) bearbeiten (ein direktes Verändern ist nicht möglich) oder die Methoden `scatter_update()` oder `scatter_nd_update()` nutzen:

```
v.assign(2 * v)           # => [[2., 4., 6.], [8., 10., 12.]]  
v[0, 1].assign(42)        # => [[2., 42., 6.], [8., 10., 12.]]  
v[:, 2].assign([0., 1.])  # => [[2., 42., 0.], [8., 10., 1.]]  
v.scatter_nd_update(indices=[[0, 0], [1, 2]], updates=[100., 200.])  
                           # => [[100., 42., 0.], [8., 10., 200.]]
```

- ¶ In der Praxis werden Sie Variablen selten manuell erstellen müssen, da Keras eine Methode `add_weight()` bereitstellt, die sich für Sie darum kümmert (wie wir noch sehen werden). Und Modellparameter werden im Allgemeinen direkt von den Optimierern aktualisiert, daher müssen Sie Variablen auch nur selten manuell verändern.

Andere Datenstrukturen

TensorFlow unterstützt eine Reihe weiterer Datenstrukturen, unter anderem die folgenden (Details finden Sie im Abschnitt »Tensors and Operations« des Notebooks oder in [Anhang F](#)):

Spärlich besetzte Tensoren (`tf.SparseTensor`)

Effiziente Repräsentation von Tensoren, die vorwiegend Nullen enthalten. Das Paket `tf.sparse` enthält Operationen für spärliche besetzte Tensoren.

Tensoren-Arrays (`tf.TensorArray`)

Listen von Tensoren. Sie haben standardmäßig eine feste Größe, können aber optional dynamisch gemacht werden. Alle enthaltenen Tensoren müssen die gleiche Form und den gleichen Datentyp besitzen.

Ragged-Tensoren (tf.RaggedTensor)

Statische Listen von Tensor-Listen, bei denen jeder Tensor die gleiche Form und den gleichen Datentyp besitzt. Das Paket `tf.ragged` enthält Operationen für Ragged-Tensoren.

String-Tensoren

Normale Tensoren vom Typ `tf.string`. Sie repräsentieren Bytestrings, keine Unicodestrings. Wollen Sie also einen String-Tensor mit einem Unicodestring erstellen (zum Beispiel einen normalen Python-3-String wie "café"), wird dieser automatisch nach UTF-8 konvertiert (zum Beispiel `b"caf\xc3\xa9"`). Alternativ können Sie Unicodestrings durch Tensoren des Typs `tf.int32` repräsentieren, bei denen jedes Element einem Unicode Code Point entspricht (zum Beispiel `[99, 97, 102, 233]`). Das Paket `tf.strings` (mit einem `s` am Ende) enthält Operationen für Byte- und Unicodestrings (und das Umwandeln ineinander). Sie müssen sich bewusst sein, dass ein `tf.string` atomar ist und seine Länge daher nicht in der Form des Tensors auftaucht. Haben Sie in einen Unicode-Tensor umgewandelt (also einen vom Typ `tf.int32` mit Unicode Code Points), ist die Länge in der Form zu sehen.

Sets

Werden als normale (oder spärlich besetzte) Tensoren repräsentiert. So steht beispielsweise `tf.constant([[1, 2], [3, 4]])` für die beiden Sets $\{1, 2\}$ und $\{3, 4\}$. Ganz allgemein wird jedes Set durch einen Vektor in der letzten Achse des Tensors repräsentiert. Sie können Sets mithilfe der Operationen aus dem Paket `tf.sets` bearbeiten.

Queues

Speichern Tensoren über mehrere Schritte hinweg. TensorFlow bietet verschiedene Arten von Queues an: einfache First-In-First-Out-(FIFO-)Queues (FIFO-Queue), Queues, die einzelne Elemente priorisieren können (PriorityQueue), ihre Elemente durchmischen (RandomShuffleQueue) und Elemente unterschiedlicher Form per Padding zusammenbringen (PaddingFIFOQueue). Diese Klassen finden sich alle im Paket `tf.queue`.

Mit den Ihnen so zur Verfügung stehenden Tensoren, Operationen, Variablen und diversen anderen Datenstrukturen können Sie jetzt Ihre Modelle und Trainingsalgorithmen anpassen!

Modelle und Trainingsalgorithmen anpassen

Beginnen wir mit dem Erstellen einer eigenen Verlustfunktion – einem einfachen und häufigen Anwendungsfall.

Eigene Verlustfunktion

Stellen Sie sich vor, Sie wollten ein Regressionsmodell trainieren, aber Ihr Trainingsdatensatz ist ein bisschen verrauscht. Natürlich räumen Sie Ihren Datensatz zuerst auf, indem Sie die Outlier entfernen oder korrigieren, aber das reicht nicht aus – der Datensatz ist immer noch verrauscht. Welche Verlustfunktion sollten Sie verwenden? Der mittlere quadratische Fehler bestraft große Fehler eventuell zu stark und sorgt dafür, dass Ihr Modell ungenau wird. Der mittlere absolute Fehler bestraft die Outlier nicht so sehr, aber es kann recht lange dauern, bis das Training

konvergiert, und das trainierte Modell wird eventuell nicht sehr genau sein. Das ist möglicherweise der richtige Moment, um statt des guten alten MSE den Huber-Fehler auszuprobieren (siehe [Kapitel 10](#)). Dieser ist aktuell nicht Teil der offiziellen Keras-API, steht aber in `tf.keras` zur Verfügung (nutzen Sie einfach eine Instanz der Klasse `keras.losses.Huber`). Aber tun wir einmal so, als ob es ihn nicht gäbe – das Implementieren ist wirklich super einfach! Erstellen Sie nur eine Funktion, die die Labels und Vorhersagen als Argumente übernimmt, und verwenden Sie TensorFlow-Operationen, um den Fehler jeder Instanz zu berechnen:

```
def huber_fn(y_true, y_pred):  
    error = y_true - y_pred  
    is_small_error = tf.abs(error) < 1  
    squared_loss = tf.square(error) / 2  
    linear_loss = tf.abs(error) - 0.5  
    return tf.where(is_small_error, squared_loss, linear_loss)
```



Für eine bessere Performance sollten Sie eine vektorisierte Implementierung nutzen, wie das in diesem Beispiel geschieht. Und wenn Sie vom Graph-Feature von TensorFlow profitieren wollen, sollten Sie nur TensorFlow-Operationen einsetzen.

Es lohnt sich auch, einen Tensor mit einem Verlust pro Instanz zurückzugeben statt den durchschnittlichen Verlust zu liefern. So kann Keras bei Bedarf Kategorien- oder Sample-Gewichte anwenden (siehe [Kapitel 10](#)).

Nun können Sie diesen Verlust beim Kompilieren des Keras-Modells verwenden und dann Ihr Modell trainieren:

```
model.compile(loss=huber_fn, optimizer="nadam")  
model.fit(X_train, y_train, [...])
```

Das ist alles! Für jeden Batch wird Keras während des Trainings die Funktion `huber_fn()` aufrufen, um den Verlust zu berechnen und damit einen Gradientenschritt durchzuführen. Zudem wird es sich den Gesamtverlust seit dem Beginn jeder Epoche merken und den durchschnittlichen Verlust anzeigen.

Aber was passiert mit dieser eigenen Verlustfunktion, wenn Sie das Modell sichern?

Modelle mit eigenen Komponenten sichern und laden

Das Sichern eines Modells mit einer eigenen Verlustfunktion funktioniert problemlos, da Keras den Namen der Funktion mitspeichert. Wenn Sie es laden, müssen Sie ein Dictionary mitgeben, das den Funktionsnamen mit der eigentlichen Funktion verbindet. Oder allgemeiner: Wenn Sie

ein Modell mit eigenen Objekten laden, müssen Sie die Namen auf die Objekte abbilden:

```
model = keras.models.load_model("my_model_with_a_custom_loss.h5",
                                custom_objects={"huber_fn": huber_fn})
```

Mit der aktuellen Implementierung wird jeder Fehler zwischen -1 und 1 als »klein« angesehen. Aber was ist, wenn Sie einen anderen Grenzwert nutzen wollen? Eine Lösung ist, eine Funktion zu erstellen, die eine konfigurierte Verlustfunktion erzeugt:

```
def create_huber(threshold=1.0):

    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss = threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)

    return huber_fn

model.compile(loss=create_huber(2.0), optimizer="nadam")
```

Wenn Sie das Modell sichern, wird der `threshold` leider nicht mit gespeichert. Sie werden ihn also beim Laden des Modells wieder angeben müssen (beachten Sie, dass der Name der zu verwendenden Funktion `"huber_fn"` ist – also der Name der Funktion, die Sie Keras mitgegeben haben, nicht der Name der erzeugenden Funktion):

```
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})
```

Sie können dies lösen, indem Sie eine Subklasse von `keras.losses.Loss` erstellen und dann deren Methode `get_config()` implementieren:

```
class HuberLoss(keras.losses.Loss):

    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)

    def call(self, y_true, y_pred):
        error = y_true - y_pred
```

```

    is_small_error = tf.abs(error) < self.threshold

    squared_loss = tf.square(error) / 2

    linear_loss = self.threshold * tf.abs(error) - self.threshold**2 / 2

    return tf.where(is_small_error, squared_loss, linear_loss)

def get_config(self):

    base_config = super().get_config()

    return {**base_config, "threshold": self.threshold}

```

- Die Keras-API gibt aktuell nur an, wie Sie Subklassen zum Definieren von Schichten, Modellen, Callbacks und Regularisierern verwenden. Bauen Sie andere Komponenten (wie Verlustfunktionen, Metriken, Initialisierer oder Constraints) mit Subklassen, lassen sich diese eventuell nicht auf andere Keras-Implementierungen portieren. Vermutlich wird die Keras-API so angepasst werden, dass auch Subklassen für all diese Komponenten möglich sind.

Schauen wir uns den Code genauer an:

- Der Konstruktor übernimmt `**kwargs` und übergibt sie an den übergeordneten Konstruktor, der sich um die Standard-Hyperparameter kümmert – den Namen der Verlustfunktion und den `reduction`-Algorithmus zum Aggregieren der Verluste der einzelnen Instanzen. Standardmäßig ist das "`sum_over_batch_size`", womit der Verlust aus der Summe der Instanzverluste entsteht, gewichtet mit den Sample-Gewichten (sofern vorhanden) und geteilt durch die Batchgröße (nicht durch die Summe der Gewichte, womit es sich *nicht* um das gewichtete Mittel handelt).⁵ Andere mögliche Werte sind "`sum`" und `None`.
- Die Methode `call()` übernimmt die Labels und Vorhersagen, berechnet alle Instanzverluste und liefert sie zurück.
- Die Methode `get_config()` gibt ein Dictionary zurück, in dem jeder Name eines Hyperparameters auf seinen Wert abgebildet wird. Sie ruft zuerst die Methode `get_config()` der übergeordneten Klasse auf und fügt dann die neuen Hyperparameter diesem Dictionary hinzu (beachten Sie, dass die praktische Syntax `{**x}` mit Python 3.5 dazukam).

Jetzt können Sie beim Komplizieren des Modells Instanzen dieser Klasse verwenden:

```
model.compile(loss=HuberLoss(2.), optimizer="adam")
```

Sichern Sie das Modell, wird der Grenzwert mit abgespeichert; laden Sie es, müssen Sie nur den Klassennamen auf die Klasse selbst abbilden:

```
model = keras.models.load_model("my_model_with_a_custom_loss_class.h5",
```

```
        custom_objects={"HuberLoss": HuberLoss})
```

Sichern Sie ein Modell, ruft Keras die Methode `get_config()` der Instanz der Verlustklasse auf und sichert die Konfiguration als JSON in der HDF5-Datei. Laden Sie das Modell, ruft es die Klassenmethode `from_config()` der Klasse `HuberLoss` auf: Diese Methode wird durch die Basisklasse implementiert (`Loss`), sie erzeugt eine Instanz der Klasse und übergibt `**config` an den Konstruktor.

Das ist schon alles für die Verlustfunktion! Gar nicht so schwer, oder? Genauso einfach sind eigene Aktivierungsfunktionen, Initialisierer, Regularisierer und Constraints. Schauen wir uns diese als Nächstes an.

Eigene Aktivierungsfunktionen, Initialisierer, Regularisierer und Constraints

Ein Großteil der Funktionalität von Keras, wie zum Beispiel Verlustfunktionen, Regularisierer, Constraints, Initialisierer, Metriken, Aktivierungsfunktionen, Schichten und sogar ganze Modelle, kann auf die gleiche Weise angepasst werden. Meist werden Sie nur eine einfache Funktion mit den passenden Ein- und Ausgabeparametern schreiben müssen. Hier ein paar Beispiele für eine eigene Aktivierungsfunktion (entsprechend `keras.activations.softplus()` oder `tf.nn.softplus()`), einen eigenen Glorot-Initialisierer (entsprechend `keras.initializers.glorot_normal()`), einen eigenen ℓ_1 -Regularisierer (entsprechend `keras.regularizers.l1(0.01)`) und einen eigenen Constraint, der sicherstellt, dass die Gewichte alle positiv sind (entsprechend `keras.constraints.nonneg()` oder `tf.nn.relu()`):

```
def my_softplus(z): # Rückgabewert ist einfach tf.nn.softplus(z)
    return tf.math.log(tf.exp(z) + 1.0)

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))

def my_positive_weights(weights): # Rückgabewert ist einfach tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

Wie Sie sehen, hängen die Argumente von der Art der Funktion ab. Diese eigenen Funktionen können dann ganz normal eingesetzt werden, zum Beispiel:

```

layer = keras.layers.Dense(30, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)

```

Die Aktivierungsfunktion wird auf die Ausgabe dieser Dense-Schicht angewendet und deren Ergebnis an die nächste Schicht weitergegeben. Die Gewichte der Schicht werden mit den Werten aus dem Initialisierer besetzt. Bei jedem Trainingsschritt werden die Gewichte an die Regularisierungsfunktion weitergegeben, um die Regularisierungsverluste zu berechnen, die dann zum Gesamtverlust addiert werden, um den abschließenden Verlust für das Training zu berechnen. Und schließlich wird die Constraint-Funktion nach jedem Trainingsschritt aufgerufen, und die Gewichte der Schicht werden durch die begrenzten Gewichte ersetzt.

Besitzt eine Funktion Hyperparameter, die zusammen mit dem Modell gesichert werden müssen, werden Sie Subklassen der passenden Klassen erstellen wollen, wie zum Beispiel `keras.regularizers.Regularizer`, `keras.constraints.Constraint`, `keras.initializers.Initializer` oder `keras.layers.Layer` (für alle Schichten, einschließlich der Aktivierungsfunktion). So ähnlich wie bei der eigenen Verlustfunktion ist das Folgende eine einfache Klasse für die ℓ_1 -Regularisierung, die ihren Hyperparameter `factor` sichert (dieses Mal müssen wir den übergeordneten Konstruktor oder die Methode `get_config()` nicht aufrufen, da sie nicht in der übergeordneten Klasse definiert wurden):

```

class MyL1Regularizer(keras.regularizers.Regularizer):

    def __init__(self, factor):
        self.factor = factor

    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))

    def get_config(self):
        return {"factor": self.factor}

```

Beachten Sie, dass Sie die Methode `call()` für Verlustfunktionen, Schichten (einschließlich Aktivierungsfunktionen) und Modelle implementieren müssen, während für Regularisierer, Initialisierer und Constraints `__call__()` erforderlich ist. Bei Metriken sieht das Ganze ein bisschen anders aus, wie wir gleich sehen werden.

Eigene Metriken

Verlustfunktionen und Metriken sind konzeptionell verschieden: Verlustfunktionen (zum Beispiel die Kreuzentropie) werden vom Gradientenverfahren genutzt, um ein Modell zu

trainieren, daher müssen sie differenzierbar sein (zumindest dort, wo sie ausgewertet werden), und ihre Gradienten sollen nicht überall 0 sein. Zudem ist es in Ordnung, wenn sie sich nicht so einfach durch Menschen interpretieren lassen. Im Gegensatz dazu werden Metriken (zum Beispiel die Genauigkeit) verwendet, um ein Modell zu *evaluieren*: Sie müssen leicht interpretierbar sein, brauchen sich aber nicht differenzierbar zu lassen und dürfen auch überall einen Gradienten von 0 haben.

Trotzdem läuft das Definieren einer eigenen Metrikfunktion in den meisten Fällen genauso ab wie das Definieren einer eigenen Verlustfunktion. Tatsächlich könnten wir sogar die weiter oben erstellte Huber-Verlustfunktion als Metrik einsetzen⁶ – sie würde problemlos funktionieren (und die Persistenz würde ebenfalls genauso ablaufen, in diesem Fall würde nur der Name der Funktion "huber_fn" abgespeichert):

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

Keras berechnet diese Metrik während des Trainings für jeden Batch und merkt sich deren Mittelwert ab dem Beginn der Epoche. Meist ist das genau das, was Sie wollen. Aber nicht immer! Stellen Sie sich zum Beispiel die Genauigkeit eines Binärklassifizierers vor. Wie wir in [Kapitel 3](#) gesehen haben, ist die Genauigkeit die Anzahl der echt Positiven geteilt durch die Anzahl der positiven Vorhersagen (einschließlich echt und falsch Positiver). Angenommen, das Modell würde im ersten Batch fünf positive Vorhersagen treffen, von denen vier korrekt wären: Das sind 80% Genauigkeit. Nun stellen Sie sich vor, das Modell würde im zweiten Batch drei positive Vorhersagen treffen, aber alle drei wären falsch: Das sind 0% Genauigkeit für den zweiten Batch. Würden Sie einfach nur den Mittelwert beider Genauigkeiten bestimmen, wären das 40%. Aber Moment – das ist *nicht* die Genauigkeit des Modells für die beiden Batches! Tatsächlich gab es insgesamt vier echt Positive ($4 + 0$) von acht positiven Vorhersagen ($5 + 3$), damit liegt die Gesamtgenauigkeit bei 50% und nicht bei 40%. Wir brauchen also ein Objekt, das sich die Anzahl der echt Positiven und der falsch Positiven merkt und bei Bedarf daraus das Verhältnis errechnet. Genau das tut die Klasse `keras.metrics.Precision`:

```
>>> precision = keras.metrics.Precision()  
  
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])  
<tf.Tensor: id=581729, shape=(), dtype=float32, numpy=0.8>  
  
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])  
<tf.Tensor: id=581780, shape=(), dtype=float32, numpy=0.5>
```

In diesem Beispiel haben wir ein `Precision`-Objekt erstellt, es dann wie eine Funktion verwendet und Labels und Vorhersagen für den ersten und dann für den zweiten Batch übergeben (wir hätten auch Sample-Gewichte mitgeben können). Wir haben die gleiche Anzahl von echt und falsch Positiven wie im oben besprochenen Beispiel verwendet. Nach dem ersten Batch wird eine Genauigkeit von 80% zurückgegeben, nach dem zweiten Batch eine von 50% (was der Gesamtgenauigkeit bis dato entspricht, nicht der des zweiten Batches). Dies wird als

Streaming-Metrik (oder *zustandsbehaftete Metrik*) bezeichnet, da sie Batch für Batch aktualisiert wird.

Wir können jederzeit die Methode `result()` aufrufen, um den aktuellen Wert der Metrik zu erhalten. Auch können wir uns ihre Variablen (mit der Anzahl der echt und falsch Positiven) über das Attribut `variables` anschauen und diese über die Methode `reset_states()` zurücksetzen:

```
>>> precision.result()
<tf.Tensor: id=581794, shape=(), dtype=float32, numpy=0.5>
>>> precision.variables
[<tf.Variable 'true_positives:0' [...] numpy=array([4.], dtype=float32)>,
 <tf.Variable 'false_positives:0' [...] numpy=array([4.], dtype=float32)>]
>>> precision.reset_states() # Beide Variablen werden auf 0.0 zurückgesetzt.
```

Müssen Sie eine solche Streaming-Metrik erstellen, erzeugen Sie eine Subklasse der Klasse `keras.metrics.Metric`. Hier ein einfaches Beispiel, das sich den Gesamt-Huber-Fehler und die Anzahl der bisherigen Instanzen merkt. Wird sie nach dem Ergebnis gefragt, gibt sie das Verhältnis zurück, was einfach der durchschnittliche Huber-Fehler ist:

```
class HuberMetric(keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs) # Basisargumente (z. B. dtype)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")
    def update_state(self, y_true, y_pred, sample_weight=None):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
    def result(self):
        return self.total / self.count
    def get_config(self):
```

```

base_config = super().get_config()

return {**base_config, "threshold": self.threshold}

```

Schauen wir uns diesen Code an:⁷

- Der Konstruktor nutzt die Methode `add_weight()`, um die für das Erfassen des Metrikstatus notwendigen Variablen über mehrere Batches hinweg zu verwenden – in diesem Fall die Summe aller Huber-Fehler (`total`) sowie die Anzahl der bisherigen Instanzen (`count`). Sie könnten die Variablen nach Wunsch auch selbst erstellen. Keras merkt sich jede `tf.Variable`, die als Attribut gesetzt wird (und allgemeiner jedes »trackbare« Objekt, wie zum Beispiel Schichten oder Modelle).
- Die Methode `update_state()` wird aufgerufen, wenn Sie eine Instanz dieser Klasse als Funktion verwenden (wie wir das mit dem `Precision`-Objekt getan haben). Sie aktualisiert die Variablen durch die gegebenen Labels und Vorhersagen für einen Batch (und die Sample-Gewichte, aber in diesem Fall ignorieren wir sie).
- Die Methode `result()` berechnet das Ergebnis und liefert es zurück – in diesem Fall die durchschnittliche Huber-Metrik über alle Instanzen. Nutzen Sie die Metrik als Funktion, wird zuerst die Methode `update_state()` und dann `result()` aufgerufen, um das Ergebnis zurückzuliefern.
- Wir implementieren auch die Methode `get_config()`, um sicherzustellen, dass `threshold` zusammen mit dem Modell gesichert wird.
- Die Standardimplementierung der Methode `reset_states()` setzt alle Variablen auf 0,0 zurück (aber Sie können sie bei Bedarf auch überschreiben).



Keras kümmert sich um das Persistieren der Variablen – Sie müssen nichts weiter tun.

Definieren Sie eine Metrik über eine einfache Funktion, ruft Keras diese automatisch für jeden Batch auf und merkt sich den Mittelwert für jede Epoche – so wie wir das manuell getan haben. Der einzige Vorteil unserer Klasse `HuberMetric` ist daher, dass der `threshold` gesichert wird. Aber natürlich können manche Metriken – wie die Genauigkeit – nicht einfach Mittelwerte über die Batches erstellen. In solchen Fällen gibt es keine andere Option, als eine Streaming-Metrik zu implementieren.

Nachdem wir nun eine Streaming-Metrik gebaut haben, wird das Erstellen einer eigenen Schicht wie ein Spaziergang wirken!

Eigene Schichten

Gelegentlich werden Sie eine Architektur mit einer exotischen Schicht nutzen wollen, für die TensorFlow keine Standardimplementierung bietet. Dann benötigen Sie eine eigene Schicht. Oder Sie wollen einfach eine sich sehr stark wiederholende Architektur erstellen, die sehr viele

identische Blöcke mit Schichten enthält. Da wäre es vielleicht praktischer, jeden Block mit Schichten als eine eigene Schicht zu behandeln. Ist das Modell beispielsweise eine Folge von Schichten A, B, C, A, B, C, A, B, C, möchten Sie vielleicht eine eigene Schicht D definieren, die die Schichten A, B, C enthält, sodass Ihr Modell einfach nur D, D, D wäre. Schauen wir uns an, wie wir eigene Schichten bauen können.

Es gibt Schichten ohne Gewichte, wie zum Beispiel `keras.layers.Flatten` oder `keras.layers.ReLU`. Wollen Sie eine eigene Schicht ohne Gewichte erzeugen, ist es am einfachsten, eine Funktion zu schreiben und diese in einer Schicht `keras.layers.Lambda` zu verpacken. So wendet beispielsweise die folgende Schicht die Exponentialfunktion auf ihre Eingaben an:

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

Diese eigene Schicht kann dann wie jede andere Schicht genutzt werden – mit der Sequential API, der Functional API oder der Subclassing API. Sie können sie auch als Aktivierungsfunktion verwenden (oder Sie nutzen `activation=tf.exp`, `activation=keras.activations.exponential` oder einfach `activation="exponential"`). Die Exponential-Schicht kommt manchmal in der Ausgabeschicht eines Regressionsmodells zum Einsatz, wenn die vorherzusagenden Werte in sehr unterschiedlichen Größenordnungen liegen (zum Beispiel 0,001, 10 und 1000).

Wie Sie vermutlich schon erraten haben, müssen Sie zum Erstellen einer eigenen, zustandsbehafteten Schicht (also einer mit Gewichten) eine Subklasse von `keras.layers.Layer` anlegen. So implementiert die folgende Klasse zum Beispiel eine vereinfachte Version der Dense-Schicht:

```
class MyDense(keras.layers.Layer):

    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="kernel", shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units], initializer="zeros")
```

```

super().build(batch_input_shape) # muss am Ende stehen

def call(self, X):
    return self.activation(X @ self.kernel + self.bias)

def compute_output_shape(self, batch_input_shape):
    return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])

def get_config(self):
    base_config = super().get_config()
    return {**base_config, "units": self.units,
            "activation": keras.activations.serialize(self.activation)}

```

Schauen wir uns den Code genauer an:

- Der Konstruktor übernimmt alle Hyperparameter als Argumente (in diesem Beispiel `units` und `activation`) und vor allem ein Argument `**kwargs`. Er ruft den übergeordneten Konstruktor auf und übergibt ihm `kwargs`: Dieser kümmert sich um die Standardargumente wie `input_shape`, `trainable` und `name`. Dann sichert er die Hyperparameter als Attribute und wandelt das Argument `activation` mittels `keras.activations.get()` in die passende Aktivierungsfunktion um (sie kann mit Funktionen, Standardstrings wie "relu" oder "selu" oder einfach mit `None` umgehen).⁸
- Die Aufgabe der Methode `build()` ist es, die Variablen der Schicht zu erstellen, indem sie für jedes Gewicht die Methode `add_weight()` aufruft. `build()` wird aufgerufen, wenn die Schicht das erste Mal verwendet wird. Dann kennt Keras die Form der Eingabewerte der Schicht und wird sie an die Methode übergeben,⁹ was zum Erstellen einiger der Gewichte häufig notwendig ist. So müssen wir beispielsweise die Anzahl der Neuronen der vorherigen Schicht kennen, um die Matrix mit den Verbindungsgewichten zu erzeugen (also den "kernel"): Das entspricht der Größe der letzten Dimension der Eingaben. Am Ende der Methode `build()` (und nur dort) müssen Sie die Methode `build()` der übergeordneten Klasse aufrufen. Das teilt Keras mit, dass die Schicht gebaut wurde (es wird einfach `self.built=True` gesetzt).
- Die Methode `call()` führt die gewünschten Operationen aus. In diesem Fall berechnen wir die Matrixmultiplikation der Eingaben `X` mit dem Kernel der Schicht, addieren den Bias-Vektor und wenden die Aktivierungsfunktion auf das Ergebnis an, um damit die Ausgabe der Schicht zu erhalten.
- Die Methode `compute_output_shape()` liefert einfach die Form der Ausgaben dieser Schicht zurück. In diesem Fall entspricht sie der Eingabeform, nur dass die letzte Dimension durch die Anzahl der Neuronen pro Schicht ersetzt wurde. Beachten Sie, dass

Formen in `tf.keras` Instanzen der Klasse `tf.Tensor` `Shape` sind, die Sie mit `as_list()` in Python-Listen umwandeln können.

- Die Methode `get_config()` macht das Gleiche wie in den vorherigen Klassen. Beachten Sie, dass wir die vollständige Konfiguration der Aktivierungsfunktion durch einen Aufruf von `keras.activations.serialize()` sichern.

Jetzt können Sie eine `MyDense`-Schicht wie jede andere Schicht verwenden!

Sie können die Methode `compute_output_shape()` im Allgemeinen weglassen, da `tf.keras` die Form der Ausgabe automatisch ermittelt – außer wenn die Schicht dynamisch ist (wie wir gleich sehen werden). In anderen Keras-Implementierungen ist diese Methode entweder erforderlich, oder ihre Standardimplementierung geht davon aus, dass die Ausgabeform der Eingabeform entspricht.

Um eine Schicht mit mehreren Eingaben zu erstellen (zum Beispiel `Concatenate`), sollte es sich beim Argument der Methode `call()` um ein Tupel mit allen Eingaben handeln – genauso sollte das Argument von `compute_output_shape()` ein Tupel mit allen Formen der Eingabe-Batches sein. Um eine Schicht mit mehreren Ausgaben zu erzeugen, sollte die Methode `call()` die Liste der Ausgaben zurückgeben und `compute_output_shape()` die Liste der Batchausgabeformen liefern (eine pro Ausgabe). Die folgende Übungsschicht übernimmt beispielsweise zwei Eingaben und gibt drei Ausgaben zurück:

```
class MyMultiLayer(keras.layers.Layer):  
  
    def call(self, x):  
        x1, x2 = x  
  
        return [x1 + x2, x1 * x2, x1 / x2]  
  
    def compute_output_shape(self, batch_input_shape):  
        b1, b2 = batch_input_shape  
  
        return [b1, b1, b1] # sollte eventuell Broadcasting-Regeln beachten
```

Diese Schicht kann nun wie jede andere Schicht verwendet werden, aber natürlich nur in den Functional und Subclassing APIs, nicht in der Sequential API (die nur Schichten mit einer Ein- und einer Ausgabe akzeptiert).

Muss sich Ihre Schicht während des Trainings und des Testens unterschiedlich verhalten (zum Beispiel beim Einsatz von Dropout- oder BatchNormalization-Schichten), müssen Sie die Methode `call()` um ein Argument `training` ergänzen und damit entscheiden, was zu tun ist. Erstellen wir beispielsweise eine Schicht, die während des Trainings gaußsches Rauschen hinzufügt (zur Regularisierung), während des Testens aber nichts tut (Keras hat dafür schon eine Schicht `keras.layers.GaussianNoise`):

```
class MyGaussianNoise(keras.layers.Layer):
```

```

def __init__(self, stddev, **kwargs):
    super().__init__(**kwargs)
    self.stddev = stddev

def call(self, X, training=None):
    if training:
        noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
        return X + noise
    else:
        return X

def compute_output_shape(self, batch_input_shape):
    return batch_input_shape

```

Damit können Sie jetzt alle eigenen Schichten bauen, die Sie brauchen! Machen wir uns als Nächstes an das Erzeugen eigener Modelle.

Eigene Modelle

Wir haben uns in [Kapitel 10](#) schon angesehen, wie man eigene Modellklassen baut, als es um die Subclassing API gibt.¹⁰ Dabei gibt es keine Fallstricke: Bilden Sie eine Unterklasse der Klasse `keras.Model`, erstellen Sie Schichten und Variablen im Konstruktor und implementieren Sie die Methode `call()`, um dort all das zu tun, was Ihr Modell tun soll. Stellen Sie sich vor, Sie wollten das Modell bauen, das in [Abbildung 12-3](#) zu sehen ist.

Die Eingaben durchlaufen eine erste Dense-Schicht und dann einen *Residual Block*, der aus zwei Dense-Schichten und einer zusätzlichen Operation besteht (wie wir in [Kapitel 14](#) sehen werden, addiert ein Residual Block seine Eingaben zu seinen Ausgaben), dann diesen Block drei weitere Male, nun durch einen zweiten Residual Block, und schließlich schleusen wir das Endergebnis noch durch eine Dense-Ausgabeschicht. Dieses Modell ergibt keinen großen Sinn – es ist nur ein Beispiel, um zu zeigen, dass Sie beliebige Modelle bauen können, auch solche mit Schleifen und Skip-Verbindungen. Um dieses Modell zu implementieren, ist es am besten, zuerst eine `ResidualBlock`-Schicht zu erstellen, da wir eine Reihe identischer Blöcke bauen werden (und wir werden sie eventuell in anderen Modellen wiederverwenden wollen):

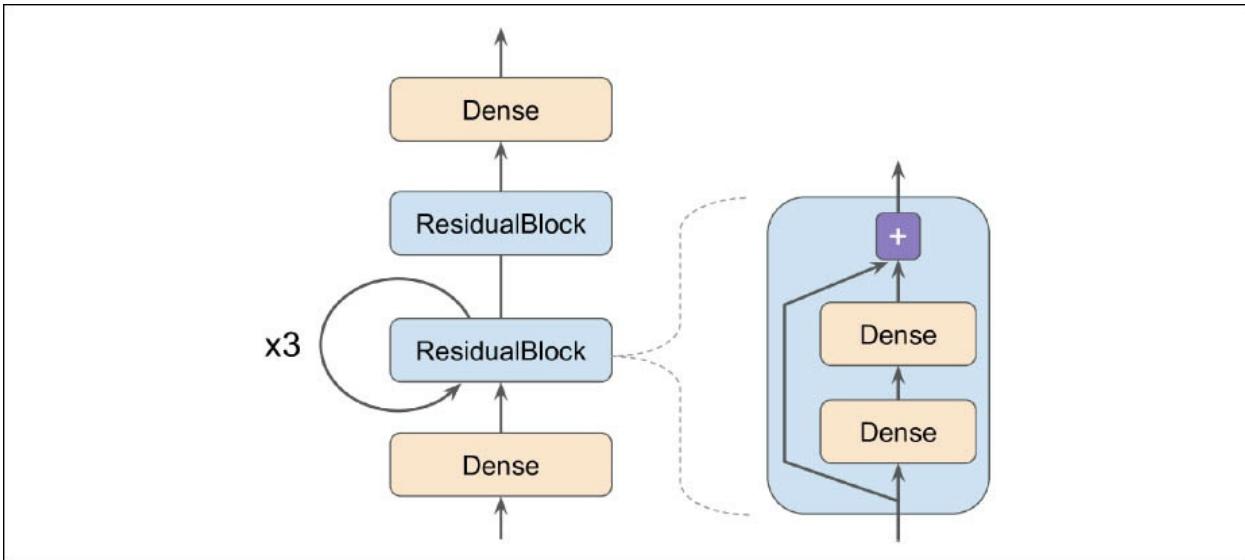


Abbildung 12-3: Beispiel für ein eigenes Modell: ein künstliches Modell mit einer eigenen ResidualBlock-Schicht und einer Skip-Verbindung

```
class ResidualBlock(keras.layers.Layer):

    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                         kernel_initializer="he_normal")
                      for _ in range(n_layers)]

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z
```

Diese Schicht ist ein bisschen speziell, da sie andere Schichten enthält. Das wird transparent von Keras abgebildet: Es erkennt automatisch, dass das Attribut `hidden` trackbare Objekte enthält (in diesem Fall Schichten), daher werden deren Variablen automatisch zur Liste der Schichtvariablen hinzugefügt. Der Rest der Klasse ist selbsterklärend. Als Nächstes wollen wir die Subclassing API verwenden, um das Modell selbst zu definieren:

```
class ResidualRegressor(keras.Model):
```

```

def __init__(self, output_dim, **kwargs):
    super().__init__(**kwargs)

    self.hidden1 = keras.layers.Dense(30, activation="elu",
                                    kernel_initializer="he_normal")

    self.block1 = ResidualBlock(2, 30)
    self.block2 = ResidualBlock(2, 30)

    self.out = keras.layers.Dense(output_dim)

def call(self, inputs):
    Z = self.hidden1(inputs)

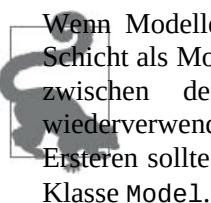
    for _ in range(1 + 3):
        Z = self.block1(Z)
        Z = self.block2(Z)

    return self.out(Z)

```

Wir erstellen die Schichten im Konstruktor und nutzen sie in der Methode `call()`. Dieses Modell kann dann wie jedes andere Modell verwendet werden (kompilieren, fitten, evaluieren und für das Erstellen von Vorhersagen verwenden). Wollen Sie es auch noch über die Methode `save()` sichern und über die Funktion `keras.models.load_model()` laden können, müssen Sie die Methode `get_config()` sowohl in der Klasse `ResidualBlock` wie auch in `ResidualRegressor` implementieren. Alternativ können Sie die Gewichte mit den Methoden `save_weights()` und `load_weights()` sichern und laden.

Die Klasse `Model` ist eine Unterklasse der Klasse `Layer`, daher können Modelle genauso wie Schichten definiert und verwendet werden. Aber ein Modell hat zusätzliche Funktionalität, unter anderem natürlich seine Methoden `compile()`, `fit()`, `evaluate()` und `predict()` (sowie ein paar Varianten), dazu die Methoden `get_layers()` (die jegliche Schichten des Modells anhand ihres Namens oder Index zurückgibt) und `save()` (sowie Unterstützung für `keras.models.load_model()` und `keras.models.clone_model()`).



Wenn Modelle mehr Funktionalität als Schichten bieten – warum definieren wir nicht jede Schicht als Modell? Nun, technisch gesehen könnten Sie das, aber es ist normalerweise sauberer, zwischen den internen Komponenten Ihres Modells (also den Schichten oder wiederverwendbaren Blöcken mit Schichten) und dem Modell selbst zu unterscheiden. Bei Ersteren sollte es sich um Unterklassen der Klasse `Layer` handeln, bei Letzteren um solche der Klasse `Model`.

Damit können Sie nun ganz natürlich und präzise nahezu jedes Modell bauen, das Sie in einem Artikel finden – mit der Sequential API, der Functional API, der Subclassing API oder einer

Mischung aus allen. »Nahezu jedes?« Ja, es gibt noch ein paar Dinge, die wir uns anschauen müssen: Wie definieren wir Verluste oder Metriken anhand der Modell-Interna, und wie bauen wir eine eigene Trainingsschleife?

Verlustfunktionen und Metriken auf Modell-Interna basieren lassen

Die eigenen Verlustfunktionen und Metriken, die wir weiter oben definiert haben, basierten alle auf den Labels und den Vorhersagen (und optional auf den Sample-Gewichten). Es gibt aber Situationen, in denen Sie die Verluste auf anderen Teilen Ihres Modells basieren lassen wollen, wie zum Beispiel auf den Gewichten oder den Aktivierungen der verborgenen Schichten. Das kann zu Regulationszwecken oder zum Überwachen interner Aspekte Ihres Modells sinnvoll sein.

Um eine eigene Verlustfunktion zu definieren, die auf Modell-Interna basiert, berechnen Sie sie abhängig vom gewünschten Modell und übergeben das Ergebnis dann an die Methode `add_loss()`. Bauen wir zum Beispiel ein eigenes Regressions-MLP-Modell, das aus fünf verborgenen Schichten und einer Ausgabeschicht besteht. Dieses eigene Modell wird zudem eine zusätzliche Ausgabe bei der obersten verborgenen Schicht haben. Der damit verbundene Verlust nennt sich *Rekonstruktionsverlust* (siehe [Kapitel 17](#)). Dabei handelt es sich um den gemittelten quadrierten Abstand zwischen der Rekonstruktion und den Eingaben. Indem wir diesen Rekonstruktionsverlust zum Hauptverlust addieren, unterstützen wir das Modell darin, so viel Informationen wie möglich in den verborgenen Schichten beizubehalten – selbst solche, die für die Regressionsaufgabe selbst nicht unbedingt notwendig sind. In der Praxis verbessert dieser Verlust manchmal die Generalisierung (es ist ein Regularisierungsverlust). Hier der Code für dieses eigene Modell mit einem eigenen Rekonstruktionsverlust:

```
class ReconstructingRegressor(keras.Model):  
    def __init__(self, output_dim, **kwargs):  
        super().__init__(**kwargs)  
        self.hidden = [keras.layers.Dense(30, activation="selu",  
                                         kernel_initializer="lecun_normal")  
                     for _ in range(5)]  
        self.out = keras.layers.Dense(output_dim)  
  
    def build(self, batch_input_shape):  
        n_inputs = batch_input_shape[-1]  
        self.reconstruct = keras.layers.Dense(n_inputs)  
        super().build(batch_input_shape)
```

```

def call(self, inputs):
    Z = inputs
    for layer in self.hidden:
        Z = layer(Z)
    reconstruction = self.reconstruct(Z)
    recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
    self.add_loss(0.05 * recon_loss)
    return self.out(Z)

```

Schauen wir uns den Code an:

- Der Konstruktor erstellt das DNN mit fünf verborgenen Dense-Schichten und einer Dense-Ausgabeschicht.
- Die Methode `build()` erzeugt eine zusätzliche Dense-Schicht, die genutzt wird, um die Eingänge des Modells zu rekonstruieren. Sie muss hier erstellt werden, weil ihre Anzahl an Einheiten gleich der Anzahl an Eingängen sein muss, und diese Zahl ist erst bekannt, wenn `build()` aufgerufen wird.
- Die Methode `call()` berechnet den Rekonstruktionsverlust (den mittleren quadratischen Abstand zwischen der Rekonstruktion und den Eingängen) und addiert ihn mithilfe von `add_loss()` zur Liste der Modellverluste.¹¹ Beachten Sie, dass wir den Rekonstruktionsverlust durch Multiplikation mit dem Faktor 0,05 herunterskalieren (das ist ein Hyperparameter, den Sie anpassen können). Damit ist sichergestellt, dass der Rekonstruktionsverlust nicht den Hauptverlust dominiert.
- Schließlich übergibt die Methode `call()` die Ausgabe der verborgenen Schichten an die Ausgabeschicht und liefert deren Ergebnisse.

Genauso können Sie eine eigene Metrik hinzufügen, die auf den Interna des Modells basiert, indem Sie sie wie gewünscht berechnen, solange das Ergebnis die Ausgabe eines Metrikobjekts ist. So können Sie beispielsweise im Konstruktor ein Objekt `keras.metrics.Mean` erstellen, es dann in der Methode `call()` aufrufen, dabei den `recon_loss` übergeben und es schließlich durch Aufruf der Methode `add_metric()` dem Modell hinzufügen. So wird Keras beim Trainieren des Modells sowohl den durchschnittlichen Verlust (der Verlust ist die Summe aus Hauptverlust und 0,05 mal dem Rekonstruktionsverlust) als auch den durchschnittlichen Rekonstruktionsfehler über eine Epoche ausgeben. Beides wird während des Trainings sinken:

Epoch 1/5

```
11610/11610 [=====] [...] loss: 4.3092 - reconstruction_error: 1.7360
```

Epoch 2/5

```
11610/11610 [=====] [...] loss: 1.1232 - reconstruction_error: 0.8964
```

[...]

In über 99% der Fälle wird all das, was wir bisher behandelt haben, ausreichen, um beliebige Modelle zu implementieren, selbst mit komplexen Architekturen, Verlustfunktionen oder Metriken. Aber in einigen seltenen Fällen werden Sie die Trainingsschleife selbst anpassen müssen. Bevor wir uns diesem Thema zuwenden, müssen wir uns anschauen, wie wir in TensorFlow automatisch Gradienten berechnen.

Gradienten per Autodiff berechnen

Um zu verstehen, wie Sie per Autodiff (siehe [Kapitel 10](#) und [Anhang D](#)) Gradienten automatisch berechnen lassen können, schauen wir uns eine einfache Spielfunktion an:

```
def f(w1, w2):  
    return 3 * w1 ** 2 + 2 * w1 * w2
```

Wenn Sie mit Analysis vertraut sind, können Sie analytisch ermitteln, dass die partielle Ableitung dieser Funktion bezüglich w_1 zu $6 \cdot w_1 + 2 \cdot w_2$ führt. Und die partielle Ableitung bezüglich w_2 ist $2 \cdot w_1$. So liefern beispielsweise am Punkt $(w_1, w_2) = (5, 3)$ diese partiellen Ableitungen die Werte 36 und 10, sodass der Gradientenvektor hier $(36, 10)$ ist. Aber wenn dies ein neuronales Netz wäre, würde die Funktion viel komplexer sein, meist mit Zehntausenden Parametern, und das analytische Bestimmen der partiellen Ableitungen per Hand wäre eine nahezu unmögliche Aufgabe. Eine Lösung wäre, eine Näherung für jede partielle Ableitung zu berechnen, indem Sie messen, um wie viel sich die Ausgabe der Funktion ändert, wenn Sie den entsprechenden Parameter verändern:

```
>>> w1, w2 = 5, 3  
>>> eps = 1e-6  
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps  
36.000003007075065  
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps  
10.00000003174137
```

Das sieht ziemlich gut aus! Es funktioniert recht vernünftig und lässt sich leicht implementieren, aber es ist nur eine Näherung, und vor allem müssen Sie $f()$ mindestens einmal pro Parameter aufrufen (nicht zweimal, da wir $f(w_1, w_2)$ nur einmal berechnen müssen). Ein (mindestens) einmaliger Aufruf von $f()$ pro Parameter macht diesen Ansatz für große neuronale Netze zu unhandlich. Daher sollten wir Autodiff einsetzen. TensorFlow macht das ziemlich leicht:

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
```

```

with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])

```

Wir definieren zuerst zwei Variablen `w1` und `w2`, dann erstellen wir einen Kontext `tf.GradientTape`, der automatisch alle Operationen aufzeichnet, die mit einer Variablen zu tun haben, und schließlich bitten wir diese Aufzeichnung, die Gradienten des Ergebnisses `z` in Bezug auf die beiden Variablen `[w1, w2]` zu berechnen. Schauen wir uns die Gradienten an, die TensorFlow ermittelt hat:

```

>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]

```

Perfekt! Jetzt ist nicht nur das Ergebnis genau (die Präzision ist nur durch die Fehler der Gleitkommazahlen begrenzt), die Methode `gradient()` durchläuft zudem die aufgezeichneten Berechnungen nur einmal (in umgekehrter Reihenfolge) – egal, wie viele Variablen es gibt. Daher ist das unglaublich effizient – fast wie Magie!



Um Speicherplatz zu sparen, packen Sie nur das absolut Notwendige in den Block `tf.GradientTape()`. Alternativ pausieren Sie die Aufzeichnung durch das Erstellen eines Blocks `with tape.stop_recording()` innerhalb des Blocks `tf.GradientTape()`.

Die Aufzeichnung wird automatisch direkt nach dem Aufruf ihrer Methode `gradient()` gelöscht, daher erhalten Sie eine Exception, wenn Sie versuchen, die Methode zweimal aufzurufen:

```

with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # RuntimeError!

```

Müssen Sie `gradient()` mehr als einmal aufrufen, müssen Sie die Aufzeichnung persistent machen und selbst löschen, wenn Sie sie nicht mehr benötigen, um die Ressourcen wieder freizugeben:¹²

```

with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

```

```

dz_dw1 = tape.gradient(z, w1) # => Tensor 36.0
dz_dw2 = tape.gradient(z, w2) # => Tensor 10.0, klappt jetzt!
del tape

```

Standardmäßig zeichnet das Tape nur Operationen mit Variablen auf. Versuchen Sie also, den Gradienten von z bezüglich etwas anderem als einer Variablen zu berechnen, wird das Ergebnis None sein:

```

c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # liefert [None, None]

```

Aber Sie können die Aufzeichnung dazu zwingen, auch auf andere Dinge zu achten, um jede Operation zu dokumentieren, die diese betrifft. Sie können dann Gradienten bezüglich dieser Tensoren berechnen, als würde es sich bei ihnen um Variablen handeln:

```

with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # liefert [Tensor 36., Tensor 10.]

```

Das kann in manchen Situationen nützlich sein, zum Beispiel beim Implementieren eines Regularisierungsverlusts, der stark variierende Aktivierungen bei wenig variierenden Eingaben bestraft: Der Verlust basiert dann auf dem Gradienten der Aktivierung bezüglich der Eingaben. Da es sich bei Letzteren nicht um Variablen handelt, müssen Sie die Aufzeichnung dazu anweisen, auf sie zu achten.

Oft wird eine Gradientenaufzeichnung dazu genutzt, die Gradienten eines einzelnen Werts (meist des Verlusts) bezüglich eines Satzes von Werten (meist die Modellparameter) zu berechnen. Hier kann Autodiff im Reverse-Modus glänzen, da es nur einen Durchlauf vorwärts und einen rückwärts benötigt, um alle Gradienten auf einmal zu ermitteln. Versuchen Sie, die Gradienten eines Vektors zu berechnen – zum Beispiel mit mehreren Verlusten –, wird TensorFlow die Gradienten der Vektorsumme ermitteln. Benötigen Sie hingegen die einzelnen Gradienten (zum Beispiel die Gradienten jedes Verlusts bezüglich der Modellparameter), müssen Sie die Methode `jacobian()` der Aufzeichnung aufrufen: Sie führt das Reverse-Mode-Autodiff jeweils für jeden Verlust im Vektor aus (standardmäßig parallel). Es ist sogar möglich, zweite partielle

Ableitungen zu berechnen (also die Hessians – die partiellen Ableitungen der partiellen Ableitungen), aber das wird in der Praxis nur selten benötigt (im Abschnitt »Computing Gradientes with Autodiff« des Notebooks finden Sie ein Beispiel).

In manchen Fällen wollen Sie Gradienten in Teilen Ihres neuronalen Netzes nicht per Backpropagation verteilen. Dann müssen Sie die Funktion `tf.stop_gradient()` verwenden. Sie gibt ihre Eingabewerte während des Vorwärtsdurchgangs weiter (wie `tf.identity()`), lässt aber keine Gradienten während der Backpropagation durch (es agiert wie eine Konstante):

```
def f(w1, w2):
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

with tf.GradientTape() as tape:
    z = f(w1, w2) # gleiches Ergebnis wie ohne stop_gradient()

gradients = tape.gradient(z, [w1, w2]) # => liefert [Tensor 30., None]
```

Und schließlich können Sie gelegentlich über numerische Probleme beim Berechnen von Gradienten stolpern. Ermitteln Sie beispielsweise die Gradienten der Funktion `my_softplus()` für große Eingabewerte, wird das Ergebnis NaN sein:

```
>>> x = tf.Variable([100.])
>>> with tf.GradientTape() as tape:
...     z = my_softplus(x)
...
>>> tape.gradient(z, [x])
<tf.Tensor: [...] numpy=array([nan], dtype=float32)>
```

Das liegt daran, dass das Berechnen der Gradienten dieser Funktion mithilfe von Autodiff zu numerischen Schwierigkeiten führt: Aufgrund der beschränkten Genauigkeit von Gleitkommazahlen gerät Autodiff in die Verlegenheit, unendlich durch unendlich teilen zu müssen (was NaN erzeugt). Zum Glück können wir analytisch ermitteln, dass die Ableitung der Softplus-Funktion einfach $1 / (1 + 1 / \exp(x))$ und damit numerisch stabil ist. Wir können nun TensorFlow anweisen, diese stabile Funktion beim Berechnen der Gradienten von `my_softplus()` zu verwenden, indem wir sie mit `@tf.custom_gradient` dekorieren und sie sowohl den normalen Wert wie auch die Funktion zum Berechnen der Ableitungen zurückgeben lassen (beachten Sie, dass sie als Eingabewerte die bisher per Backpropagation verteilten Gradienten zur Softplus-Funktion erhält – wegen der Kettenregel sollten wir sie mit den Gradienten dieser Funktion multiplizieren):

```

@tf.custom_gradient

def my_better_softplus(z):
    exp = tf.exp(z)

    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)

    return tf.math.log(exp + 1), my_softplus_gradients

```

Berechnen wir jetzt die Gradienten der Funktion `my_better_softplus()`, erhalten wir das richtige Ergebnis selbst für große Eingabewerte (trotzdem explodiert das eigentliche Ergebnis aufgrund der Exponentialfunktion; ein Workaround ist die Verwendung von `tf.where()`, um die Eingabewerte zurückzugeben, wenn sie groß sind).

Herzlichen Glückwunsch! Sie können jetzt die Gradienten jeder Funktion berechnen (sofern sie dort differenzierbar ist, wo Sie sie ermitteln wollen), Backpropagation bei Bedarf blockieren und Ihre eigenen Gradientenfunktionen schreiben! Das ist vermutlich mehr Funktionalität, als Sie jemals brauchen werden, selbst wenn Sie Ihre eigenen Trainingsschleifen schreiben – wie wir jetzt sehen werden.

Eigene Trainingsschleifen

In sehr seltenen Fällen ist die Methode `fit()` eventuell nicht flexibel genug für das, was Sie tun wollen. So nutzt beispielsweise der Wide-&-Deep-Artikel (<https://homl.info/widedeep>), den wir in [Kapitel 10](#) besprochen haben, zwei verschiedene Optimierer: einen für den Wide-Pfad und einen anderen für den Deep-Pfad. Da die Methode `fit()` nur einen Optimierer einsetzt (den wir beim Kompilieren des Modells angegeben haben), müssen Sie zum Implementieren dieses Artikels Ihre eigene Schleife schreiben.

Vielleicht wollen Sie auch eigene Trainingsschleifen schreiben, um einfach mehr darauf vertrauen zu können, dass sie genau das tun, was sie tun sollen (möglicherweise sind Ihnen manche Details der `fit()`-Methode unklar). Es kann sich manchmal sicherer anfühlen, alles explizit zu machen. Aber denken Sie daran, dass das Schreiben einer eigenen Trainingsschleife Ihren Code länger, fehleranfälliger und schlechter wartbar machen wird.



Sofern Sie nicht wirklich die zusätzliche Flexibilität benötigen, sollten Sie besser die Methode `fit()` verwenden, statt Ihre eigene Trainingsschleife zu implementieren – insbesondere wenn Sie in einem Team arbeiten.

Bauen wir zunächst ein einfaches Modell. Wir müssen es nicht kompilieren, da wir die Trainingsschleife manuell ausführen werden:

```

l2_reg = keras.regularizers.l2(0.05)

model = keras.models.Sequential([

```

```

        keras.layers.Dense(30, activation="elu", kernel_initializer="he_normal",
                           kernel_regularizer=l2_reg),
        keras.layers.Dense(1, kernel_regularizer=l2_reg)
    ])

```

Als Nächstes erzeugen wir eine kleine Funktion, die zufällig einen Batch mit Instanzen aus dem Trainingsdatensatz auswählen wird (in [Kapitel 13](#) werden wir die Data-API behandeln, die eine viel bessere Alternative bietet):

```

def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]

```

Definieren wir auch eine Funktion, die den Trainingsstatus anzeigt – einschließlich des aktuellen Schritts, der Gesamtzahl an Schritten, des durchschnittlichen Verlusts seit Beginn der Epoche (wir werden die Metrik Mean zum Berechnen nutzen) und anderer Metriken:

```

def print_status_bar(iteration, total, loss, metrics=None):

    metrics = " - ".join(["{}: {:.4f}{}".format(m.name, m.result())
                           for m in [loss] + (metrics or [])])
    end = "" if iteration < total else "\n"
    print("\r{}/{} - {}".format(iteration, total) + metrics,
          end=end)

```

Dieser Code ist selbsterklärend, nur die String-Formatierung in Python mag verwirren: `{:.4f}` formatiert eine Gleitkommazahl mit vier Nachkommastellen, `\r` (Carriage Return) zusammen mit `end=""` stellt sicher, dass die Statusleiste immer auf der gleichen Zeile ausgegeben wird. Im Notebook enthält die Funktion `print_status_bar()` einen Fortschrittsbalken, aber Sie können stattdessen auch auf die praktische Bibliothek `tqdm` zurückgreifen.

Jetzt können wir ans Eingemachte gehen! Zuerst müssen wir Hyperparameter definieren und den Optimierer, die Verlustfunktion und die Metriken (in diesem Beispiel nur MAE) auswählen:

```

n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)

```

```

loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]

```

Nun können wir unsere eigene Schleife bauen!

```

for epoch in range(1, n_epochs + 1):
    print("Epoch {} / {}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
            gradients = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(gradients, model.trainable_variables))
            mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)
        print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
        print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics:
        metric.reset_states()

```

Da passiert ziemlich viel in diesem Code, daher schauen wir ihn uns genauer an:

- Wir erstellen zwei verschachtelte Schleifen: eine für die Epochen, die andere für die Batches in einer Epoche.
- Dann sammeln wir einen zufälligen Batch aus dem Trainingsdatensatz.
- Innerhalb des Blocks `tf.GradientTape()` erstellen wir eine Vorhersage für einen Batch (mit dem Modell als Funktion) und berechnen den Verlust: Er entspricht dem Hauptverlust plus den anderen Verlusten (in diesem Modell gibt es einen Regularisierungsverlust pro Schicht). Da die Funktion `mean_square_error()` einen Verlust pro Instanz liefert, berechnen wir den Mittelwert über den Batch mithilfe von

`tf.reduce_mean()` (wollten Sie unterschiedliche Gewichte für jede Instanz anwenden, würden Sie das hier tun). Die Regularisierungsverluste sind schon jeweils auf einen einzelnen Skalar reduziert, daher müssen wir sie nur noch aufsummieren (mit `tf.add_n()`, was mehrere Tensoren der gleichen Form und mit dem gleichen Datentyp addiert).

- Als Nächstes bitten wir die Aufzeichnung, die Gradienten des Verlusts in Bezug auf jede trainierbare Variable zu berechnen (*nicht* für alle Variablen!), und wenden sie auf den Optimierer an, um einen Gradientenschritt durchzuführen.
- Dann aktualisieren wir den durchschnittlichen Verlust und die Metriken (über die aktuelle Epoche) und geben den Statusbalken aus.
- Am Ende jeder Epoche zeigen wir den Statusbalken erneut an – dieses Mal als abgeschlossen¹³ –, geben einen Zeilenumbruch aus und setzen die Status des mittleren Verlusts und der Metriken zurück.

Setzen Sie den Hyperparameter `clipnorm` oder `clipvalue` des Optimierers, wird er sich für Sie darum kümmern. Wollen Sie andere Transformationen auf die Gradienten anwenden, tun Sie das einfach vor dem Aufruf der Methode `apply_gradients()`.

Ergänzen Sie Ihr Modell um Gewichts-Constraints (zum Beispiel durch das Setzen von `kernel_constraint` oder `bias_constraint` beim Erstellen einer Schicht), sollten Sie die Trainingsschleife so anpassen, dass sie diese Constraints direkt nach `apply_gradients()` anwendet:

```
for variable in model.variables:  
    if variable.constraint is not None:  
        variable.assign(variable.constraint(variable))
```

Am wichtigsten ist, dass diese Trainingsschleife nicht mit Schichten umgehen kann, die sich im Training und beim Testen unterschiedlich verhalten (zum Beispiel `BatchNormalization` oder `Dropout`). Dafür müssen Sie das Modell mit `training=True` aufrufen und sicherstellen, dass es dies an jede erforderliche Schicht weiterleitet.

Wie Sie sehen, gibt es eine Reihe von Dingen, die Sie richtig machen müssen, und es ist wirklich nicht schwer, sie falsch zu machen. Andererseits bekommen Sie so die volle Kontrolle – es ist also Ihre Entscheidung.

Nachdem Sie nun wissen, wie Sie alle möglichen Teile Ihres Modells und Ihrer Trainingsalgorithmen anpassen können,¹⁴ schauen wir uns an, wie Sie das Automatic Graph Generation Feature von TensorFlow einsetzen: Es kann Ihren eigenen Code deutlich beschleunigen und läuft dadurch portabel auf jeder Plattform, die von TensorFlow unterstützt wird (siehe [Kapitel 19](#)).

Funktionen und Graphen in TensorFlow

In TensorFlow 1 waren Graphen unvermeidbar (und damit leider auch die mit ihnen einhergehende Komplexität), da es sich bei ihnen um einen zentralen Teil der TensorFlow-API handelte. In TensorFlow 2 gibt es sie immer noch, aber nicht in so zentraler Rolle, und sie sind wirklich viel einfacher einzusetzen. Um zu zeigen, wie leicht die Arbeit mit ihnen ist, beginnen wir mit einer trivialen Funktion, die die dritte Potenz ihres Eingabewerts berechnet:

```
def cube(x):  
    return x ** 3
```

Wir können diese Funktion offensichtlich mit einem Python-Wert aufrufen, wie zum Beispiel einem int oder einem float, oder wir übergeben ihr einen Tensor:

```
>>> cube(2)  
8  
>>> cube(tf.constant(2.0))  
<tf.Tensor: id=18634148, shape=(), dtype=float32, numpy=8.0>
```

Verwenden wir jetzt `tf.function()`, um diese Python-Funktion in eine *TensorFlow-Funktion* zu konvertieren:

```
>>> tf_cube = tf.function(cube)  
>>> tf_cube  
<tensorflow.python.eager.def_function.Function at 0x1546fc080>
```

Diese TF Function kann genauso wie die ursprüngliche Python-Funktion eingesetzt werden und wird auch die gleichen Ergebnisse liefern (aber als Tensoren):

```
>>> tf_cube(2)  
<tf.Tensor: id=18634201, shape=(), dtype=int32, numpy=8>  
>>> tf_cube(tf.constant(2.0))  
<tf.Tensor: id=18634211, shape=(), dtype=float32, numpy=8.0>
```

Unter der Motorhaube hat `tf.function()` die Berechnungen analysiert, die die Funktion `cube()` ausführt, und einen äquivalenten Rechengraphen erstellt! Wie Sie sehen können, war das ziemlich schmerzfrei (wir werden gleich erfahren, wie es funktioniert). Alternativ hätten wir `tf.function` als Dekorierer verwenden können, was tatsächlich auch häufiger genutzt wird:

```
@tf.function
```

```
def tf_cube(x):  
    return x ** 3
```

Sollten Sie die ursprüngliche Python-Funktion doch noch benötigen, steht sie als Attribut `python_function` der TF Function zur Verfügung:

```
>>> tf_cube.python_function(2)
```

8

TensorFlow optimiert den Rechengraphen, schneidet ungenutzte Knoten weg, vereinfacht Ausdrücke (so würde beispielsweise $1 + 2$ durch 3 ersetzt) und so weiter. Ist der optimierte Graph fertig, führt die TF Function die Operationen im Graphen effizient in der passenden Reihenfolge (und wenn möglich parallel) aus. Damit läuft eine TF Function im Allgemeinen viel schneller als die ursprüngliche Python-Funktion, insbesondere beim Ausführen komplexer Berechnungen.¹⁵ Meistens werden Sie gar nicht mehr wissen müssen – wollen Sie eine Python-Funktion beschleunigen, verwandeln Sie sie einfach in eine TF Function, das ist alles!

Schreiben Sie eine eigene Verlustfunktion, Metrik, Schicht oder eine andere eigene Funktionen, die Sie in einem Keras-Modell einsetzen (wie wir es in diesem Kapitel getan haben), verwandelt Keras sie automatisch in eine TF Function – Sie müssen `tf.function()` also gar nicht verwenden. Die meiste Zeit werden Sie von dieser Magie also gar nichts mitbekommen.

- ☞ Sie können Keras anweisen, Ihre Python-Funktionen *nicht* in TF Functions umzuwandeln, indem Sie beim Erstellen einer eigenen Schicht oder eines eigenen Modells `dynamic=True` setzen. Alternativ können Sie beim Aufruf der Modellmethode `compile()` den Parameter `run_eagerly=True` setzen.

Standardmäßig erzeugt eine TF Function einen neuen Graphen für jeden eindeutigen Satz an Eingabeformen und Datentypen und speichert ihn für Folgeaufrufe zwischen. Rufen Sie beispielsweise `tf_cube(tf.constant(10))` auf, wird ein Graph für int32-Tensoren der Form `[]` erzeugt. Rufen Sie dann `tf_cube(tf.constant(20))` auf, wird der gleiche Graph wiederverwendet. Aber rufen Sie nun `tf_cube(tf.constant([10, 20]))` auf, wird ein neuer Graph für int32-Tensoren der Form `[2]` erzeugt. So können TF Functions mit Polymorphismus umgehen (also variierenden Argumenttypen und -formen). Aber das gilt nur für Tensoren-Argumente: Übergeben Sie numerische Python-Werte an eine TF Function, wird für jeden unterschiedlichen Wert ein neuer Graph erzeugt – `tf_cube(10)` und `tf_cube(20)` führen also zu zwei Graphen.

Rufen Sie eine TF Function viele Male mit unterschiedlichen numerischen Python-Werten auf, werden viele Graphen erzeugt, was Ihr Programm verlangsamt und viel RAM verbraucht (Sie müssen die TF Function löschen, um es wieder freizubekommen). Python-Werte sollten nur für Argumente genutzt werden, die wenige unterschiedliche Werte annehmen, zum Beispiel Hyperparameter wie die Anzahl an Neuronen pro Schicht. Damit kann TensorFlow jede Variante Ihres Modells besser optimieren.

AutoGraph und Tracing

Wie erzeugt TensorFlow nun die Graphen? Es beginnt damit, den Quellcode der Python-Funktion zu analysieren, um alle Kontrollflussanweisungen zu finden, wie zum Beispiel `for`-Schleifen, `while`-Schleifen und `if`-Anweisungen, aber auch `break`-, `continue`- und `return`-Anweisungen. Dieser erste Schritt wird als *AutoGraph* bezeichnet. TensorFlow muss den Quellcode analysieren, weil Python keine andere Möglichkeit bietet, Anweisungen zur Kontrollflussteuerung abzufangen: Es bietet magische Methoden wie `__add__()` und `__mul__()` für Operatoren wie `+` und `*`, aber es gibt keine magischen Methoden `__while__()` oder `__if__()`. Nach dem Analysieren des Funktionscodes gibt AutoGraph eine aktualisierte Version dieser Funktion zurück, in der alle Kontrollflussanweisungen durch die passenden TensorFlow-Operationen wie `tf.while_loop()` für Schleifen oder `tf.cond()` für `if`-Anweisungen ersetzt wurden. So analysiert AutoGraph beispielsweise in [Abbildung 12-4](#) den Quellcode der Python-Funktion `sum_squares()` und erzeugt die Funktion `tf_sum_squares()`. In dieser Funktion ist die `for`-Schleife durch die Definition der `loop_body()`-Funktion ersetzt (mit dem Rumpf der ursprünglichen `for`-Schleife), gefolgt von einem Aufruf der Funktion `for_stmt()`. Dieser Aufruf baut die passende `tf.while_loop()` im Rechengraphen.

Als Nächstes ruft TensorFlow diese »aktualisierte« Funktion auf, aber statt das Argument mitzugeben, übergibt es einen *symbolischen Tensor* – einen Tensor ohne echte Werte, sondern nur mit Namen, Datentyp und Form. Rufen Sie beispielsweise `sum_squares(tf.constant(10))` auf, wird die Funktion `tf_sum_squares()` mit einem symbolischen Tensor vom Typ `int32` und der Form `[]` aufgerufen. Die Funktion läuft dann in einem *Graph-Modus* – jede TensorFlow-Operation fügt einen Knoten im Graphen hinzu, der sie und ihre(n) Ausgabetensor(en) repräsentiert (im Gegensatz zum regulären Modus, der *Eager Execution* oder *Eager-Modus* genannt wird). Im Graph-Modus führen TF-Operationen keine Berechnungen durch. Das sollte Ihnen vertraut vorkommen, wenn Sie TensorFlow 1 kennen, da es sich beim Graph-Modus um den Standardmodus handelte. In [Abbildung 12-4](#) sehen Sie, dass die Funktion `tf_sum_squares()` mit einem symbolischen Tensor als Argument aufgerufen wird (in diesem Fall einem `int32`-Tensor der Form `[]`) und der endgültige Graph durch das Tracing erstellt wird. Die Knoten stehen für Operationen und die Pfeile für Tensoren (sowohl die generierte Funktion wie auch der Graph sind vereinfacht dargestellt).

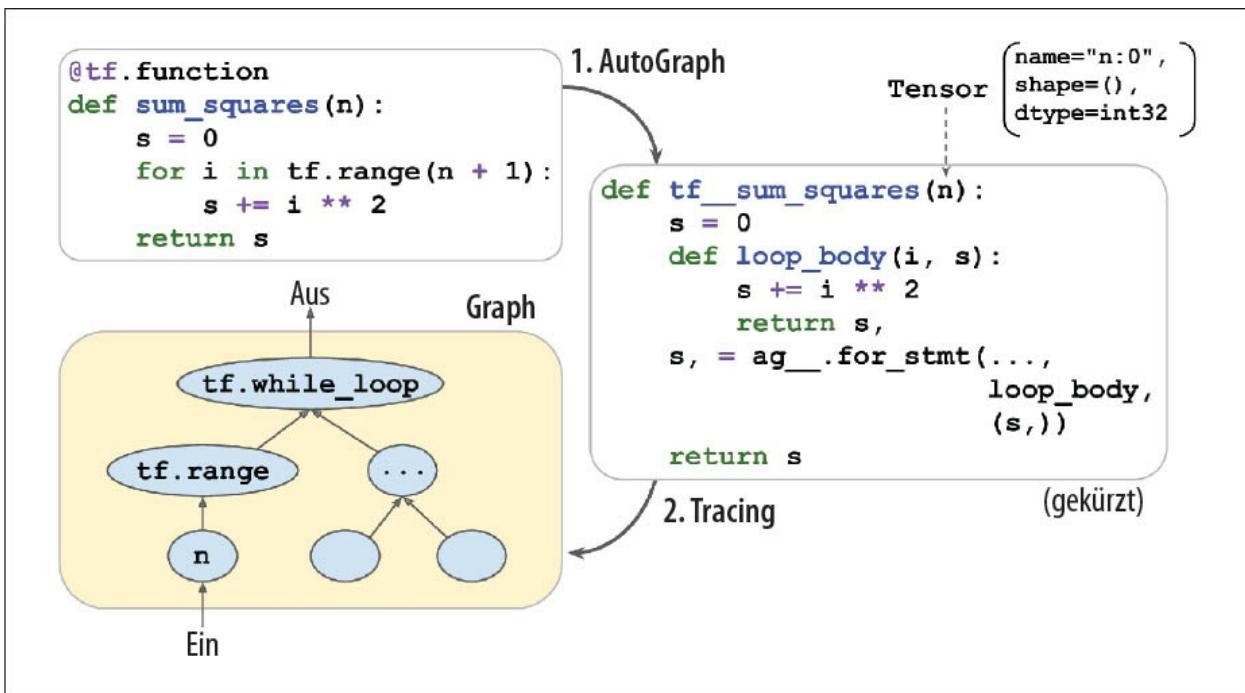


Abbildung 12-4: Wie TensorFlow Graphen mithilfe von AutoGraph und Tracing erzeugt



Um den Quellcode der generierten Funktion zu sehen, können Sie `tf.autograph.to_code(sum_squares.python_function)` aufrufen. Dieser Code soll nicht schön aussehen, kann aber manchmal beim Debuggen hilfreich sein.

Regeln für TF Functions

Sehr oft ist es trivial, eine Python-Funktion, die TensorFlow-Operationen ausführt, in eine TF Function umzuwandeln. Dekorieren Sie sie mit `@tf.function` oder lassen Sie Keras sich darum für Sie kümmern. Aber es gibt ein paar Regeln, die zu beachten sind:

- Rufen Sie eine externe Bibliothek auf, einschließlich NumPy, oder sogar die Standardbibliothek, wird dieser Aufruf nur beim Tracing durchgeführt und nicht Teil des Graphen sein. Ein TensorFlow-Graph kann tatsächlich nur TensorFlow-Konstrukte enthalten (Tensoren, Operationen, Variablen, Datensätze und so weiter). Stellen Sie also sicher, dass Sie `tf.reduce_sum()` statt `np.sum()`, `tf.sort()` statt der eingebauten Funktion `sorted()` und so weiter verwenden (sofern Sie nicht tatsächlich wollen, dass der Code nur während des Tracings läuft). Das hat ein paar zusätzliche Auswirkungen:
 - Definieren Sie eine TF Function `f(x)`, die nur `np.random.rand()` zurückgibt, wird nur dann eine Zufallszahl erzeugt, wenn die Funktion getraced wird. `f(tf.constant(2.))` und `f(tf.constant(3.))` liefern also die gleiche Zufallszahl zurück, aber `f(tf.constant([2., 3.]))` führt zu einem anderen Wert. Ersetzen Sie `np.random.rand()` durch `tf.random.uniform([])`, wird bei jedem Aufruf eine neue Zufallszahl generiert, da die Operation Teil des Graphen wird.
 - Hat Ihr Nicht-TensorFlow-Code Nebeneffekte (wie das Loggen von etwas oder

das Aktualisieren eines Python-Zählers), sollten Sie nicht davon ausgehen, dass diese Effekte bei jedem Aufruf der TF Function auftreten, da sie nur während des Tracings ausgeführt werden.

- Sie können beliebigen Python-Code in einer Operation `tf.py_func()` verpacken, aber das wird die Performance verringern, da TensorFlow für diesen Code keine Graphen-Optimierungen durchführen kann. Zudem reduziert sich die Portierbarkeit, da der Graph nur auf Plattformen läuft, auf denen Python vorhanden ist (und auf denen die richtigen Bibliotheken installiert sind).
- Sie können andere Python-Funktionen oder TF Functions aufrufen, diese sollten aber den gleichen Regeln folgen, da TensorFlow deren Operationen im Rechengraphen erfasst. Beachten Sie, dass diese anderen Funktionen nicht mit `@tf.function` dekoriert werden müssen.
- Erzeugt die Funktion eine TensorFlow-Variable (oder andere zustandsbehaftete TensorFlow-Objekte, wie zum Beispiel Datensätze oder eine Queue), muss dies beim ersten Aufruf geschehen und auch nur dann, denn sonst erhalten Sie eine Exception. Es ist normalerweise besser, Variablen außerhalb der TF Function zu erstellen (zum Beispiel in der Methode `build()` einer eigenen Schicht). Wollen Sie der Variablen einen neuen Wert zuweisen, achten Sie darauf, dass Sie deren Methode `assign()` aufrufen und nicht den `=`-Operator verwenden.
- Der Quellcode Ihrer Python-Funktion sollte für TensorFlow erreichbar sein. Steht er nicht zur Verfügung (zum Beispiel weil Sie Ihre Funktion in der Python-Shell definieren, die keinen Zugriff auf den Quellcode erlaubt, oder nur die kompilierten `*.pyc`-Dateien in die Produktivumgebung deployen), wird der Graph nicht erstellt werden können oder nur eingeschränkt funktionieren.
- TensorFlow erfasst nur `for`-Schleifen, die über einen Tensor oder einen Datensatz iterieren. Achten Sie daher darauf, `for i in tf.range(x)` zu nutzen statt `for i in range(x)`, denn sonst wird die Schleife im Graphen nicht ausgewertet. Stattdessen läuft sie während des Tracings. (Das kann erwünscht sein, wenn die `for`-Schleife den Graphen aufbauen soll, zum Beispiel um jede Schicht in einem neuronalen Netz zu erstellen.)
- Wie sonst auch sollten Sie aus Performancegründen wann immer möglich eine vektorisierte Implementierung bevorzugen, statt mit Schleifen zu arbeiten.

Zeit für eine Zusammenfassung. In diesem Kapitel haben wir einen kurzen Überblick über TensorFlow gegeben und uns dann dessen Low-Level-API angeschaut – Tensoren, Operationen, Variablen und spezielle Datenstrukturen. Diese haben wir dann genutzt, um so gut wie jede Komponente in `tf.keras` anzupassen. Und schließlich haben wir uns angeschaut, wie TF Functions die Performance verbessern können, wie Graphen mithilfe von AutoGraph und Tracing erstellt werden und welche Regeln zu befolgen sind, wenn Sie TF Functions schreiben (möchten Sie noch ein bisschen tiefer eintauchen, zum Beispiel zum Betrachten der erzeugten Graphen, finden Sie technische Details in [Anhang G](#)).

Im nächsten Kapitel werden wir uns anschauen, wie wir Daten mit TensorFlow effizient laden und vorverarbeiten.

Übungen

1. Wie würden Sie TensorFlow in einem kurzen Satz beschreiben? Was sind seine wichtigsten Features? Kennen Sie andere beliebte Deep-Learning-Bibliotheken?
2. Ist TensorFlow ein leicht auszutauschender Ersatz für NumPy? Was sind die Hauptunterschiede zwischen ihnen?
3. Erhalten Sie für `tf.range(10)` und `tf.constant(np.arange(10))` das gleiche Ergebnis?
4. Können Sie neben den normalen Tensoren sechs weitere Datenstrukturen aus TensorFlow benennen?
5. Eine eigene Verlustfunktion kann durch das Schreiben einer Funktion oder durch das Erstellen einer Subklasse von `keras.losses.Loss` erzeugt werden. Wann würden Sie welche Option verwenden?
6. Genauso kann eine eigene Metrik in einer Funktion oder als Unterklasse von `keras.metrics.Metric` erstellt werden. Wann nutzen Sie welche Option?
7. Wann sollten Sie eine eigene Schicht und wann ein eigenes Modell erstellen?
8. Was für Anwendungsfälle erfordern das Schreiben von Code für Ihre eigene Trainingsschleife?
9. Können eigene Keras-Komponenten beliebigen Python-Code enthalten, oder müssen sie in TF Functions konvertierbar sein?
10. Was sind die wichtigsten Regeln, wenn Sie eine Funktion in eine TF Function umwandeln können wollen?
11. Wann würden Sie ein dynamisches Keras-Modell erstellen müssen? Wie tun Sie das? Warum machen Sie nicht alle Ihre Modelle dynamisch?
12. Implementieren Sie eine eigene Schicht, die eine Schichtnormalisierung durchführt (wir werden solche Schichten in [Kapitel 15](#) einsetzen):
 - a. Die Methode `build()` sollte zwei trainierbare Gewichte α und β besitzen, die beide die Form `input_shape[-1:]` und den Datentyp `tf.float32` haben. α sollte mit 1s initialisiert werden, β mit 0s.
 - b. Die Methode `call()` sollte den Mittelwert μ und die Standardabweichung σ aller Features jeder Instanz berechnen. Dazu können Sie `tf.nn.moments(inputs, axes=-1, keepdims=True)` verwenden, das den Mittelwert μ und die Varianz σ^2 aller Instanzen liefert (die Wurzel der Varianz gibt Ihnen die Standardabweichung). Dann sollte die Funktion $\alpha \otimes (X - \mu) / (\sigma + \epsilon) + \beta$ berechnen und zurückgeben, wobei \otimes die elementweise Multiplikation (*) und ϵ ein glättender Term ist (eine kleine Konstante, die eine Division durch null vermeidet, zum Beispiel 0,001).
 - c. Stellen Sie sicher, dass Ihre eigene Schicht die gleichen (oder nahezu gleichen) Ergebnisse wie `keras.layers.LayerNormalization` liefert.
13. Trainieren Sie ein Modell mit einer eigenen Trainingsschleife, um den Fashion-MNIST-Datensatz anzugehen (siehe [Kapitel 10](#)).

- a. Zeigen Sie Epoche, Iteration, mittleren Trainingsverlust und mittlere Genauigkeit für jede Epoche an (aktualisiert bei jeder Iteration), dazu den Validierungsverlust und die Genauigkeit am Ende jeder Epoche.
- b. Probieren Sie verschiedene Optimierer mit anderen Lernraten für die oberen und unteren Schichten aus.

Lösungen zu diesen Übungen finden Sie in [Anhang A](#).

Daten mit TensorFlow laden und vorverarbeiten

Bisher haben wir nur Datensätze verwendet, die in den Speicher passen, aber Deep-Learning-Systeme werden häufig mit sehr großen Datensätzen trainiert, die nicht ins RAM passen. Das Aufnehmen und effiziente Vorverarbeiten eines großen Datensatzes kann mit anderen Deep-Learning-Bibliotheken nur mit Schwierigkeiten möglich sein, aber TensorFlow macht es dank seiner *Data-API* sehr einfach: Sie erstellen einfach ein Dataset-Objekt und teilen ihm mit, wo es die Daten findet und wie diese zu transformieren sind. TensorFlow kümmert sich um die ganzen Implementierungsdetails, wie zum Beispiel Multithreading, Queueing, Batching und Prefetching. Zudem arbeitet die Data-API nahtlos mit tf.keras zusammen!

Direkt aus der Box kann die Data-API aus Textdateien (wie CSV-Dateien), Binärdateien mit Datensätzen fester Größe und Binärdateien im TFRecord-Format von TensorFlow lesen, das Datensätze variabler Größe unterstützt. TFRecord ist ein flexibles und effizientes Binärformat, das auf Protocol Buffers basiert (einem Open-Source-Binärformat). Die Data-API bietet zudem Unterstützung für das Lesen aus SQL-Datenbanken. Und es gibt viele Open-Source-Erweiterungen, um aus allen möglichen Arten von Datenquellen zu lesen, wie zum Beispiel von Googles Big-Query-Service.

Das effiziente Lesen riesiger Datensätze ist nicht die einzige Schwierigkeit: Die Daten müssen auch vorverarbeitet und üblicherweise normalisiert werden. Zudem bestehen sie nicht immer nur aus normalen numerischen Feldern: Es kann Textmerkmale, kategorische Merkmale und so weiter geben. Diese müssen codiert werden, zum Beispiel per One-Hot-Codierung, Bag-of-Words-Codierung oder als *Embeddings* (wie wir sehen werden, handelt es sich bei einem Embedding um einen trainierbaren, dichten Vektor, der eine Kategorie oder ein Token repräsentiert). Eine Option für all diese Vorverarbeitung ist, eigene Vorverarbeitungsschichten zu schreiben. Eine andere ist der Einsatz der Standard-Vorverarbeitungsschichten, die Keras zur Verfügung stellt.

In diesem Kapitel werden wir die Data-API, das TFRecord-Format und das Erstellen eigener Vorverarbeitungsschichten sowie das Verwenden der Standardschichten von Keras behandeln. Auch werden wir uns kurz ein paar Projekte aus dem Ökosystem von TensorFlow anschauen:

TF Transform (tf.Transform)

Ermöglicht das Schreiben einer einzelnen Vorverarbeitungsfunktion, die vor dem eigentlichen Training (zu seiner Beschleunigung) im Batchmodus über Ihren kompletten Trainingsdatensatz läuft, dann in eine TF Function exportiert und in Ihr trainiertes Modell aufgenommen wird, sodass sie sich nach dem Deployen in die Produktivumgebung um das Vorverarbeiten neuer Instanzen kümmern kann.

TF Datasets (TFDS)

Stellt eine praktische Funktion zum Herunterladen vieler gebräuchlicher Datensätze aller Art bereit, einschließlich großer wie ImageNet. Zudem gibt es ein hilfreiches Dataset-Objekt, um die Daten mithilfe der Data-API zu bearbeiten.

Also legen wir los!

Die Data-API

Die gesamte Data-API ist um das Konzept eines *Datasets* herum aufgebaut: Wie Sie sicher vermuten, wird damit eine Folge von Datenobjekten abgebildet. Normalerweise werden Sie Datasets verwenden, die nach und nach Daten von der Festplatte lesen, aber aus Gründen der Einfachheit wollen wir mithilfe von `tf.data.Dataset.from_tensor_slices()` ein Dataset erstellen, das vollständig im RAM vorliegt:

```
>>> X = tf.range(10) # beliebiger Daten-Tensor  
>>> dataset = tf.data.Dataset.from_tensor_slices(X)  
>>> dataset  
<TensorSliceDataset shapes: (), types: tf.int32>
```

Die Funktion `from_tensor_slices()` erwartet einen Tensor und erstellt ein `tf.data.Dataset`, dessen Elemente alle die einzelnen »Slices« von X (entlang der ersten Dimension) sind, somit enthält dieser Datensatz 10 Elemente: die Tensoren 0, 1, 2, ..., 9. In diesem Fall hätten wir das gleiche Dataset erhalten, wenn wir `tf.data.Dataset.range(10)` genutzt hätten.

Sie können einfach über die Elemente eines Datasets iterieren:

```
>>> for item in dataset:  
...     print(item)  
...  
tf.Tensor(0, shape=(), dtype=int32)  
tf.Tensor(1, shape=(), dtype=int32)  
tf.Tensor(2, shape=(), dtype=int32)  
[...]  
tf.Tensor(9, shape=(), dtype=int32)
```

Transformationen verketten

Haben Sie erst einmal ein Dataset, können Sie alle möglichen Transformationen darauf

anwenden, indem Sie dessen Transformationsmethoden aufrufen. Jede Methode gibt ein neues Dataset zurück, sodass Sie die Transformationen wie folgt verketteten können (diese Kette ist in [Abbildung 13-1](#) dargestellt):

```
>>> dataset = dataset.repeat(3).batch(7)

>>> for item in dataset:
...     print(item)
...
...
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

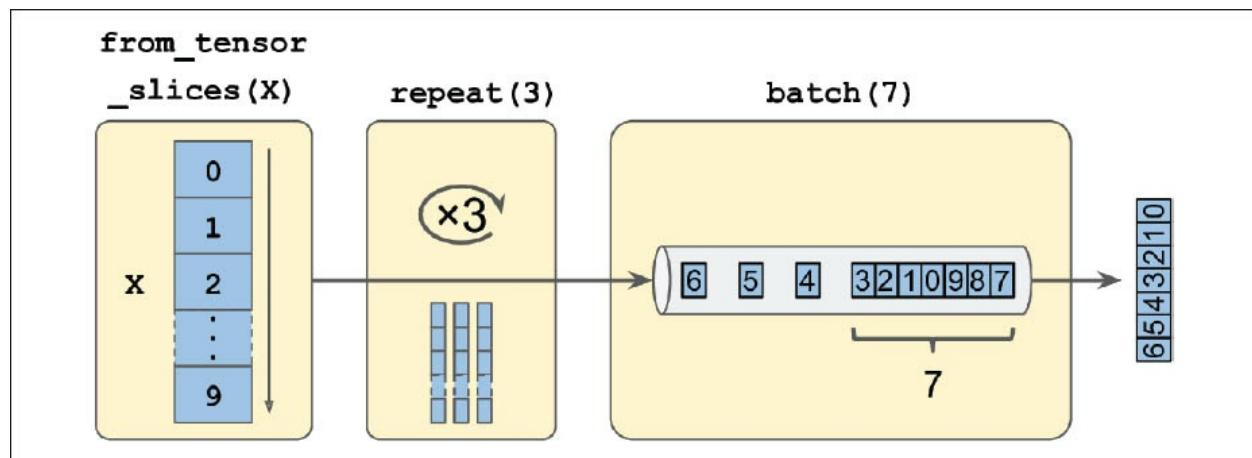


Abbildung 13-1: Dataset-Transformationen verketten

In diesem Beispiel rufen wir zuerst die Methode `repeat()` des ursprünglichen Datasets auf, die ein neues Dataset zurückliefert, das die Elemente des ersten Datasets drei Mal enthält. Natürlich werden damit nicht alle Daten drei Mal in den Speicher kopiert! (Wenn Sie diese Methode ohne Argumente aufrufen, wird das neue Dataset das Quell-Dataset unendlich oft wiederholen, sodass der Code, der darüber iteriert, selbst entscheiden muss, wann er aufhört.) Dann rufen wir die Methode `batch()` für dieses neue Dataset auf und erhalten wieder ein neues Dataset. Dieses wird die Elemente des vorherigen Datasets in Batches aus sieben Einheiten gruppieren. Schließlich iterieren wir über die Elemente dieses finalen Datasets. Wie Sie sehen können, musste die Methode `batch()` einen letzten Batch mit nur zwei statt sieben Elementen ausgeben, aber wenn Sie sie mit `drop_remainder=True` aufrufen, wird dieser letzte Batch verworfen, sodass alle Batches genau die gleiche Größe besitzen.



Die Dataset-Methoden *modifizieren* keine Datasets, sondern sie *erstellen* neue. Stellen Sie daher sicher, dass Sie eine Referenz auf diese neuen Datasets vorhalten (zum Beispiel mit `dataset = ...`), denn sonst wird nichts passieren.

Sie können die Elemente auch durch einen Aufruf der Methode `map()` transformieren. So wird beispielsweise ein neues Dataset mit verdoppelten Elementen wie folgt erzeugt:

```
>>> dataset = dataset.map(lambda x: x * 2) # Elemente: [0, 2, 4, 6, 8, 10, 12]
```

Diese Funktion rufen Sie auf, wenn Sie Ihre Daten vorverarbeiten wollen. Manchmal gehören dazu Berechnungen, die ziemlich umfassend sein können, wie zum Beispiel ein Reshaping oder das Rotieren eines Bilds, daher werden Sie normalerweise mehrere Threads nutzen wollen, um das Ganze zu beschleunigen. Dazu müssen Sie nur das Argument `num_parallel_calls` setzen. Beachten Sie, dass die Funktion, die Sie an die Methode `map()` übergeben, in eine TF Function umwandelbar sein muss (siehe [Kapitel 12](#)).

Während die Methode `map()` eine Transformation mit jedem Element durchführt, wendet die Methode `apply()` eine Transformation auf das Dataset als Ganzes an. So wendet beispielsweise der folgende Code die Funktion `unbatch()` auf das Dataset an (diese Funktion ist momentan noch experimentell, aber sie wird es sehr wahrscheinlich in einem zukünftigen Release in die zentrale API schaffen). Jedes Element im neuen Dataset ist dann ein Tensor mit einer einzelnen Ganzzahl statt ein Batch aus sieben Ganzzahlen:

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch()) # Elemente: 0, 2, 4 ...
```

Es ist auch möglich, das Dataset mit der Methode `filter()` einzuschränken:

```
>>> dataset = dataset.filter(lambda x: x < 10) # Elemente: 0 246 8024 6 ...
```

Sie werden häufig nur ein paar Elemente aus einem Dataset nutzen wollen. Dazu können Sie die Methode `take()` einsetzen:

```
>>> for item in dataset.take(3):
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
```

Daten durchmischen

Wie Sie wissen, funktioniert das Gradientenverfahren am besten, wenn die Instanzen im

Trainingsdatensatz unabhängig und gleichförmig verteilt sind (siehe [Kapitel 4](#)). Eine einfache Möglichkeit, das sicherzustellen, ist ein Durchmischen der Instanzen mithilfe der Methode `shuffle()`. Damit wird ein neues Dataset erstellt, das zunächst einen Puffer mit den ersten Elementen des Quell-Datasets befüllt. Wird es dann nach einem Element gefragt, holt es zufällig eines aus dem Puffer und ersetzt es durch ein frisches aus dem Quell-Dataset, bis es Letzteren vollständig durchiteriert hat. Danach holt es nur noch Elemente zufällig aus dem Puffer, bis auch dieser leer ist. Sie müssen die Puffergröße festlegen, und es ist wichtig, diese groß genug zu wählen, denn ansonsten wird das Durchmischen nicht sehr effektiv sein.¹ Überschreiten Sie nur nicht die Menge an verfügbarem RAM. Auch wenn Sie genug davon haben, müssen Sie nicht über die Größe des Datasets hinausgehen. Sie können einen Zufalls-Seed mitgeben, wenn Sie bei jeder Ausführung Ihres Programms die gleiche Zufallsreihenfolge nutzen wollen. Der folgende Code erzeugt beispielsweise ein Dataset mit den Werten 0 bis 9, die drei Mal wiederholt werden, mit einer Puffergröße von 5 durchmischt sind und einen Zufalls-Seed von 42 nutzen. Schließlich werden sie in Batches zu sieben Zahlen ausgegeben:

```
>>> dataset = tf.data.Dataset.range(10).repeat(3) # 0 bis 9, 3 Mal  
>>> dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)  
>>> for item in dataset:  
...     print(item)  
  
...  
tf.Tensor([0 2 3 6 7 9 4], shape=(7,), dtype=int64)  
tf.Tensor([5 0 1 1 8 6 5], shape=(7,), dtype=int64)  
tf.Tensor([4 8 7 1 2 3 0], shape=(7,), dtype=int64)  
tf.Tensor([5 4 2 7 8 9 9], shape=(7,), dtype=int64)  
tf.Tensor([3 6], shape=(2,), dtype=int64)
```

- ☞ Rufen Sie `repeat()` für ein durchmisches Dataset auf, wird standardmäßig bei jeder Iteration eine neue Reihenfolge erzeugt. Das ist im Allgemeinen eine gute Idee, aber wenn Sie immer die gleiche Reihenfolge nutzen wollen (zum Beispiel für Tests oder zum Debuggen), können Sie `reshuffle_each_iteration=False` setzen.

Bei einem großen Dataset, das nicht in den Speicher passt, reicht dieser einfache Durchmischungspuffer-Ansatz eventuell nicht aus, da der Puffer im Vergleich zum Dataset klein sein wird. Eine Lösung ist, die Quelldaten selbst zu durchmischen (so können Sie zum Beispiel unter Linux Textdateien mit dem Befehl `shuf` durchmischen). Das wird die Durchmischung definitiv verbessern! Aber auch wenn die Quelldaten vermischt sind, werden Sie sie noch mehr mischen wollen, denn ansonsten wird die gleiche Reihenfolge bei jeder Epoche genutzt, und das Modell ist am Ende vielleicht biased (zum Beispiel aufgrund irgendwelcher seltsamer Muster,

die zufällig in der Reihenfolge der Quelldaten vorhanden sind). Um die Instanzen noch mehr zu durchmischen, werden die Quelldaten oft in mehrere Dateien aufgeteilt, und diese werden dann während des Trainings in zufälliger Reihenfolge eingelesen. Aber Instanzen, die sich in derselben Datei befinden, werden danach trotzdem nahe beieinanderliegen. Um das zu vermeiden, können Sie zufällig mehrere Dateien auswählen, diese gleichzeitig einlesen und dabei die Datensätze verschränken. Danach können Sie mithilfe von `shuffle()` einen Durchmischungspuffer daraufsetzen. Wenn das alles nach viel Arbeit klingt, brauchen Sie sich keine Sorgen zu machen: Mit der Data-API sind das nur ein paar Zeilen Code. Schauen wir uns an, wie das geht.

Zeilen aus mehreren Dateien verschränken

Nehmen wir an, dass Sie den Datensatz mit den kalifornischen Immobilienpreisen geladen haben, er durchmischt wurde (sofern er nicht schon durchmischt war) und Sie ihn dann in einen Trainingsdatensatz, einen Validierungsdatensatz und einen Testdatensatz unterteilt haben. Anschließend wurde jeder Datensatz in viele CSV-Dateien unterteilt, die alle wie die folgende aussehen (jede Zeile enthält acht Eingabemerkmale und den Median des Hauswerts als Ziel):

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul,AveOccup,Lat,Long,MedianHouseValue  
3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442  
5.3275,5.0,6.4900,0.9910,3464.0,3.4433,33.69,-117.39,1.687  
3.1,29.0,7.5423,1.5915,1328.0,2.2508,38.44,-122.98,1.621  
[...]
```

Gehen wir außerdem davon aus, dass `train_filepaths` die Liste der Dateinamen der Trainingsdateien (mit Pfaden) enthält (und Sie auch `valid_filepaths` und `test_filepaths` besitzen):

```
>>> train_filepaths  
['datasets/housing/my_train_00.csv', 'datasets/housing/my_train_01.csv' ...]
```

Alternativ könnten Sie Muster für die Dateinamen verwenden, zum Beispiel `train_filepaths = "datasets/housing/my_train_*.csv"`. Erstellen wir nun ein Dataset nur mit diesen Dateien:

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

Standardmäßig gibt die Funktion `list_files()` ein Dataset zurück, das die Dateinamen durchmischt. Das ist normalerweise hilfreich, aber Sie können auch `shuffle=False` setzen, wenn Sie das aus irgendeinem Grund nicht wollen.

Als Nächstes können Sie die Methode `interleave()` aufrufen, um aus fünf Dateien

gleichzeitig zu lesen und deren Zeilen miteinander zu verschränken (mit der Methode `skip()` wird die erste Zeile jeder Datei mit dem Header übersprungen):

```
n_readers = 5

dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

Die Methode `interleave()` erzeugt ein Dataset, das sich fünf Dateinamen aus dem `filepath_dataset` holt und für jeden davon die übergebene Funktion aufruft (in diesem Beispiel ein Lambda), um ein neues Dataset zu erstellen (in diesem Fall ein `TextLineDataset`). Zu diesem Zeitpunkt gibt es insgesamt sieben Datasets: das Filepath-Dataset, das Interleave-Dataset und die fünf `TextLineDatasets`, die intern vom Interleave-Dataset erstellt wurden. Iterieren wir über das Interleave-Dataset, wird es diese fünf `TextLineDatasets` durchlaufen und jeweils reihum eine Zeile auslesen, bis alle Datasets am Ende angelangt sind. Dann wird es sich die nächsten fünf Dateinamen aus dem `filepath_dataset` holen, sie genauso verschränken und so weiter, bis auch die Dateinamen ausgegangen sind.



Damit das Verschränken besonders gut funktioniert, ist es sinnvoll, Dateien mit identischer Länge zu haben, denn ansonsten werden die Enden der längsten Dateien nicht mehr verschränkt.

Standardmäßig führt `interleave()` nichts parallel aus – es liest einfach eine Zeile nach der anderen sequenziell aus jeder Datei. Wollen Sie, dass die Dateien tatsächlich parallel gelesen werden, können Sie das Argument `num_parallel_threads` auf die gewünschte Anzahl an Threads setzen (beachten Sie, dass die Methode `map()` ebenfalls dieses Argument besitzt). Sie können es sogar auf `tf.data.experimental.AUTOTUNE` setzen, um TensorFlow selbst abhängig von der CPU-Verfügbarkeit dynamisch die richtige Zahl an Threads bestimmen zu lassen (das ist aber bisher noch ein experimentelles Feature). Schauen wir uns an, was das Dataset jetzt enthält:

```
>>> for line in dataset.take(5):
...     print(line.numpy())
...
b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782'
b'4.1812,52.0,5.7013,0.9965,692.0,2.4027,33.73,-118.31,3.215'
b'3.6875,44.0,4.5244,0.9930,457.0,3.1958,34.04,-118.15,1.625'
b'3.3456,37.0,4.5140,0.9084,458.0,3.2253,36.67,-121.7,2.526'
```

```
b'3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442'
```

Das sind die ersten Zeilen (wir ignorieren die Header-Zeile) von fünf zufällig gewählten CSV-Dateien. Sieht gut aus! Aber wie Sie sehen, handelt es sich nur um Bytestrings – wir müssen die Daten noch parsen und skalieren.

Daten vorverarbeiten

Implementieren wir eine kleine Funktion, die dieses Vorverarbeiten übernimmt:

```
X_mean, X_std = [...] # Mittelwert und Skala jedes Merkmals im  
# Trainingsdatensatz  
  
n_inputs = 8  
  
def preprocess(line):  
  
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]  
  
    fields = tf.io.decode_csv(line, record_defaults=defs)  
  
    x = tf.stack(fields[:-1])  
  
    y = tf.stack(fields[-1:])  
  
    return (x - X_mean) / X_std, y
```

Schauen wir uns diesen Code an:

- Als Erstes geht der Code davon aus, dass wir Mittelwert und Standardabweichung jedes Merkmals im Trainingsdatensatz vorab berechnet haben. `X_mean` und `X_std` sind eindimensionale Tensoren (oder NumPy-Arrays) mit acht Gleitkommawerten – einem pro Eingabemerkmals.
- Die Funktion `preprocess()` erwartet eine CSV-Zeile und parst diese. Dazu nutzt sie die Funktion `tf.io.decode_csv()`, die zwei Argumente erwartet: Das erste ist die zu parsende Zeile, das zweite ein Array mit dem Standardwert für jede Spalte in der CSV-Datei. Dieses Array informiert TensorFlow nicht nur über den Standardwert jeder Spalte, sondern auch über die Anzahl an Spalten und deren Typ. In diesem Beispiel legen wir fest, dass alle Merkmalsspalten Gleitkommazahlen sind und fehlende Werte durch 0 ersetzt werden sollen, zudem geben wir noch ein leeres Array vom Typ `tf.float32` als Standardwert für die letzte Spalte (das Ziel) mit: So weiß TensorFlow, dass diese Spalte Gleitkommazahlen enthält, es aber keinen Standardwert gibt und es eine Exception werfen soll, wenn es auf einen fehlenden Wert trifft.
- Die Funktion `decode_csv()` gibt eine Liste mit Skalar-Tensoren zurück (einen pro Spalte), aber wir müssen eindimensionale Tensor-Arrays liefern. Daher rufen wir `tf.stack()` für alle Tensoren außer den letzten (das Ziel) auf: Damit werden die

Tensoren in einem eindimensionalen Array zusammengefasst. Das Gleiche tun wir für den Zielwert (womit ein eindimensionales Tensor-Array mit einem einzelnen Wert statt einem Skalar-Tensor entsteht).

- Und schließlich skalieren wir die Eingabemerkmale, indem wir die Mittelwerte abziehen und dann durch die Standardabweichung teilen. Das Ergebnis geben wir als Tupel mit den skalierten Merkmalen und dem Ziel zurück.

Testen wir diese Vorverarbeitungsfunktion:

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')

(<tf.Tensor: id=6227, shape=(8,), dtype=float32, numpy=
array([ 0.16579159,  1.216324 , -0.05204564, -0.39215982, -0.5277444 ,
       -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,
<tf.Tensor: [...]>, numpy=array([2.782], dtype=float32)>)
```

Das sieht gut aus! Wir können jetzt die Funktion auf das Dataset anwenden.

Alles zusammenbringen

Um den Code wiederverwendbar zu machen, wollen wir alles bisher Besprochene in einer kleinen Hilfsfunktion zusammenbringen: Diese wird ein Dataset erstellen und zurückliefern, die Immobiliendaten effizient aus mehreren CSV-Dateien laden, sie vorverarbeiten, durchmischen, optional wiederholen und in Batches aufteilen (siehe Abbildung 13-2):

```
def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                      n_read_threads=None, shuffle_buffer_size=10000,
                      n_parse_threads=5, batch_size=32):

    dataset = tf.data.Dataset.list_files(filepaths)

    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)

    dataset = dataset.shuffle(shuffle_buffer_size).repeat(repeat)

    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)

    return dataset.batch(batch_size).prefetch(1)
```

In diesem Code sollte alles nachvollziehbar sein mit Ausnahme der letzten Zeile (`prefetch(1)`), die aus Performancegründen wichtig ist.

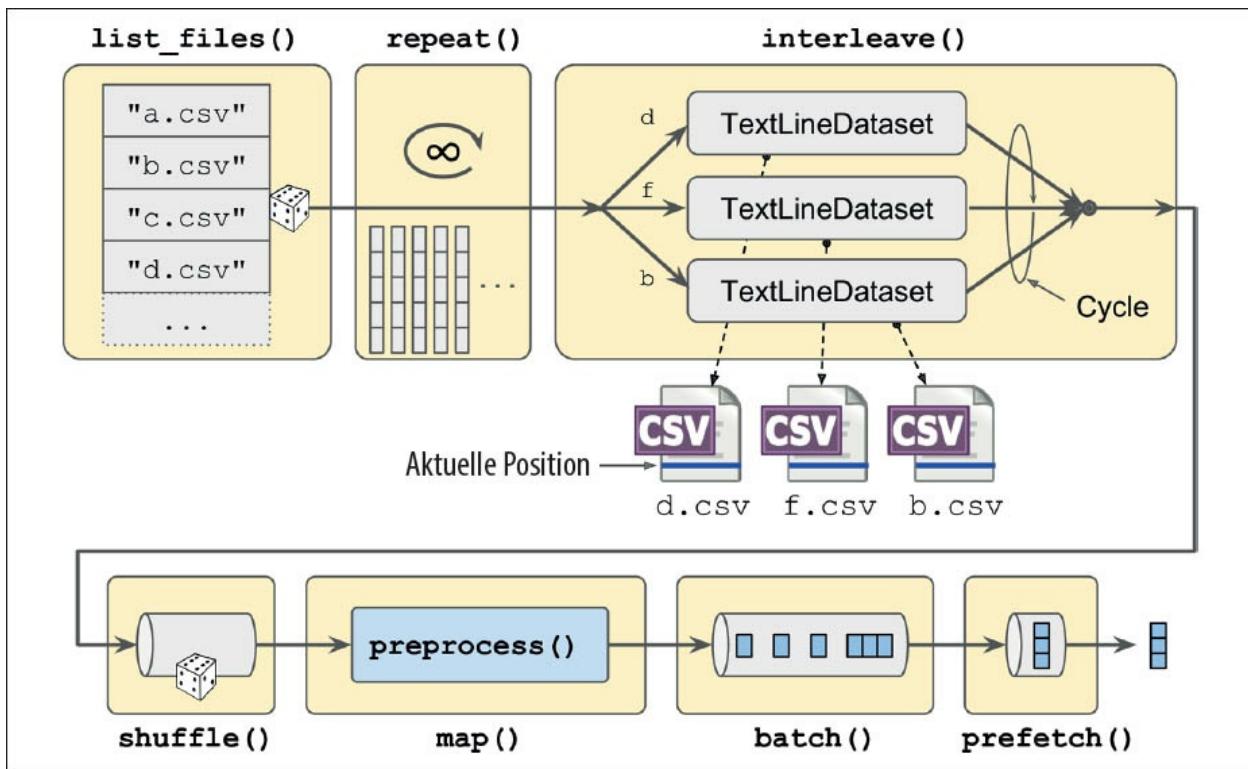


Abbildung 13-2: Laden und Vorverarbeiten von Daten aus mehreren CSV-Dateien

Prefetching

Durch den Aufruf von `prefetch(1)` am Ende der Funktion erstellen wir ein Dataset, das immer versuchen wird, einen Batch Vorsprung zu haben.² Mit anderen Worten: Während unser Trainingsalgorithmus an einem Batch arbeitet, ist das Dataset parallel dazu schon damit beschäftigt, den nächsten Batch vorzubereiten (zum Beispiel die Daten von der Festplatte zu lesen und vorzuverarbeiten). Das kann die Performance deutlich verbessern, wie in [Abbildung 13-3](#) zu sehen ist. Stellen wir zudem sicher, dass das Laden und Vorverarbeiten mit mehreren Threads geschieht (durch Setzen von `num_parallel_calls` beim Aufruf von `interleave()` und `map()`), können wir mehrere Cores der CPU ausnutzen und hoffentlich das Vorbereiten eines Datenbatchs in weniger Zeit schaffen, als das Ausführen eines Trainingsschritts auf der GPU dauert – so wird die GPU zu nahezu 100% ausgelastet (außer in der Zeit des Datentransfers von der CPU zur GPU³), und das Training wird deutlich schneller ablaufen.

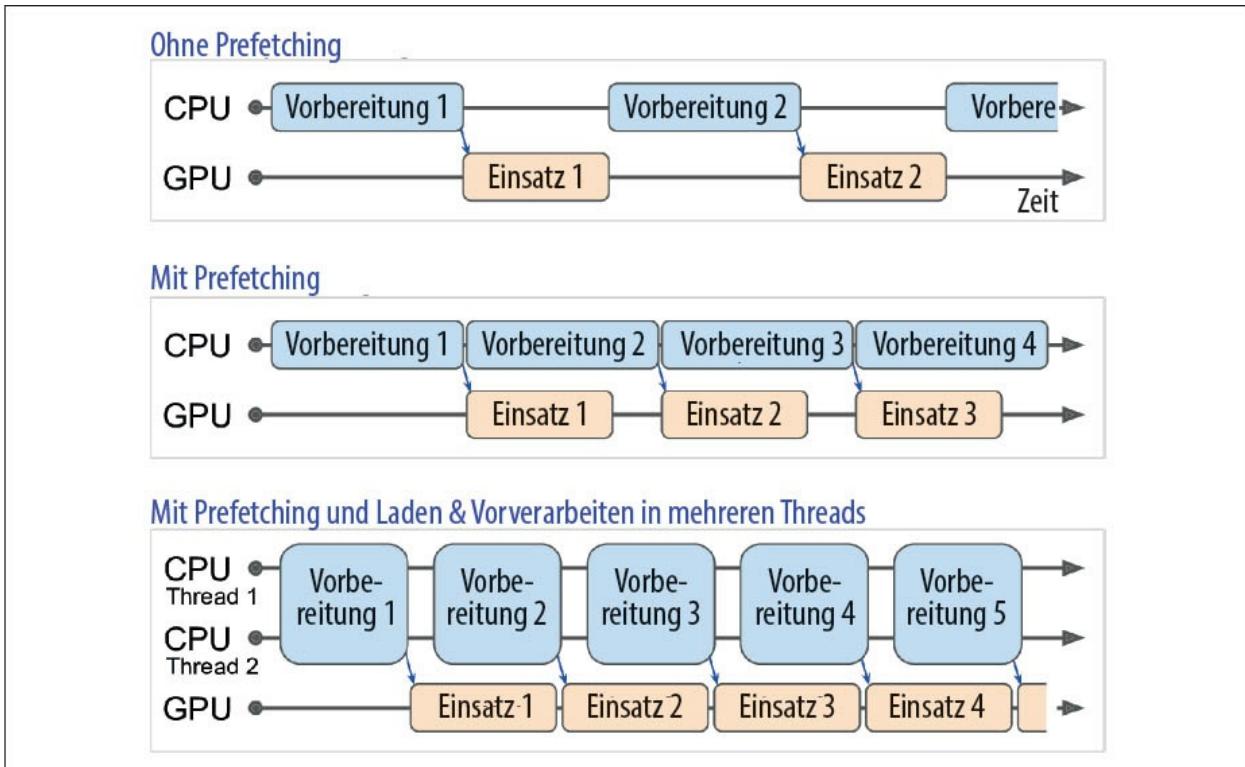


Abbildung 13-3: Durch Prefetching arbeiten CPU und GPU parallel: Während die GPU an einem Batch zu tun hat, bereitet die CPU den nächsten vor.

- Planen Sie, eine GPU-Karte zu kaufen, sind deren Rechenleistung und Speichergröße natürlich sehr wichtig (insbesondere ist viel RAM entscheidend für die Bildverarbeitung). Aber genauso wichtig für eine gute Performance ist die *Speicherbandbreite* – dabei handelt es sich um die Menge an Gigabyte von Daten, die pro Sekunde in das RAM oder dort heraus transferiert werden können.

Ist das Dataset klein genug, dass es in den Speicher passt, können Sie das Training deutlich beschleunigen, indem Sie die Methode `cache()` des Datasets nutzen, um dessen Inhalt im RAM zu puffern. Sie sollten das im Allgemeinen nach dem Laden und Vorverarbeiten der Daten, aber vor dem Durchmischen, Wiederholen, Aufteilen in Batches und Prefetching tun. So wird jede Instanz nur einmal gelesen und vorverarbeitet (statt einmal pro Epoche), aber die Daten werden trotzdem für jede Epoche unterschiedlich durchmischt, und der nächste Batch wird weiterhin im Voraus vorbereitet.

Sie wissen jetzt, wie Sie effiziente Eingabepipelines zum Laden und Vorverarbeiten von Daten aus mehreren Textdateien bauen können. Wir haben die am häufigsten zum Einsatz kommenden Methoden für Datasets besprochen, aber es gibt noch ein paar mehr, die Sie kennen sollten: `concatenate()`, `zip()`, `window()`, `reduce()`, `shard()`, `flat_map()` und `padded_batch()`. Es gibt auch noch ein paar mehr Klassenmethoden `from_generator()` und `from_tensor()`, die ein neues Dataset aus einem Python-Generator bzw. aus einer Liste von Tensoren erstellen. Schauen Sie für mehr Details in die API-Dokumentation. Beachten Sie auch, dass in `tf.experimental` weitere experimentelle Features zur Verfügung stehen, von

denen es viele in zukünftigen Releases vermutlich in die eigentliche API schaffen werden (so lohnt beispielsweise die Klasse `CsvDataset` einen Blick, aber auch die Methode `make_csv_dataset()`, die sich bemüht, den Datentyp jeder Spalte selbst zu ermitteln).

Datasets mit tf.keras verwenden

Jetzt können wir die Funktion `csv_reader_dataset()` nutzen, um ein Dataset für den Trainingsdatensatz zu erstellen. Beachten Sie, dass wir es nicht wiederholen müssen, da sich `tf.keras` schon darum kümmert. Wir erstellen auch Datasets für den Validierungs- und den Testdatensatz:

```
train_set = csv_reader_dataset(train_filepaths)

valid_set = csv_reader_dataset(valid_filepaths)

test_set = csv_reader_dataset(test_filepaths)
```

Jetzt können wir mit diesen Datasets ganz einfach ein Keras-Modell bauen und trainieren.⁴ Dazu müssen wir nur statt `X_train`, `y_train`, `X_valid` und `y_valid` die Trainings- und Validierungsdatasets an die Methode `fit()` übergeben:⁵

```
model = keras.models.Sequential(...)

model.compile(...)

model.fit(train_set, epochs=10, validation_data=valid_set)
```

Genauso können wir ein Dataset an die Methoden `evaluate()` und `predict()` übergeben:

```
model.evaluate(test_set)

new_set = test_set.take(3).map(lambda X, y: X) # als ob es 3 neue Instanzen gäbe

model.predict(new_set) # ein Dataset mit neuen Instanzen
```

Anders als die anderen Sets enthält das `new_set` normalerweise keine Labels (wenn es das tut, wird Keras sie ignorieren). Beachten Sie, dass Sie in all diesen Fällen immer noch NumPy-Arrays statt Datasets nutzen können, wenn Sie das möchten (aber natürlich müssen sie zuvor geladen und vorverarbeitet worden sein).

Wollen Sie Ihre eigene Trainingsschleife bauen (wie in [Kapitel 12](#)), können Sie ganz normal über das Trainings-Dataset iterieren:

```
for X_batch, y_batch in train_set:

    [...] # einen Gradientenschritt durchführen
```

Tatsächlich ist es sogar möglich, eine TF Function zu erstellen (siehe [Kapitel 12](#)), die die

gesamte Trainingsschleife ausführt:

```
@tf.function

def train(model, optimizer, loss_fn, n_epochs, [...]):
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, [...])

    for X_batch, y_batch in train_set:

        with tf.GradientTape() as tape:

            y_pred = model(X_batch)

            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))

            loss = tf.add_n([main_loss] + model.losses)

            grads = tape.gradient(loss, model.trainable_variables)

            optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Herzlichen Glückwunsch, Sie wissen nun, wie Sie leistungsfähige Eingabepipelines mit der Data-API bauen! Bisher haben wir CSV-Dateien genutzt, die sehr verbreitet sind, eine einfache Struktur haben und sich gut nutzen lassen. Aber sie sind nicht sehr effizient und unterstützen nur schlecht große oder komplexe Datenstrukturen (wie zum Beispiel Bilder oder Audiodaten). Schauen wir uns daher noch das TF-Record-Format an.

- 👉 Sind Sie mit CSV-Dateien mehr als zufrieden (oder welches Format Sie auch immer verwenden), müssen Sie TFRecords nicht einsetzen. Wie sagt man so schön: Wenn es nicht kaputt ist, reparieren Sie es nicht! TFRecords sind nützlich, wenn der Flaschenhals beim Trainieren das Laden und Parsen der Daten ist.

Das TFRecord-Format

Das TFRecord-Format ist das von TensorFlow bevorzugte Format zum Speichern großer Datenmengen, die es dann wieder effizient einlesen kann. Es handelt sich um ein einfaches Binärformat, das nur eine Folge von binären Datensätzen unterschiedlicher Länge enthält (jeder Datensatz besteht aus einer Länge, einer CRC-Prüfsumme, um kontrollieren zu können, dass die Länge nicht verfälscht wurde, dann den eigentlichen Daten und schließlich einer CRC-Prüfsumme für die Daten). Sie können eine TFRecord-Datei einfach mit der Klasse `tf.io.TFRecordWriter` erstellen:

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:
    f.write(b"Dies ist der erste Datensatz")
    f.write(b"And dies ist der zweite Datensatz")
```

Und Sie können ein `tf.data.TFRecordDataset` verwenden, um eine oder mehrere TFRecord-Dateien einzulesen:

```
filepaths = ["my_data.tfrecord"]

dataset = tf.data.TFRecordDataset(filepaths)

for item in dataset:
    print(item)
```

Das wird Folgendes ausgeben:

```
tf.Tensor(b'Dies ist der erste Datensatz', shape=(), dtype=string)
tf.Tensor(b'Und dies ist der zweite Datensatz', shape=(), dtype=string)
```

- * Standardmäßig liest ein `TFRecordDataset` die Dateien nacheinander ein, aber Sie können es auch Dateien parallel lesen und deren Datensätze verschränken lassen, indem Sie `num_parallel_reads` setzen. Alternativ erreichen Sie das gleiche Ergebnis auch durch Verwenden von `list_files()` und `interleave()`, so wie wir es zuvor beim Lesen mehrerer CSV-Dateien getan haben.

Komprimierte TFRecord-Dateien

Manchmal kann es sinnvoll sein, Ihre TFRecord-Dateien zu komprimieren, insbesondere wenn sie über eine Netzwerkverbindung geladen werden müssen. Sie können eine komprimierte TFRecord-Datei durch das Setzen des Arguments `options` erstellen:

```
options = tf.io.TFRecordOptions(compression_type="GZIP")

with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
```

Beim Lesen einer komprimierten TFRecord-Datei müssen Sie den Kompressionstyp angeben:

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                    compression_type="GZIP")
```

Eine kurze Einführung in Protocol Buffer

Auch wenn jeder Datensatz ein beliebiges Binärformat nutzen kann, enthalten TFRecord-Dateien normalerweise serialisierte Protocol Buffer (auch als *Protobufs* bezeichnet). Dabei handelt es sich um ein portables, erweiterbares und effizientes Binärformat, das von Google im Jahr 2001 entwickelt und 2008 zu Open Source gemacht wurde – Protobufs werden mittlerweile häufig eingesetzt, insbesondere in gRPC (<https://grpc.io>), Googles Remote-Procedure-Call-System. Sie sind mithilfe einer einfachen Sprache definiert, die wie folgt aussehen kann:

```

syntax = "proto3";

message Person {
    string name = 1;
    int32 id = 2;
    repeated string email = 3;
}

```

Diese Definition besagt, dass wir Version 3 des Protobuf-Formats verwenden, und sie legt fest, dass jedes Person-Objekt (optional) einen name vom Typ `string`, eine id vom Typ `int32` und null oder mehr email-Felder (vom Typ `string`) hat. Die Zahlen 1, 2 und 3 sind die Feldkennungen: Sie werden in der Binärrepräsentation jedes Datensatzes verwendet. Haben Sie eine Definition in einer `.proto`-Datei, können Sie sie kompilieren. Dazu ist `protoc` (der Protobuf-Compiler) erforderlich, der Zugriffsklassen in Python (oder einer anderen Sprache) generiert. Beachten Sie, dass die Protobuf-Definitionen, die wir verwenden, schon für Sie kompiliert wurden und deren Python-Klassen Teil von TensorFlow sind, daher werden Sie `protoc` nicht benötigen. Sie müssen nur wissen, wie Sie die Protobuf-Zugriffsklassen in Python einsetzen. Um die Grundlagen zu illustrieren, wollen wir uns ein einfaches Beispiel anschauen, das die für den Protobuf Person generierten Zugriffsklassen einsetzt (der Code ist in den Kommentaren erläutert):

```

>>> from person_pb2 import Person # generierte Zugriffsklasse importieren
>>> person = Person(name="Al", id=123, email=["a@b.com"]) # Person erstellen
>>> print(person) # Person anzeigen
name: "Al"
id: 123
email: "a@b.com"
>>> person.name # ein Feld lesen
"Al"
>>> person.name = "Alice" # ein Feld verändern
>>> person.email[0] # wiederholt vorkommende Felder
>>> # können wie Arrays angesprochen werden
"a@b.com"
>>> person.email.append("c@d.com") # E-Mail-Adresse hinzufügen

```

```

>>> s = person.SerializeToString() # Objekt in einen Bytestring serialisieren
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person() # neue Person erstellen
>>> person2.ParseFromString(s) # Bytestring parsen (27 Byte lang)
27
>>> person == person2 # jetzt sind sie gleich
True

```

Kurz gesagt, importieren wir die von protoc generierte Klasse Person, erstellen eine Instanz und spielen mit ihr, geben sie aus und lesen und schreiben ein paar Felder, dann serialisieren wir sie mit der Methode `SerializeToString()`. Das sind dann die Binärdaten, die gespeichert oder über das Netzwerk geschickt werden können. Beim Lesen oder Empfangen dieser Binärdaten können wir sie mit der Methode `ParseFromString()` parsen und erhalten eine Kopie des Objekts, das serialisiert wurde.⁶

Wir könnten das serialisierte Person-Objekt in einer TFRecord-Datei sichern und sie dann wieder laden und parsen – alles würde funktionieren. Aber `SerializeToString()` und `ParseFromString()` sind keine TensorFlow-Operationen (und auch die anderen Operationen in diesem Code nicht), daher können sie nicht mit in eine TF Function aufgenommen werden (außer durch ein Verpacken in eine `tf.py_func()`-Operation, was den Code langsamer und weniger gut portierbar machen würde, wie wir in [Kapitel 12](#) gesehen haben). Zum Glück besitzt TensorFlow spezielle Protobuf-Definitionen, für die es Parsing-Operationen bereitstellt.

TensorFlow-Protobufs

Der wichtigste Protobuf, der typischerweise in einer TFRecord-Datei genutzt wird, ist der Example-Protobuf, der eine Instanz in einem Dataset repräsentiert. Er enthält eine Liste benannter Merkmale, von denen jedes entweder eine Liste von Bytestrings, Floats oder Integern sein kann. Dies ist die Protobuf-Definition:

```

syntax = "proto3";

message BytesList { repeated bytes value = 1; }

message FloatList { repeated float value = 1 [packed = true]; }

message Int64List { repeated int64 value = 1 [packed = true]; }

message Feature {
    oneof kind {
        BytesList bytes_list = 1;
    }
}

```

```

    FloatList float_list = 2;

    Int64List int64_list = 3;

}

};

message Features { map<string, Feature> feature = 1; };

message Example { Features features = 1; };

```

Die Definitionen von BytesList, FloatList und Int64List sind klar genug. Beachten Sie, dass [packed = true] für eine effizientere Codierung für wiederholte numerische Felder genutzt wird. Feature enthält entweder eine BytesList, eine FloatList oder eine Int64List. Ein Features (mit einem s) enthält ein Dictionary, das einen Merkmalsnamen auf einen entsprechenden Merkmalswert abbildet.⁷ Und schließlich enthält ein Example nur ein Features-Objekt. So könnten Sie ein tf.train.Example erstellen, das dieselbe Person wie oben repräsentiert, und es in eine TFRecord-Datei schreiben:

```

from tensorflow.train import BytesList, FloatList, Int64List

from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                b"c@d.com"]))
        }))

```

Der Code ist ein bisschen lang und wiederholt sich, aber er ist ziemlich klar (und Sie könnten ihn problemlos in einer kleinen Hilfsfunktion verpacken). Nachdem wir nun einen Example-Protobuf haben, können wir ihn durch den Aufruf der Methode `SerializeToString()` serialisieren und die sich daraus ergebenden Daten in eine TFRecord-Datei schreiben:

```

with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())

```

Normalerweise würden Sie viel mehr als nur ein Example schreiben! Typischerweise erstellen Sie ein Konvertierungsskript, das aus Ihrem aktuellen Format liest (zum Beispiel einer CSV-Datei), einen Example-Protobuf für jede Instanz erstellt, sie serialisiert und dann in verschiedene TFRecord-Dateien schreibt – und sie idealerweise auch noch durchmischt. Das ist ein bisschen Arbeit, daher sollten Sie sich sicher sein, dass sie erforderlich ist (vielleicht funktioniert Ihre Pipeline ja wunderbar mit CSV-Dateien).

Nachdem wir nun eine nette TFRecord-Datei mit einem serialisierten Example haben, wollen wir versuchen, sie zu laden.

Examples laden und parsen

Um den serialisierten Example-Protobuf zu laden, werden wir erneut ein `tf.data.TFRecordDataset` verwenden und jedes Example mithilfe von `tf.io.parse_single_example()` parsen. Dies ist eine TensorFlow-Operation, daher kann sie in eine TF Function aufgenommen werden. Es sind mindestens zwei Argumente erforderlich: ein Stringskalar-Tensor mit den serialisierten Daten und eine Beschreibung jedes Features. Die Beschreibung ist ein Dictionary, das jeden Feature-Namen entweder auf einen `tf.io.FixedLenFeature`-Deskriptor mit Form, Typ und Standardwert des Features oder auf einen `tf.io.VarLenFeature`-Deskriptor nur mit dem Typ abbildet (Letzterer kommt zum Einsatz, wenn die Länge der Feature-Liste variieren kann, wie zum Beispiel für das `emails`-Feature).

Der folgende Code definiert erst ein Deskriptor-Dictionary, dann iteriert er über das `TFRecordDataset` und parst den serialisierten Example-Protobuf, den dieses Dataset enthält:

```
feature_description = {  
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),  
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),  
    "emails": tf.io.VarLenFeature(tf.string),  
}  
  
for serialized_example in tf.data.TFRecordDataset(["my_contacts.tfrecord"]):  
    parsed_example = tf.io.parse_single_example(serialized_example,  
                                                feature_description)
```

Die Features mit fester Länge werden als normale Tensoren geparsert, die mit variabler Länge hingegen als Sparse-Tensoren. Sie können einen Sparse-Tensor mithilfe von `tf.sparse.to_dense()` in einen Dense-Tensor umwandeln, aber in diesem Fall ist es sinnvoller, einfach auf die Werte zuzugreifen:

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
```

```

<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>

```

Eine BytesList kann beliebige Binärdaten enthalten, einschließlich serialisierter Objekte. So können Sie beispielsweise `tf.io.encode_jpeg()` nutzen, um ein Bild im JPEG-Format zu codieren, und diese Binärdaten in eine BytesList stecken. Wenn Ihr Code später den TFRecord ausliest, wird er mit dem Parsen des Example beginnen und dann `tf.io.decode_jpeg()` aufrufen müssen, um die Daten zu parsen und das Originalbild zu erhalten (oder Sie können `tf.io.decode_image()` verwenden, das BMP-, GIF-, JPEG- und PNG-Bilder decodieren kann). Sie können auch einen beliebigen Tensor in einer BytesList ablegen, indem Sie ihn mit `tf.io.serialize_tensor()` serialisieren und dann den daraus entstehenden Bytestring in ein Bytes List-Feature stecken. Wenn Sie später den TFRecord parsen, können Sie diese Daten mit `tf.io.parse_tensor()` wieder herausholen.

Statt ein Example nach dem anderen mit `tf.io.parse_single_example()` zu parsen, können Sie sie auch Batch für Batch mit `tf.io.parse_example()` parsen:

```

dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(10)

for serialized_examples in dataset:
    parsed_examples = tf.io.parse_example(serialized_examples,
                                           feature_description)

```

Wie Sie sehen, wird der Example-Protobuf in den meisten Fällen vermutlich ausreichen. Aber er kann ein bisschen unbequem im Einsatz sein, wenn Sie mit Listen von Listen arbeiten. Stellen Sie sich beispielsweise vor, Sie wollten Textdokumente klassifizieren. Jedes Dokument wird vielleicht durch eine Liste mit Sätzen dargestellt, wobei jeder Satz aus einer Liste mit Wörtern besteht. Und vielleicht hat jedes Dokument auch noch eine Liste mit Kommentaren, wobei jeder Kommentar als Liste mit Wörtern repräsentiert wird. Es kann darüber hinaus noch kontextbezogene Daten geben, wie zum Beispiel Autor, Titel und Datum der Veröffentlichung. Für solche Anwendungsfälle ist der SequenceExample-Protobuf von TensorFlow gedacht.

Listen von Listen mit dem SequenceExample-Protobuf verarbeiten

Dies ist die Definition des SequenceExample-Protobufs:

```

message FeatureList { repeated Feature feature = 1; };

message FeatureLists { map<string, FeatureList> feature_list = 1; };

message SequenceExample {
    Features context = 1;
}

```

```
    FeatureLists feature_lists = 2;  
};
```

Ein SequenceExample enthält ein Features-Objekt für die Kontextdaten und ein FeatureLists-Objekt mit einem oder mehreren benannten FeatureList-Objekten (z.B. eine FeatureList namens "content" und eine andere namens "comments"). Jede FeatureList enthält eine Liste mit Feature-Objekten, von denen jedes eine Liste mit Bytestrings, 64-Bit-Integer-Werten oder Gleitkommazahlen sein kann (in diesem Beispiel würde jedes Feature einen Satz oder einen Kommentar darstellen, vielleicht in Form einer Liste von Wortkennungen). Sie bauen, serialisieren und parsen ein SequenceExample genauso wie ein Example, aber Sie müssen `tf.io.parse_single_sequence_example()` zum Parsen eines einzelnen SequenceExample oder `tf.io.parse_sequence_example()` für einen Batch verwenden. Beide Funktionen geben ein Tupel mit den Kontext-Features (als Dictionary) und der Feature-Listen (ebenfalls als Dictionary) zurück. Enthalten die Feature-Listen Sequenzen unterschiedlicher Größe (wie im vorherigen Beispiel), wollen Sie sie eventuell per `tf.RaggedTensor.from_sparse()` in einen Ragged-Tensor konvertieren (im Notebook finden Sie den gesamten Code):

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(  
    serialized_sequence_example, context_feature_descriptions,  
    sequence_feature_descriptions)  
  
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

Jetzt wissen Sie, wie Sie Daten effizient speichern, laden und parsen können. Im nächsten Schritt bereiten Sie sie so auf, dass sie in ein neuronales Netz gespeist werden können.

Die Eingabemerkmale vorverarbeiten

Zum Vorbereiten Ihrer Daten für ein neuronales Netz müssen Sie alle Merkmale in numerische Merkmale umwandeln, sie im Allgemeinen normalisieren und noch andere Dinge mit ihnen tun. Insbesondere wenn Ihre Daten kategorische Merkmale oder Textmerkmale sind, müssen sie in Zahlen umgewandelt werden. Das kann im Voraus beim Vorbereiten Ihrer Datendateien mit einem passenden Tool geschehen (zum Beispiel NumPy, pandas oder Scikit-Learn). Alternativ können Sie Ihre Daten während des Ladens mit der Data-API vorverarbeiten (zum Beispiel mit der Dataset-Methode `map()`, wie Sie weiter oben gesehen haben), oder Sie nehmen eine Vorverarbeitungsschicht direkt in Ihr Modell mit auf. Schauen wir uns diese letzte Option genauer an.

So können Sie beispielsweise eine Standardisierungsschicht mit einer Lambda-Schicht implementieren. Bei jedem Merkmal wird der Mittelwert abgezogen, und es wird durch seine Standardabweichung dividiert (plus einen kleinen Glättungsterm, um eine Division durch null zu

vermeiden):

```
means = np.mean(X_train, axis=0, keepdims=True)
stds = np.std(X_train, axis=0, keepdims=True)
eps = keras.backend.epsilon()
model = keras.models.Sequential([
    keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),
    [...] # andere Schichten
])
```

Das ist nicht allzu schwer! Aber vielleicht bevorzugen Sie eine nette kleine eigene Schicht (so wie der `StandardScaler` in Scikit-Learn), statt sich mit globalen Variablen wie `means` und `stds` herumzuschlagen:

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs - self.means_) / (self.stds_ + keras.backend.epsilon())
```

Bevor Sie diese Standardisierungsschicht nutzen können, werden Sie sie durch Aufruf der Methode `adapt()` auf Ihre Daten anpassen müssen und dabei ein Datenbeispiel mitliefern. Damit kann die Schicht die passenden Mittelwerte und Standardabweichungen für jedes Merkmal verwenden:

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

Dieses Beispiel muss groß genug sein, um Ihren Datensatz zu repräsentieren, aber es muss sich nicht um den vollständigen Trainingsdatensatz handeln: Im Allgemeinen reichen ein paar Hundert zufällig gewählte Instanzen aus (aber das hängt von Ihrer Aufgabe ab). Danach können Sie diese Vorverarbeitungsschicht wie eine normale Schicht einsetzen:

```
model = keras.Sequential()
model.add(std_layer)
```

```
[...] # den Rest des Modells erzeugen
model.compile([...])
model.fit([...])
```

Sollten Sie der Meinung sein, dass Keras schon solch eine Standardisierungsschicht mitbringen müsste, habe ich gute Nachrichten für Sie: Wenn Sie diese Zeilen lesen, wird die Schicht `keras.layers.Normalization` vermutlich verfügbar sein. Sie funktioniert sehr ähnlich wie unsere eigene Standardization-Schicht: Erst wird die Schicht erstellt, dann wird sie an Ihre Daten angepasst, indem der Methode `adapt()` Beispiele mitgegeben werden, und schließlich wird die Schicht ganz normal eingesetzt.

Jetzt wollen wir uns den kategorischen Merkmalen zuwenden. Beginnen wir mit dem Codieren als One-Hot-Vektoren.

Kategorische Merkmale mit One-Hot-Vektoren codieren

Denken Sie an das Merkmal `ocean_proximity` im Datensatz mit den kalifornischen Immobilienpreisen, den wir in [Kapitel 2](#) genutzt haben: Es handelt sich um ein kategorisches Merkmal mit fünf möglichen Werten: "`<1H OCEAN`", "`INLAND`", "`NEAR OCEAN`", "`NEAR BAY`" und "`ISLAND`". Wir müssen dieses Merkmal codieren, bevor wir es an ein neuronales Netz übergeben können. Da es nur sehr wenige Kategorien gibt, können wir eine One-Hot-Codierung verwenden. Dazu müssen wir zuerst jede Kategorie auf ihren Index abbilden, was über eine Lookup-Tabelle abläuft:

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices = tf.range(len(vocab), dtype=tf.int64)
table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets = 2
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)
```

Gehen wir diesen Code durch:

- Wir definieren zuerst das *Vokabular* – die Liste aller möglichen Kategorien.
- Dann erstellen wir einen Tensor mit den zugehörigen Indizes (0 bis 4).
- Anschließend erstellen wir einen Initialisierer für die Lookup-Tabelle und übergeben ihm die Liste an Kategorien und ihre zugehörigen Indizes. In diesem Beispiel haben wir diese Daten schon, daher nutzen wir einen `KeyValue TensorInitializer`. Würden sich die Kategorien in einer Textdatei befinden (mit einer Kategorie pro Zeile), würden wir stattdessen einen `TextFileInitializer` nutzen.
- In den letzten beiden Zeilen erstellen wir die Lookup-Tabelle, übergeben den Initialisierer und legen die Anzahl der *Out-of-Vocabulary*- (OOV-)Buckets fest. Schlagen wir eine Kategorie nach, die nicht im Vokabular enthalten ist, berechnet die

Lookup-Tabelle einen Hash dieser Kategorie und nutzt ihn, um die unbekannte Kategorie einem der OOV-Buckets zuzuweisen. Deren Indizes beginnen nach den bekannten Kategorien, daher sind in diesem Beispiel die Indizes der zwei OOV-Buckets 5 und 6.

Warum brauchen wir OOV-Buckets? Nun, wenn es viele Kategorien gibt (zum Beispiel Postleitzahlen, Städte, Wörter, Produkte oder Benutzer) und der Datensatz ebenfalls umfangreich ist oder sich immer wieder ändert, kann es praktisch sein, die vollständige Liste der Kategorien abrufen zu können. Eine Möglichkeit ist, das Vokabular auf Datenbeispielen basieren zu lassen (statt auf dem gesamten Trainingsdatensatz) und ein paar OOV-Buckets für die anderen Kategorien hinzuzufügen, die sich nicht darin befanden. Je mehr unbekannte Kategorien Sie während des Trainings erwarten, desto mehr OOV-Buckets sollten Sie nutzen. Gibt es nicht genug Buckets, führt das zu Kollisionen – unterschiedliche Kategorien werden im selben Bucket landen, und das neuronale Netz wird nicht zwischen ihnen unterscheiden können (zumindest nicht basierend auf diesem Merkmal).

Nutzen wir nun diese Lookup-Tabelle, um einen kleinen Batch kategorischer Merkmale in One-Hot-Vektoren zu codieren:

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])

>>> cat_indices = table.lookup(categories)

>>> cat_indices

<tf.Tensor: id=514, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>

>>> cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)

>>> cat_one_hot

<tf.Tensor: id=524, shape=(4, 7), dtype=float32, numpy=
array([[0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.]], dtype=float32)>
```

Wie Sie sehen, wurde "NEAR BAY" auf Index 3 abgebildet, die unbekannte Kategorie "DESERT" auf einen der beiden OOV-Buckets (bei Index 5), und "INLAND" wurde zweimal auf Index 1 umgesetzt. Dann haben wir `tf.one_hot()` verwendet, um diese Indizes zu One-Hot-codieren. Beachten Sie, dass wir dieser Funktion die Gesamtanzahl an Indizes mitteilen müssen, was der Größe des Vokabulars plus der Anzahl an OOV-Buckets entspricht. Jetzt wissen Sie, wie Sie kategorische Merkmale als One-Hot-Vektoren mit TensorFlow codieren!

Wie zuvor wäre es nicht schwierig, diese ganze Logik in einer netten kleinen Klasse zusammenzufassen. Deren Methode `adapt()` würde ein Datenbeispiel übernehmen und daraus

die verschiedenen Kategorien extrahieren. Sie würde eine Lookup-Tabelle erstellen, um jede Kategorie auf ihren Index abzubilden (einschließlich unbekannter Kategorien mithilfe von OOV-Buckets). Dann würde deren Methode `call()` mit der Lookup-Tabelle die Eingabekategorien auf ihre Indizes abbilden. Es gibt übrigens weitere gute Nachrichten: Wenn Sie diese Zeilen lesen, wird Keras vermutlich eine Schicht namens `keras.layers.TextVectorization` enthalten, die genau das anbietet. Ihre Methode `adapt()` wird das Vokabular aus einem Datenbeispiel extrahieren, und ihre Methode `call()` wird jede Kategorie in ihren Index im Vokabular umwandeln. Sie könnten diese Schicht an den Anfang Ihres Modells stellen, gefolgt von einer Lambda-Schicht, die die Funktion `tf.one_hot()` anwendet, wenn Sie diese Indizes in One-Hot-Vektoren umwandeln wollen.

Das ist aber nicht unbedingt die beste Lösung. Die Größe jedes One-Hot-Vektors ist die Größe des Vokabulars plus die Anzahl an OOV-Buckets. Das ist in Ordnung, wenn es nur ein paar mögliche Kategorien gibt, aber wenn das Vokabular groß ist, ist es viel effizienter, sie stattdessen mit *Embeddings* zu codieren.

 Als Faustregel gilt: Gibt es weniger als zehn Kategorien, ist das One-Hot-Codieren im Allgemeinen der richtige Weg (bei Ihnen kann es aber anders aussehen!). Ist die Anzahl an Kategorien größer als 40 (was oft der Fall ist, wenn Sie Hash-Buckets verwenden), sind Embeddings zu bevorzugen. Zwischen 10 und 50 Kategorien sollten Sie eventuell mit beiden Varianten experimentieren, um zu sehen, welche für Ihren Anwendungsfall besser funktioniert.

Kategorische Merkmale mit Embeddings codieren

Ein Embedding ist ein trainierbarer dichter Vektor, der eine Kategorie repräsentiert. Standardmäßig werden Embeddings mit Zufallswerten initialisiert, daher könnte zum Beispiel die Kategorie "NEAR BAY" initial durch einen Zufallsvektor wie `[0.131, 0.890]` repräsentiert werden, während die Kategorie "NEAR OCEAN" vielleicht durch einen anderen Zufallsvektor wie `[0.631, 0.791]` umgesetzt wird. In diesem Beispiel nutzen wir 2-D-Embeddings, aber die Anzahl der Dimensionen ist ein Hyperparameter, an dem Sie drehen können. Da diese Embeddings trainierbar sind, werden sie sich während des Trainings nach und nach verbessern – und da sie recht ähnliche Kategorien repräsentieren, wird das Gradientenverfahren dafür sorgen, dass sie am Ende näher beieinanderliegen, während sie sich vom Embedding für die Kategorie "INLAND" eher entfernen (siehe [Abbildung 13-4](#)). Tatsächlich gilt: Je besser die Repräsentation ist, desto einfacher wird es für das neuronale Netz sein, genaue Vorhersagen zu treffen, daher tendiert das Training dazu, Embeddings zu nützlichen Repräsentationen der Kategorien zu machen. Das nennt sich Representation Learning (wir werden noch andere Formen des Representation Learning in [Kapitel 17](#) kennenlernen).

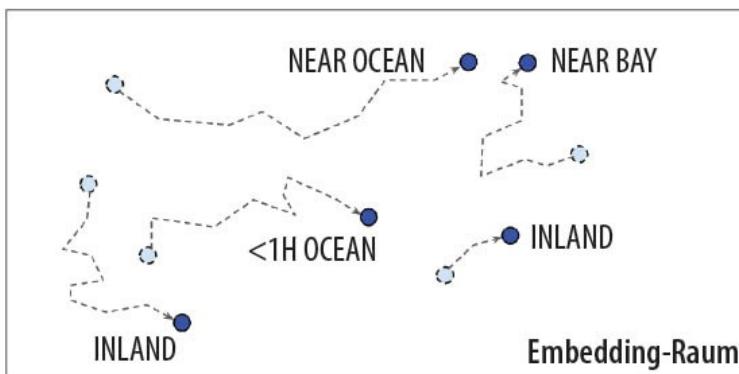


Abbildung 13-4: Embeddings werden im Verlauf des Trainings nach und nach besser.

Word Embeddings

Embeddings sind nicht nur allgemein nützliche Repräsentationen für die aktuelle Aufgabe, oft können diese gleichen Embeddings auch erfolgreich für andere Aufgaben wiederverwendet werden. Das häufigste Beispiel dafür sind *Word Embeddings* (also Embeddings einzelner Wörter): Arbeiten Sie an einer Aufgabe zur Verarbeitung natürlicher Sprache, ist es oft besser, vortrainierte Word Embeddings zu verwenden, als selbst welche zu trainieren.

Die Idee, Vektoren zum Darstellen von Wörtern zu verwenden, geht zurück bis in die 1960er-Jahre, und es wurden viele ausgefeilte Techniken zum Erzeugen nützlicher Vektoren eingesetzt, einschließlich neuronaler Netze. Aber im Jahr 2013 gewann das Ganze deutlich an Fahrt, als Tomáš Mikolov und andere Forscher bei Google einen Artikel (<https://homl.info/word2vec>)⁸ veröffentlichten, in dem sie eine effiziente Technik zum Lernen von Word Embeddings mithilfe neuronaler Netze beschreiben, die deutlich besser funktioniert als frühere Versuche. Das erlaubte ihnen, Embeddings für einen sehr großen Textkorpus zu erlernen – sie trainierten ein neuronales Netz darauf, die Wörter in der Nähe gegebener Wörter vorherzusagen, und erreichten erstaunliche Word Embeddings. So lagen beispielsweise Synonyme sehr nahe beieinander, und semantisch zusammengehörige Wörter wie Frankreich, Spanien und Italien kamen ebenfalls zu einem Cluster zusammen.

Es geht aber nicht nur um Nähe: Word Embeddings wurden auch entlang aussagekräftiger Achsen im Embedding-Raum organisiert. Hier ein berühmtes Beispiel: Berechnen Sie König – Mann + Frau (addieren bzw. subtrahieren Sie also die Embedding-Vektoren dieser Wörter), liegt das Ergebnis sehr nahe beim Embedding für das Wort Königin (siehe Abbildung 13-5). Oder anders gesagt: Die Word Embeddings codieren das Konzept von Geschlechtern! Genauso können Sie Madrid – Spanien + Frankreich berechnen, und das Ergebnis liegt nahe bei Paris, was zu zeigen scheint, dass die Idee einer Hauptstadt ebenfalls in den Embeddings codiert ist.

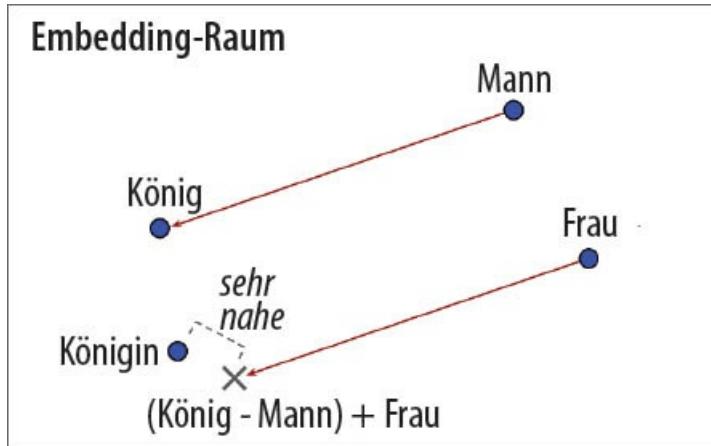


Abbildung 13-5: Word Embeddings ähnlicher Wörter tendieren dazu, nahe beieinanderzuliegen, und manche Achsen scheinen bedeutungsvolle Konzepte zu codieren.

Leider spiegeln sich in Word Embeddings manchmal auch unsere schlimmsten Vorurteile wider. So haben sie beispielsweise gelernt, dass Mann zu König wie Frau zu Königin steht, aber sie schienen auch gelernt zu haben, dass Mann zu Doktor wie Frau zu Krankenschwester steht – was für ein sexistisches Vorurteil! Fairerweise sei gesagt, dass gerade dieses Beispiel vermutlich übertrieben ist, wie in einem Artikel (<https://homl.info/fairembeds>) aus dem Jahr 2019 von Malvina Nissim et al. erwähnt wurde.⁹ Ungeachtet dessen ist das Sicherstellen von Fairness in Deep-Learning-Algorithmen ein wichtiges und aktives Forschungsgebiet.

Schauen wir uns an, wie wir Embeddings manuell implementieren können, um zu verstehen, wie sie funktionieren (danach werden wir einfach eine Keras-Schicht einsetzen). Zuerst müssen wir eine *Embedding-Matrix* mit den Embeddings jeder Kategorie erstellen und sie mit Zufallswerten initialisieren. Diese Matrix hat eine Zeile pro Kategorie und OOV-Bucket und eine Spalte pro Embedding-Dimension:

```
embedding_dim = 2

embed_init = tf.random.uniform([len(vocab) + num_oov_buckets, embedding_dim])

embedding_matrix = tf.Variable(embed_init)
```

In diesem Beispiel nutzen wir 2-D-Embeddings, aber als Faustregel gilt, dass Embeddings meist 10 bis 300 Dimensionen haben – abhängig von der Aufgabe und der Größe des Vokabulars (Sie werden diesen Hyperparameter anpassen müssen).

Diese Embedding-Matrix ist eine zufällig gefüllte 6×2 -Matrix, die in einer Variablen abgelegt wurde (damit sie während des Trainings durch das Gradientenverfahren angepasst werden kann):

```
>>> embedding_matrix
```

```
<tf.Variable 'Variable:0' shape=(6, 2) dtype=float32, numpy=
array([[0.6645621 , 0.44100678],
       [0.3528825 , 0.46448255],
       [0.03366041, 0.68467236],
       [0.74011743, 0.8724445 ],
       [0.22632635, 0.22319686],
       [0.3103881 , 0.7223358 ]], dtype=float32)>
```

Jetzt codieren wir den gleichen Batch mit kategorischen Merkmalen wie zuvor, dieses Mal aber über diese Embeddings:

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])

>>> cat_indices = table.lookup(categories)

>>> cat_indices

<tf.Tensor: id=741, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>

>>> tf.nn.embedding_lookup(embedding_matrix, cat_indices)

<tf.Tensor: id=864, shape=(4, 2), dtype=float32, numpy=
array([[0.74011743, 0.8724445 ],
       [0.3103881 , 0.7223358 ],
       [0.3528825 , 0.464448255],
       [0.3528825 , 0.464448255]], dtype=float32)>
```

Die Funktion `tf.nn.embedding_lookup()` holt sich die Zeile in der Embedding-Matrix mit dem angegebenen Index – das ist alles, was sie tut. So besagt beispielsweise die Lookup-Tabelle, dass sich die Kategorie "INLAND" an Indexposition 1 befindet, daher gibt die Funktion `tf.nn.embedding_lookup()` das Embedding aus Zeile 1 (zweimal) wieder: [0.3528825, 0.46448255].

Keras stellt eine Schicht `keras.layers.Embedding` zur Verfügung, die sich um die (standardmäßig trainierbare) Embedding-Matrix kümmert – wird die Schicht erstellt, initialisiert sie die Embedding-Matrix mit Zufallswerten, und wenn sie mit Kategorie-Indizes aufgerufen wird, gibt sie die Zeilen aus der Embedding-Matrix an diesen Indexwerten zurück:

```

...
>>> embedding(cat_indices)

<tf.Tensor: id=814, shape=(4, 2), dtype=float32, numpy=
array([[ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358],
       [-0.01471175, -0.00355174],
       [-0.01471175, -0.00355174]], dtype=float32)>

```

Wenn wir alles zusammenfügen, können wir jetzt ein Keras-Modell erstellen, das kategorische Merkmale verarbeiten kann (zusammen mit normalen numerischen Merkmalen) und dazu in der Lage ist, ein Embedding für jede Kategorie (und jeden OOV-Bucket) zu trainieren:

```

regular_inputs = keras.layers.Input(shape=[8])

categories = keras.layers.Input(shape=[], dtype=tf.string)

cat_indices = keras.layers.Lambda(lambda cats: table.lookup(cats))(categories)

cat_embed = keras.layers.Embedding(input_dim=6, output_dim=2)(cat_indices)

encoded_inputs = keras.layers.concatenate([regular_inputs, cat_embed])

outputs = keras.layers.Dense(1)(encoded_inputs)

model = keras.models.Model(inputs=[regular_inputs, categories],
                           outputs=[outputs])

```

Dieses Modell erwartet zwei Eingaben: eine reguläre Eingabe mit den acht numerischen Merkmalen pro Instanz und eine kategorische Eingabe (mit einem kategorischen Merkmal pro Instanz). Es verwendet eine Lambda-Schicht, um den Index jeder Kategorie nachzuschlagen, dann liest es die Embeddings für diese Indizes aus. Als Nächstes hängt es Embeddings und normale Eingabewerte aneinander, um die codierten Eingaben zu erhalten, die dann an ein neuronales Netz übergeben werden können. Wir könnten hier ein beliebiges neuronales Netz einfügen, aber wir haben nur eine Dense-Ausgabeschicht hinzugefügt und dann das Keras-Modell erstellt.

Ist die Schicht `keras.layers.TextVectorization` verfügbar, können Sie deren Methode `adapt()` aufrufen, damit sie das Vokabular aus einem Datenbeispiel extrahiert (sie wird sich auch darum kümmern, für Sie die Lookup-Tabelle anzulegen). Dann können Sie sie zu Ihrem Modell hinzufügen, damit sie den Index-Lookup durchführt (und damit die Lambda-Schicht aus dem vorherigen Codebeispiel ersetzt).

One-Hot-Codierung nach einer Dense-Schicht (ohne Aktivierungsfunktion und Bias) entspricht

einer Embedding-Schicht. Aber die Embedding-Schicht benötigt viel weniger Berechnungen (die Performanceunterschiede werden deutlich, wenn die Embedding-Matrix wächst). Die Gewichtsmatrix der Dense-Schicht spielt die Rolle der Embedding-Matrix. So verhalten sich One-Hot-Vektoren der Größe 20 und eine Dense-Schicht mit 10 Einheiten wie eine Embedding-Schicht mit `input_dim=20` und `output_dim=10`. Es wäre Verschwendug, mehr Embedding-Dimensionen zu verwenden, als die folgende Schicht Einheiten besitzt.

Schauen wir uns die Vorverarbeitungsschichten von Keras nun etwas genauer an.

Vorverarbeitungsschichten von Keras

Das TensorFlow-Team arbeitet daran, einen Satz von Standard-Vorverarbeitungsschichten in Keras (<https://homl.info/preproc>) anzubieten. Wenn Sie diese Zeilen lesen, werden sie vermutlich verfügbar sein. Die API kann sich noch ein bisschen ändern, daher sollten Sie im Notebook zu diesem Kapitel nachschauen, wenn sich Dinge unerwartet verhalten. Diese neue API wird wohl die bestehende Feature Columns API ablösen, die sich nur aufwendiger und weniger intuitiv nutzen lässt (wollen Sie trotzdem mehr über die Feature Columns API erfahren, schauen Sie in das Notebook zu diesem Kapitel).

Wir haben schon zwei dieser Schichten erwähnt: Die Schicht `keras.layers.Normalization` führt eine Standardisierung durch (sie entspricht der Standardization-Schicht, die wir weiter oben definiert hatten), die Schicht `TextVectorization` wird jedes Wort in den Eingaben in seinen Index im Vokabular abbilden können. In beiden Fällen erstellen Sie die Schicht, rufen deren Methode `adapt()` mit einem Datenbeispiel auf und verwenden sie dann ganz normal in Ihrem Modell. Die anderen Vorverarbeitungsschichten werden dem gleichen Muster folgen.

Die API wird auch eine Schicht `keras.layers.Discretization` anbieten, die kontinuierliche Daten in verschiedene Körbe aufteilt und jeden davon als One-Hot-Vektor codiert. So könnten Sie sie beispielsweise verwenden, um Preise in drei Kategorien zu diskretisieren (niedrig, mittel, hoch), die dann als `[1, 0, 0]`, `[0, 1, 0]` bzw. `[0, 0, 1]` codiert würden. Natürlich geht damit viel Information verloren, aber in manchen Fällen kann das dem Modell helfen, Muster zu erkennen, die ansonsten beim reinen Blick auf die kontinuierlichen Werte nicht offensichtlich wären.

Die Schicht `Discretization` wird nicht differenzierbar sein, und sie sollte nur am Anfang Ihres Modells Verwendung finden. Tatsächlich werden die Vorverarbeitungsschichten des Modells während des Trainings eingefroren, sodass ihre Parameter nicht vom Gradientenverfahren beeinflusst werden – daher müssen sie auch nicht differenzierbar sein. Das heißt ebenfalls, dass Sie eine Embedding-Schicht nicht direkt in einer eigenen Vorverarbeitungsschicht einsetzen sollten, wenn diese trainierbar sein soll: Stattdessen sollte sie getrennt Ihrem Modell hinzugefügt werden – so wie wir es im vorherigen Codebeispiel getan haben.

Es wird auch möglich sein, mehrere Vorverarbeitungsschichten mit der Klasse `PreprocessingStage` zu verketten. So erzeugt beispielsweise der folgende Code eine Vorverarbeitungspipeline, die erst die Eingaben normalisiert und sie dann diskretisiert (das

erinnert Sie vielleicht an Pipelines bei Scikit-Learn). Nach dem Anpassen dieser Pipeline an ein Datenbeispiel können Sie sie wie eine normale Schicht in Ihren Modellen verwenden (aber auch wieder nur am Anfang des Modells, da sie eine nicht differenzierbare Vorverarbeitungsschicht enthält):

```
normalization = keras.layers.Normalization()  
discretization = keras.layers.Discretization([...])  
pipeline = keras.layers.PreprocessingStage([normalization, discretization])  
pipeline.adapt(data_sample)
```

Die Schicht `TextVectorization` wird zudem eine Option haben, um Worthäufigkeitsvektoren statt Wortindizes auszugeben. Enthält das Vokabular beispielsweise drei Wörter – sagen wir `["and", "basketball", "more"]` –, wird der Text `"more and more"` auf den Vektor `[1, 0, 2]` abgebildet: `"and"` erscheint einmal, das Wort `"bas ketball"` gar nicht und das Wort `"more"` zweimal. Diese Textrepräsentation wird *Bag of Words* genannt, da die Reihenfolge der Wörter komplett verloren geht. Häufige Wörter wie `"and"` werden in den meisten Texten einen großen Wert haben, auch wenn es meist die am wenigsten interessanten sind (zum Beispiel ist im Text `"more and more basketball"` das Wort `"basketball"` ganz klar das wichtigste und dabei kein sehr häufig vorkommendes). Daher sollte die Worthäufigkeit so normalisiert werden, dass die Wichtigkeit häufiger Wörter reduziert wird. Meist dividiert man dazu jede Worthäufigkeit durch den Logarithmus der Gesamtzahl an Trainingsinstanzen, in denen das Wort vorkommt. Diese Technik nennt sich *Term-Frequency × Inverse-Document-Frequency* (TF-IDF). Nehmen wir beispielsweise an, die Wörter `"and"`, `"basketball"` und `"more"` tauchen in 200, 10 und 100 Textinstanzen im Trainingsdatensatz auf: In diesem Fall ist der Vektor `[1/log(200), 0/log(10), 2/log(100)]`, was in etwa `[0.19, 0., 0.43]` entspricht. Die Schicht `TextVectorization` wird (vermutlich) eine Option zum Ausführen von TF-IDF anbieten.



Reichen Ihnen die Standard-Vorverarbeitungsschichten nicht für Ihre Aufgabe aus, werden Sie immer noch die Möglichkeit haben, Ihre eigenen Vorverarbeitungsschichten zu erstellen, so wie wir es weiter oben mit der Klasse `Standardization` gemacht haben. Erzeugen Sie eine Subklasse von `keras.layers.Preprocessing` Layer mit einer Methode `adapt()`, die ein Argument `data_sample` und optional ein Argument `reset_state` übernehmen sollte: Wenn Letzteres `True` ist, sollte die Methode `adapt()` jeglichen bestehenden Status zurücksetzen, bevor sie den neuen Status berechnet; ist es `False`, sollte sie versuchen, den bestehenden Status zu aktualisieren.

Wie Sie sehen, werden diese Vorverarbeitungsschichten von Keras das Leben viel leichter machen! Aber unabhängig davon, ob Sie sich dazu entscheiden, Ihre eigenen Vorverarbeitungsschichten zu schreiben oder die von Keras einzusetzen (oder sogar die der Feature Columns API), geschieht die gesamte Vorverarbeitung »unterwegs«. Während des Trainings kann es aber sinnvoll sein, die Vorverarbeitung vorab durchzuführen. Schauen wir, warum wir das tun sollten und wie wir das erreichen können.

TF Transform

Wenn die Vorverarbeitung rechenintensiv ist, kann es das Training deutlich beschleunigen, wenn Sie sie vorher durchführen – die Daten werden *vor* dem Training einmal pro Instanz vorverarbeitet, statt einmal pro Instanz und Epoche *während* des Trainings. Wie zuvor erwähnt, können Sie die Methode `cache()` des Datasets nutzen, wenn es vollständig in den Speicher passt. Ist es aber zu groß, helfen Tools wie Apache Beam oder Spark. Mit ihnen können Sie Datenverarbeitungspipelines effizient über große Datenmengen laufen lassen – selbst verteilt über mehrere Server –, sodass Sie sie einsetzen können, um alle Trainingsdaten vor dem Training vorzubereiten.

Das funktioniert großartig und kann das Training tatsächlich beschleunigen, aber es gibt ein Problem: Stellen Sie sich vor, Sie wollten das trainierte Modell in einer mobilen App deployen. In diesem Fall müssen Sie in Ihrer App Code schreiben, der sich um das Vorverarbeiten der Daten kümmert, bevor sie in das Modell gefüttert werden. Und stellen Sie sich außerdem vor, Sie wollten das Modell nach *TensorFlow.js* deployen, sodass es in einem Webbrower laufen kann. Auch hier werden Sie entsprechenden Code schreiben müssen. Das kann ein Wartungsalbtraum werden: Wann immer Sie die Vorverarbeitungslogik ändern wollen, müssen Sie Ihren Code in Apache Beam und in Ihrer mobilen App sowie Ihren JavaScript-Code anpassen. Das ist nicht nur zeitaufwendig, sondern auch fehleranfällig – irgendwann schleichen sich kleine Unterschiede zwischen den Vorverarbeitungsoperationen vor dem Training und denen in Ihrer App oder im Browser ein. Dieser *Training/Serving Skew* wird zu Fehlern oder einer schlechteren Leistung führen.

Eine Verbesserung wäre, das trainierte Modell zu nehmen (trainiert mit Daten, die durch Ihren Apache-Beam- oder Spark-Code vorverarbeitet wurden) und vor dem Deployen in Ihre App oder in den Brower zusätzliche Vorverarbeitungsschichten hinzuzufügen, die sich im Einsatz um das Vorbereiten der Daten kümmern. Das ist auf jeden Fall besser, da Sie jetzt nicht zwei Versionen Ihres Vorverarbeitungscodes haben.

Aber wäre es nicht toll, wenn Sie Ihre Vorverarbeitungsoperationen nur einmal definieren müssten? Dafür wurde TF Transform geschaffen. Es ist Teil von TensorFlow Extended (TFX) (<https://tensorflow.org/tfx>), einer End-to-End-Plattform, um TensorFlow-Modelle in die Produktion zu bringen. Um eine TFX-Komponente wie TF Transform nutzen zu können, müssen Sie es zunächst installieren – es kommt bei TensorFlow nicht automatisch mit. Dann definieren Sie Ihre Vorverarbeitungsfunktion nur einmal (in Python), indem Sie Funktionen von TF Transform zum Skalieren, Aufteilen in Buckets und so weiter einsetzen. Sie können bei Bedarf auch andere TensorFlow-Operationen verwenden. So sähe unsere Vorverarbeitungsfunktion aus, wenn wir nur zwei Merkmale hätten:

```
import tensorflow_transform as tft

def preprocess(inputs): # inputs = Batch mit Eingangsmerkmalen
    median_age = inputs["housing_median_age"]
```

```

ocean_proximity = inputs["ocean_proximity"]

standardized_age = tft.scale_to_z_score(median_age)

ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)

return {

    "standardized_median_age": standardized_age,

    "ocean_proximity_id": ocean_proximity_id

}

```

Als Nächstes können Sie mit TF Transform die Funktion `preprocess()` mit Apache Beam auf den gesamten Trainingsdatensatz anwenden (es stellt eine Klasse `AnalyzeAndTransformDataset` bereit, die Sie dafür in Ihrer Apache-Beam-Pipeline einsetzen können). Dabei werden auch alle notwendigen Statistiken über den gesamten Trainingsdatensatz ermittelt: In diesem Beispiel sind das Mittelwert und Standardabweichung des Merkmals `housing_median_age` und das Vokabular für das Merkmal `ocean_proximity`. Die Komponenten, die diese Statistiken berechnen, werden als *Analyzer* bezeichnet.

Wichtig ist darüber hinaus, dass TF Transform auch eine äquivalente TensorFlow-Funktion generiert, die Sie zum Deployen an das Modell andocken können. Diese TF Function enthält ein paar Konstanten, die mit all den von Apache Beam berechneten Statistiken korrespondieren (Mittelwert, Standardabweichung und Vokabular).

Mit der Data-API, TFRecords, den Vorverarbeitungsschichten von Keras und TF Transform können Sie hochskalierbare Eingabepipelines zum Training bauen und von einer schnellen und portablen Datenvorverarbeitung im Produktivumfeld profitieren.

Aber was ist, wenn Sie nur einen Standarddatensatz einsetzen wollen? Nun, in diesem Fall ist alles viel einfacher: Sie verwenden TFDS!

Das TensorFlow-Datasets-(TFDS-)Projekt

Das TensorFlow- Datasets (<https://tensorflow.org/datasets>)-Projekt macht es sehr leicht, häufig genutzte Datensätze herunterzuladen – von kleinen wie MNIST oder Fashion MNIST bis zu riesigen Datensätzen wie ImageNet (dafür werden Sie einiges an Plattenplatz brauchen!). Es gibt Bilddatensätze, Textdatensätze (einschließlich Übersetzungsdatensätzen) sowie Audio- und Videodatensätze. Die vollständige Liste finden Sie unter <https://homl.info/tfds> zusammen mit einer Beschreibung für jeden Datensatz.

TFDS wird nicht automatisch mit TensorFlow mitgeliefert, daher müssen Sie die Bibliothek `tensorflow-datasets` installieren (zum Beispiel mit pip). Dann rufen Sie die Funktion `tfds.load()` auf, was die gewünschten Daten herunterlädt (sofern sie nicht schon einmal geladen wurden) und als Dictionary mit Datasets zurückgibt (meist eines zum Training und eines zum Testen, aber das hängt vom gewählten Datensatz ab). Laden wir beispielsweise MNIST herunter:

```
import tensorflow_datasets as tfds

dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["test"]
```

Sie können dann beliebige Transformationen darauf anwenden (meist durchmischen, in Batches unterteilen und prefetchen) und können dann Ihr Modell trainieren. Hier ein einfaches Beispiel:

```
mnist_train = mnist_train.shuffle(10000).batch(32).prefetch(1)

for item in mnist_train:

    images = item["image"]

    labels = item["label"]

    [...]
```



Die Funktion `load()` kann die heruntergeladenen Dateien durchmischen, setzen Sie einfach `shuffle_files=True`. Das reicht aber eventuell nicht aus, daher ist es besser, die Trainingsdaten noch mehr zu durchmischen.

Beachten Sie, dass jedes Element im Dataset ein Dictionary mit den Merkmalen und den Labels ist. Aber Keras erwartet, dass es sich bei jedem Element um ein Tupel mit zwei Elementen handelt (wieder die Merkmale und die Labels). Sie können das Dataset mit der Methode `map()` umwandeln, zum Beispiel so:

```
mnist_train = mnist_train.shuffle(10000).batch(32)

mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))

mnist_train = mnist_train.prefetch(1)
```

Aber es ist einfacher, das die `load()`-Funktion für Sie machen zu lassen, indem Sie `as_supervised=True` setzen (offensichtlich funktioniert das nur mit gelabelten Datensätzen). Sie können auch die Batchgröße angeben. Dann können Sie das Dataset direkt an Ihr `tf.keras`-Modell übergeben:

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)

mnist_train = dataset["train"].prefetch(1)

model = keras.models.Sequential([...])

model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")

model.fit(mnist_train, epochs=5)
```

Das war ein ziemlich technisches Kapitel, und Sie haben vielleicht das Gefühl, dass Sie sich ein bisschen zu sehr von der abstrakten Schönheit neuronaler Netze entfernt haben, aber Tatsache ist, dass zum Deep Learning oftmals große Mengen an Daten gehören. Zu wissen, wie Sie diese effizient laden, parsen und vorverarbeiten, ist eine wichtige Fähigkeit. Im nächsten Kapitel werden wir uns die Architektur von Convolutional Neural Networks zur Bildverarbeitung und für viele andere Anwendungsfälle anschauen.

Übungen

1. Warum sollten Sie die Data-API verwenden?
2. Was sind die Vorteile des Aufteilens eines großen Datensatzes in viele Dateien?
3. Wie können Sie während des Trainings erkennen, dass Ihre Eingabepipeline der Flaschenhals ist? Was können Sie dagegen tun?
4. Können Sie Binärdaten in eine TFRecord-Datei sichern, oder sind nur serialisierte Protocol Buffer möglich?
5. Warum sollten Sie sich die Mühe machen, all Ihre Daten in das Example-Protobuf-Format zu konvertieren? Warum sollten Sie nicht Ihre eigene Protobuf-Definition verwenden?
6. Wann würden Sie beim Einsatz von TFRecords die Komprimierung aktivieren? Warum sollten Sie das nicht systematisch tun?
7. Daten können direkt beim Schreiben der Datendateien, innerhalb der tf.data-Pipeline, in Vorverarbeitungsschichten innerhalb Ihres Modells oder mithilfe von TF Transform vorverarbeitet werden. Können Sie ein paar Vor- und Nachteile jeder Option aufzählen?
8. Benennen Sie ein paar verbreitete Techniken, die Sie zum Codieren kategorischer Merkmale einsetzen können. Wie ist es mit Texten?
9. Laden Sie den Fashion-MNIST-Datensatz (den wir in [Kapitel 10](#) vorgestellt haben), teilen Sie ihn in einen Trainings-, einen Validierungs- und einen Testdatensatz auf, durchmischen Sie den Trainingsdatensatz und sichern Sie jedes Dataset in mehreren TFRecord-Dateien. Jeder Datensatz sollte ein serialisierter Example-Protobuf mit zwei Merkmalen sein: dem serialisierten Bild (verwenden Sie `tf.io.serialize_tensor()`, um jedes Bild zu serialisieren) und dem Label.¹⁰ Dann verwenden Sie tf.data, um ein effizientes Dataset für jeden Datensatz zu erstellen. Und schließlich nutzen Sie ein Keras-Modell, um diese Datasets zu trainieren, wobei Sie eine Vorverarbeitungsschicht zum Standardisieren jedes Eingabemerkals verwenden. Versuchen Sie, die Eingabepipeline so effizient wie möglich zu gestalten, und nutzen Sie TensorBoard, um die Profiling-Daten zu visualisieren.
10. In dieser Übung laden Sie einen Datensatz herunter, teilen ihn auf, erstellen ein `tf.data.Dataset`, um ihn zu laden und effizient vorzuverarbeiten, und dann bauen und trainieren Sie ein binäres Klassifikationsmodell mit einer Embedding-Schicht:
 - a. Laden Sie das Large Movie Review Dataset (<https://homl.info/imdb>) mit 50.000 Filmkritiken aus der Internet Movie Database (<https://imdb.com/>). Die Daten sind in zwei Verzeichnissen *train* und *test* organisiert, wobei jedes ein Unterverzeichnis *pos*

mit 12.500 positiven Kritiken und ein Unterverzeichnis *neg* mit 12.500 negativen Kritiken enthält. Jede Kritik ist in einer eigenen Textdatei abgelegt. Es gibt andere Dateien und Ordner (einschließlich vorverarbeiteter Bag of Words), aber die werden wir in dieser Übung ignorieren.

- b. Teilen Sie den Testdatensatz in einen Validierungsdatensatz (15.000) und einen Testdatensatz (10.000) auf.
- c. Verwenden Sie tf.data, um ein effizientes Dataset für jeden Datensatz zu erstellen.
- d. Erzeugen Sie ein binäres Klassifikationsmodell mit einer TextVectorization-Schicht zum Vorverarbeiten jeder Kritik. Ist die Schicht TextVectorization noch nicht vorhanden (oder lieben Sie die Herausforderung), versuchen Sie, Ihre eigene Vorverarbeitungsschicht zu erstellen: Sie können die Funktionen im Paket `tf.strings` nutzen, zum Beispiel `lower()`, um alles in Kleinbuchstaben umzuwandeln, oder `regex_replace()`, um Satzzeichen durch Leerzeichen zu ersetzen, oder `split()` zum Aufteilen der Wörter an den Leerzeichen. Sie sollten eine Lookup-Tabelle zur Ausgabe der Wortindizes verwenden, die in der Methode `adapt()` vorzubereiten ist.
- e. Fügen Sie eine Embedding-Schicht hinzu und berechnen Sie das mittlere Embedding für jede Kritik, multipliziert mit der Wurzel der Anzahl der Wörter (siehe [Kapitel 16](#)). Dieses umskalierte mittlere Embedding kann dann an den Rest des Modells weitergegeben werden.
- f. Trainieren Sie das Modell und finden Sie heraus, welche Genauigkeit Sie erhalten. Versuchen Sie, Ihre Pipelines zu optimieren, um das Training so schnell wie möglich zu machen.
- g. Nutzen Sie TFDS, um denselben Datensatz viel einfacher zu laden: `tfds.load("imdb_reviews")`.

Lösungen zu diesen Übungen finden Sie in [Anhang A](#).

Deep Computer Vision mit 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 darin, dass die Wahrnehmung hauptsächlich außerhalb unseres Bewusstseins, also in den auf Sicht, Gehör oder andere Sinne spezialisierten Bereichen unseres Gehirns stattfindet. Bis sensorische Informationen unser Bewusstsein erreichen, sind 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 und Keras implementieren lassen. Anschließend präsentieren wir einige der besten CNN-Architekturen sowie andere Bilderkennungsaufgaben, unter anderem Objekterkennung (das Klassifizieren mehrerer Objekte in einem Bild und das Platzieren von Bounding Boxes um sie herum) und semantische Segmentierung (das Klassifizieren jedes Pixels in Bezug auf die Kategorie des Objekts, zu dem er gehört).

Der Aufbau des visuellen Cortex

David H. Hubel und Torsten Wiesel führten in den Jahren 1958 (<https://homl.info/71>)¹ und 1959 (<https://homl.info/72>)² eine Reihe Experimente an Katzen durch (und ein paar Jahre später an

Affen (<https://homl.info/73>)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 14-1 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 14-1 beachten Sie, dass jedes Neuron nur mit wenigen Neuronen der vorherigen Schicht verbunden ist). Diese mächtige Architektur ist in der Lage, alle möglichen komplexen Muster in einem beliebigen Teil des Gesichtsfelds zu erkennen.

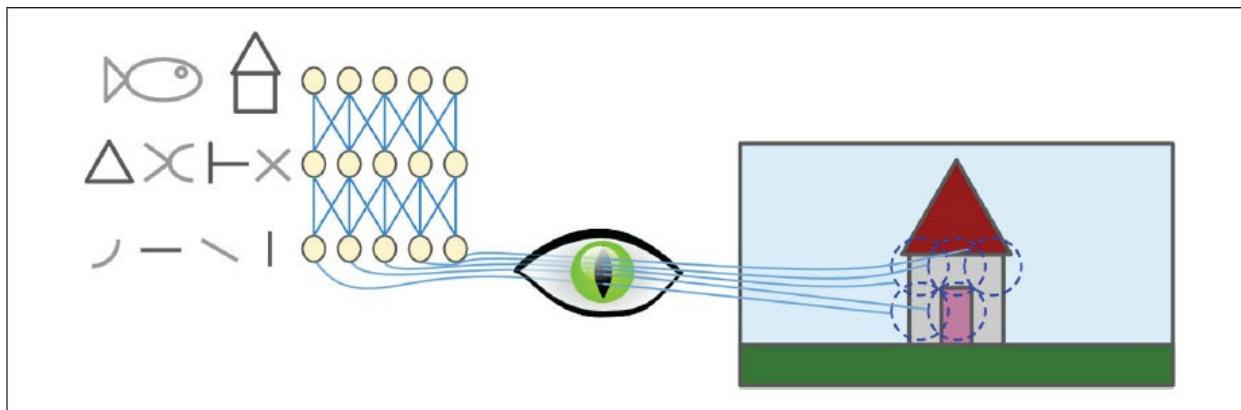


Abbildung 14-1: Biologische Neuronen im visuellen Cortex reagieren auf bestimmte Muster in kleinen Bereichen des visuellen Sichtfelds, die als Wahrnehmungsfelder bezeichnet werden; auf dem Weg des visuellen Signals durch die Hirnbereiche reagieren Neuronen auf immer komplexere Muster in größeren Wahrnehmungsfeldern.

Diese Untersuchungen des visuellen Cortex inspirierten das im Jahr 1980 vorgestellte Neocognitron (<https://homl.info/74>)⁴, das sich nach und nach zu dem entwickelte, was wir heute als *Convolutional Neural Networks* bezeichnen. Ein wichtiger Meilenstein war dabei ein Artikel (<https://homl.info/75>) aus dem Jahr 1998⁵ 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 10.000 Pixel, und wenn die erste Schicht nur 1.000 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 und die gemeinsame Nutzung von Gewichten.

Convolutional Layers

Der wichtigste Bestandteil eines CNN sind die *Convolutional Layers*: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 14-2). Im Gegenzug ist jedes Neuron im zweiten Convolutional Layer ausschließlich mit Neuronen innerhalb eines kleinen Rechtecks der ersten Schicht verbunden. Durch diese Architektur kann sich das Netzwerk 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.

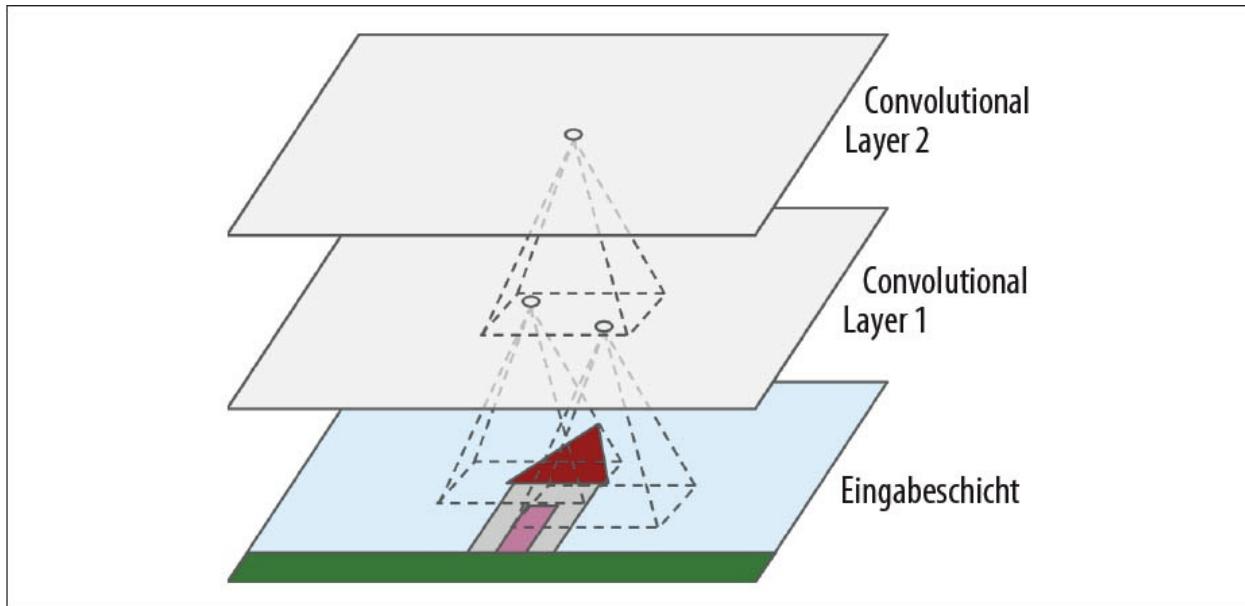


Abbildung 14-2: Schichten eines CNN mit rechteckigen lokalen Wahrnehmungsfeldern

- „ Alle bisher von uns betrachteten mehrschichtigen neuronalen Netze bestanden 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 vorherigen 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 14-3](#)). Damit eine Schicht die gleiche Höhe und Breite wie die vorherige hat, ist es üblich, Nullen um die Eingaben herum zu platzieren wie die Abbildung zeigt. Dies nennt man auch *Zero Padding*.

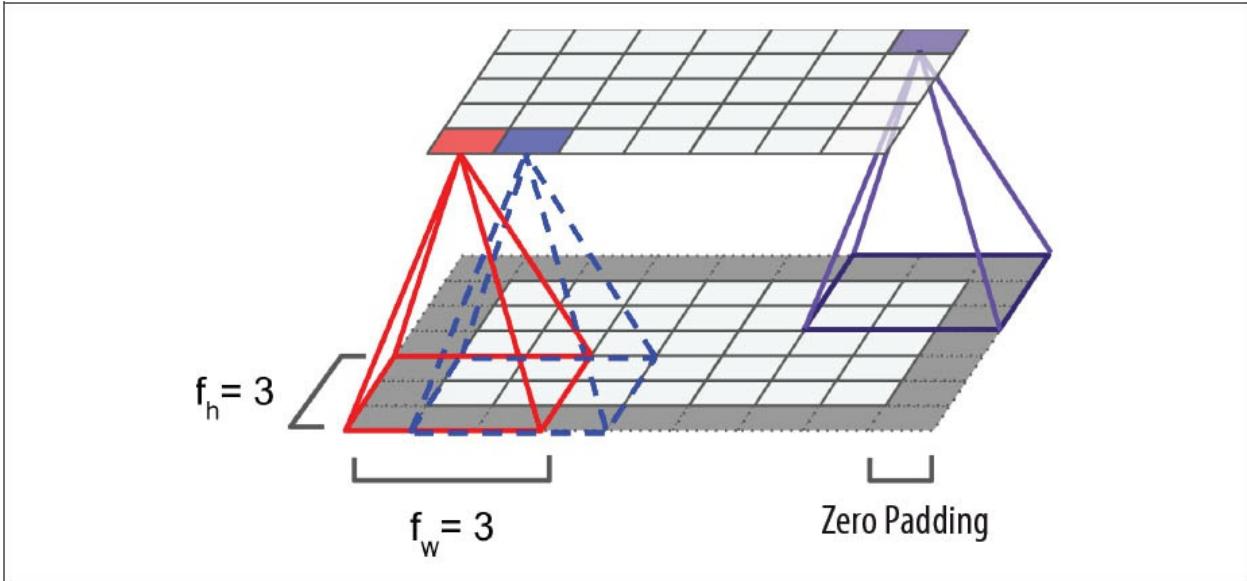


Abbildung 14-3: Verbindungen zwischen Schichten und Zero Padding

Es ist ebenfalls möglich, eine große Eingabeschicht mit einer viel kleineren Schicht zu verbinden, indem die Wahrnehmungsfelder wie in [Abbildung 14-4](#) in größeren Abständen gestaffelt werden. Das verringert die Rechenkomplexität des Modells deutlich. Der Abstand zwischen zwei aufeinanderfolgenden Wahrnehmungsfeldern wird als *Schrittweite* (engl. stride) bezeichnet. Im Diagramm 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 beide 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 vorherigen Schicht in den Zeilen $i \times s_h$ bis $i \times s_h + f_h - 1$ und den Spalten $j \times s_w + f_w - 1$ verbunden, wobei s_h und s_w die vertikalen und horizontalen Schrittweiten sind.

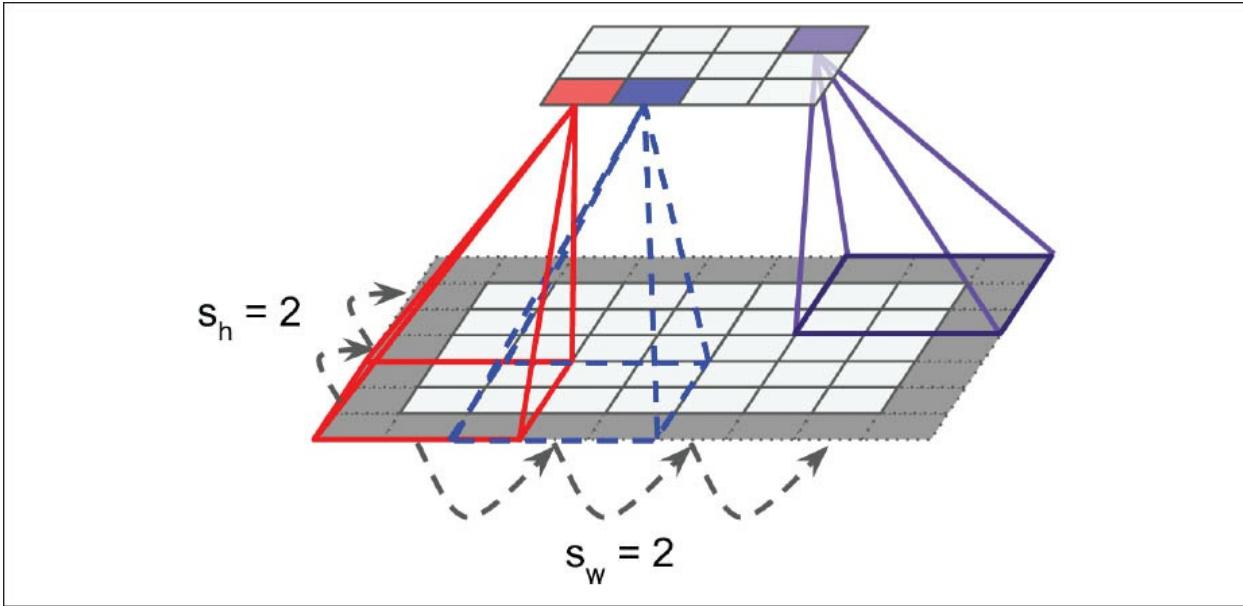


Abbildung 14-4: Dimensionsreduktion über eine Schrittweite von 2

Filter

Die Gewichte eines Neurons lassen sich als kleines Bild mit der Größe des Wahrnehmungsfelds darstellen. Das Beispiel in [Abbildung 14-5](#) 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 14-5](#) (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 den Filter am meisten aktivierenden Bildbereiche hervorhebt. Natürlich müssen Sie die Filter nicht manuell definieren: Stattdessen findet der Convolutional Layer beim Trainieren die für die Aufgabe nützlichsten Filter, und die Schichten darüber lernen, diese zu komplexeren Mustern zu kombinieren.

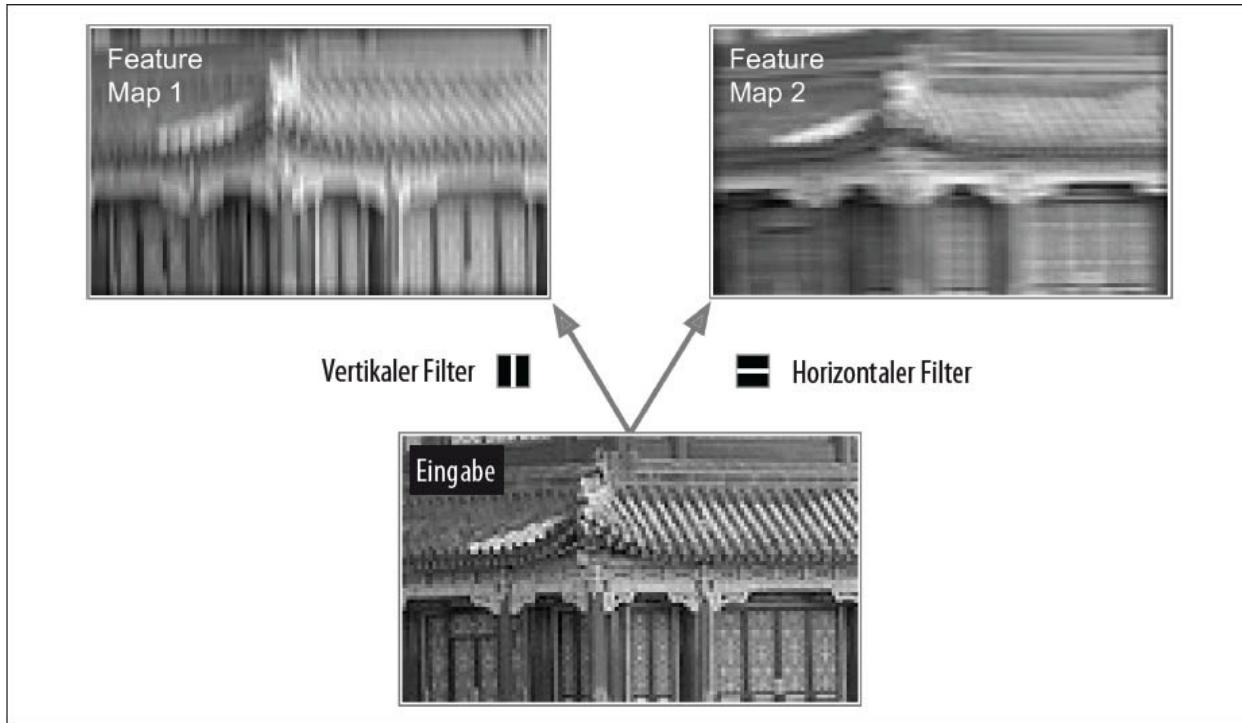


Abbildung 14-5: Anwenden von zwei unterschiedlichen Filtern, um zwei Feature Maps zu erhalten

Stapeln mehrerer Feature Maps

Bisher haben wir die Ausgabe jedes Convolutional Layer als 2-D-Schicht dargestellt, tatsächlich besteht ein solcher Layer aber aus mehreren Filtern (Sie können festlegen, wie viele), und er gibt eine Feature Map pro Filter aus, die sich besser in 3-D darstellen lassen (siehe [Abbildung 14-6](#)). Innerhalb einer Feature Map gibt es ein Neuron pro Pixel, und sämtliche Neuronen in einer gegebenen Feature Map teilen die gleichen Parameter (zum Beispiel Gewichte und Bias-Term). Neuronen in unterschiedlichen Feature Maps nutzen aber auch unterschiedliche Parameter. Das Wahrnehmungsfeld eines Neurons entspricht dem oben beschriebenen, es erstreckt sich aber über alle Feature Maps der vorherigen Schicht. Zusammengefasst, wendet ein Convolutional Layer gleichzeitig mehrere trainierbare 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 der Parameter im Modell erheblich. Kann ein CNN ein Muster an einer Stelle erkennen, kann es dieses auch an einer beliebigen anderen Stelle erkennen. 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).

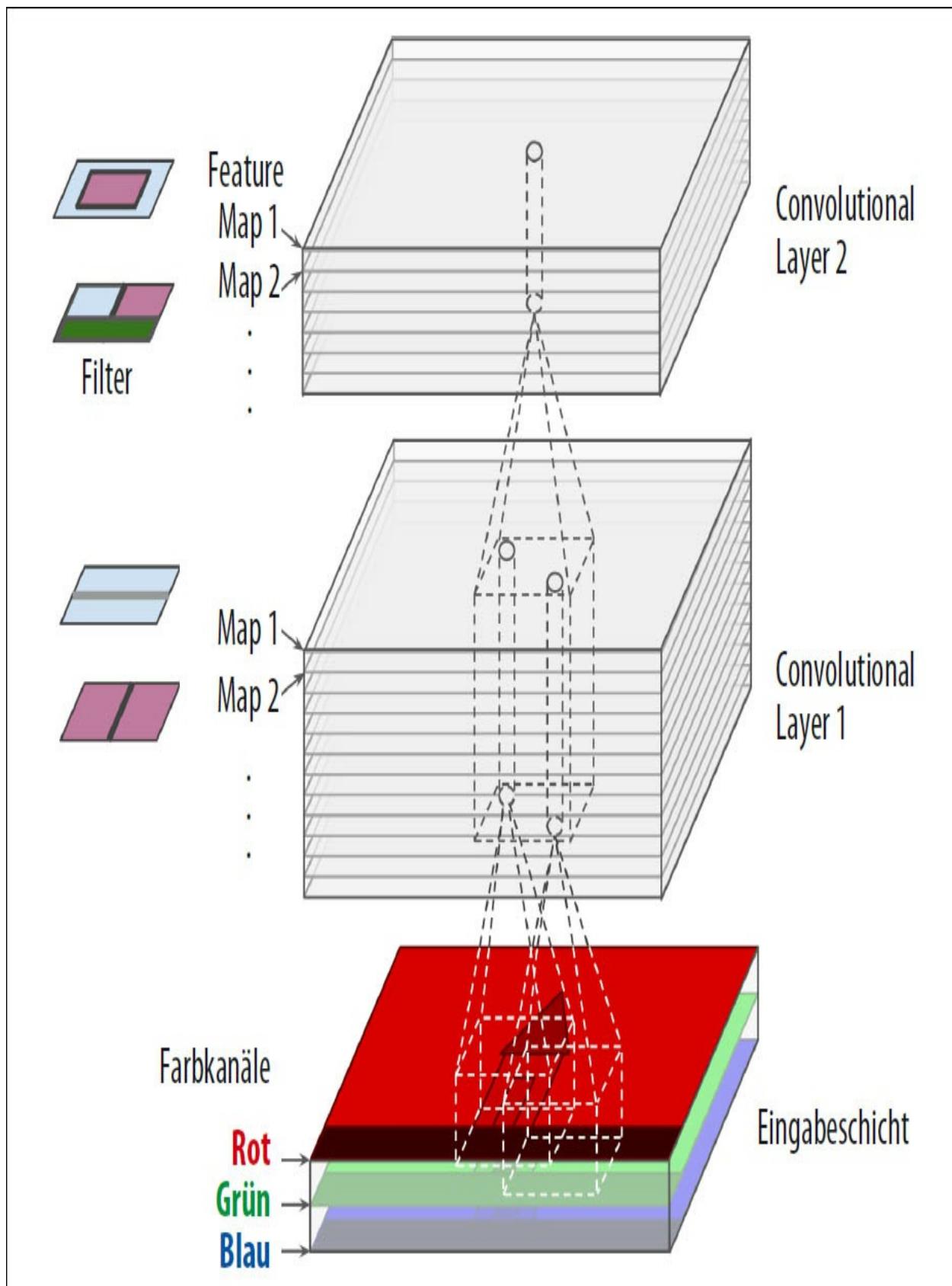


Abbildung 14-6: Convolutional Layer mit mehreren Feature Maps und Bilder mit drei Kanälen

Ein Neuron in Zeile i und Spalte j der Feature Map k im Convolutional Layer l ist mit den Ausgaben der Neuronen der vorherigen 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 vorherigen Schicht verbunden sind.

[Formel 14-1](#) 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 14-1: Berechnung der Ausgabe eines Neurons in einem Convolutional Layer

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_n'-1} x_{i',j',k'} \times w_{u,v,k',k} \quad \text{mit } \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$ ist die Ausgabe des Neurons in Zeile i und Spalte j in Feature Map k des Convolutional Layer (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 vorherigen 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 vorherige 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). Für diese Funktion muss Pillow installiert sein (zum Beispiel über `python3 -m install Pillow`). Dann erstellt er zwei Filter, wendet

diese auf beide Bilder an und zeigt schließlich eine der sich ergebenden Feature Maps als Bild an:

```
from sklearn.datasets import load_sample_image

# Laden der Beispielbilder

china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255

images = np.array([china, flower])
batch_size, height, width, channels = images.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

outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")

plt.imshow(outputs[0, :, :, 1], cmap="gray") # zeichne die 2. Feature Map
                                                # des 1. Bilds

plt.show()
```

Gehen wir diesen Code durch:

- Die Pixelintensität für jeden Farbkanal wird als Byte von 0 bis 255 dargestellt, daher skalieren wir diese Merkmale einfach mit einer Division durch 255, um Gleitkommazahlen im Bereich von 0 bis 1 zu erhalten.
- Dann erstellen wir zwei 7×7 -Filter (einen mit einer vertikalen, den anderen mit einer horizontalen weißen Linie in der Mitte).
- Wir wenden sie mit der Funktion `tf.nn.conv2d()` aus der Low-Level-Deep-Learning-API von TensorFlow auf beide Bilder an, wobei wir Zero Padding (`padding=SAME`) und eine Schrittweite von 1 wählen.
- Schließlich geben wir eine der entstandenen Feature Maps aus (entsprechend dem Bild oben rechts in [Abbildung 14-5](#)).

Die Zeile `tf.nn.conv2d()` verdient etwas mehr Erläuterung:

- `images` 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 gleich 1, es könnte sich aber auch um ein 1-D-Array mit vier Elementen handeln, 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, eine Schrittweite für Batches (um einige Datenpunkte zu überspringen) oder Kanäle (um einige der Feature Maps oder Kanäle der vorherigen Schicht zu überspringen) anzugeben.
- `padding` muss auf "same" oder "valid" gesetzt werden:
 - 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 und aufgerundet. Ist die Eingabeanzahl beispielsweise 13 und die Schrittweite 5 (siehe Abbildung 14-7), ist die Anzahl der Ausgabeneuronen 3 ($13 / 5 = 2,6$, aufgerundet auf 3). Die Nullen werden so gleichmäßig wie möglich um die Eingabedaten platziert. Ist `strides=1`, hat die Ausgabe der Schicht die gleichen räumlichen Dimensionen (Breite und Höhe) wie die Eingabe, daher der Name *same*.
 - 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 14-7 zeigt (der Einfachheit halber ist hier nur die horizontale Dimension dargestellt, das gleiche Prinzip findet natürlich auch bei der vertikalen Dimension Anwendung). Das bedeutet, dass das Wahrnehmungsfeld jedes Neurons strikt innerhalb der gültigen Positionen innerhalb der Eingaben liegt (es geht nicht über die Grenzen hinaus) – daher der Name *valid*.

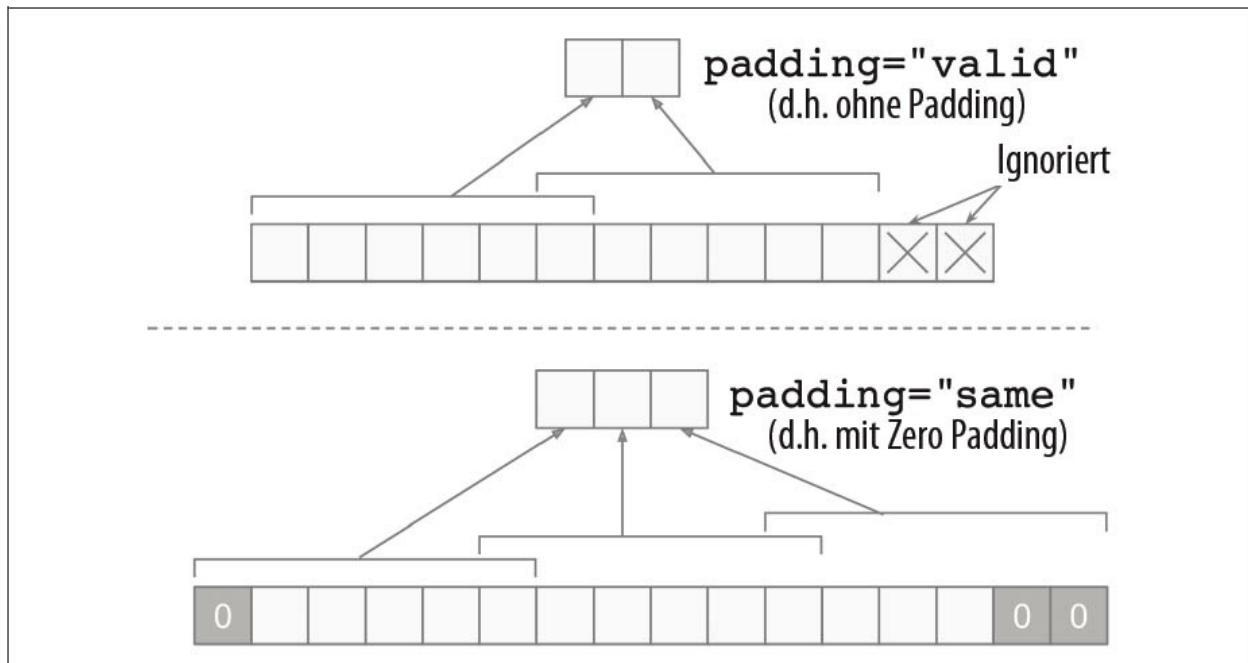


Abbildung 14-7: Padding gleich "same" oder "valid" – Breite der Eingabe: 13, Filterbreite: 6, Schrittweite: 5

In diesem Beispiel haben wir die Filter von Hand definiert, aber in einem echten CNN würden Sie normalerweise Filter als trainierbare Variablen definieren, sodass das neuronale Netz lernen kann, welche Filter am besten funktionieren (wie weiter oben erläutert). Statt die Variablen manuell zu erstellen, nutzen Sie die Schicht `keras.layers.Conv2D`:

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,  
                           padding="same", activation="relu")
```

Dieser Code erzeugt eine Conv2D-Schicht mit 32 Filtern und einer Schrittweite von 1 (sowohl horizontal wie auch vertikal) sowie einem "same"-Padding und wendet die ReLU-Aktivierungsfunktion auf ihre Ausgabe an. Wie Sie sehen können, 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), 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 Ausgabe des Convolutional Layer $200 \times 150 \times 100 \times 32 = 96$ Millionen Bits (12 MB) im RAM.⁸ Und dies nur für einen Datenpunkt! Wenn ein Batch beim Trainieren 100 Datenpunkte enthält, verbraucht diese Schicht mehr als 1,2 GB RAM!

Während der Inferenz (d.h. beim Treffen von Vorhersagen für einen neuen Datenpunkt) kann 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. Alternativ reduzieren Sie die Dimensionalität über die Schrittweite oder lassen einige Schichten aus. 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 Layers

Sofern 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).

Wie bei Convolutional Layers ist jedes Neuron in einem Pooling Layer mit einer begrenzten Anzahl Neuronen der vorherigen 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 14-8](#) zeigt einen *Max-Pooling Layer*, die verbreitetste Art eines Pooling Layer. In diesem Beispiel verwenden wir einen *Pooling-Kernel* der Größe 2×2 ⁹ einen stride von 2 und kein Padding. Nur der größte Eingabewert in jedem Wahrnehmungsfeld wird an die nächste Schicht übergeben, während alle übrigen Eingaben verworfen werden. So sind beispielsweise im unteren linken Wahrnehmungsfeld in [Abbildung 14-8](#) die Eingabewerte 1, 5, 3, 2, daher wird nur der maximale Wert 5 an die nächste Schicht weitergegeben. Aufgrund der Schrittweite von 2 hat das Ausgabebild die halbe Höhe und halbe Breite des Eingabebilds (abgerundet, da wir kein Padding verwenden).

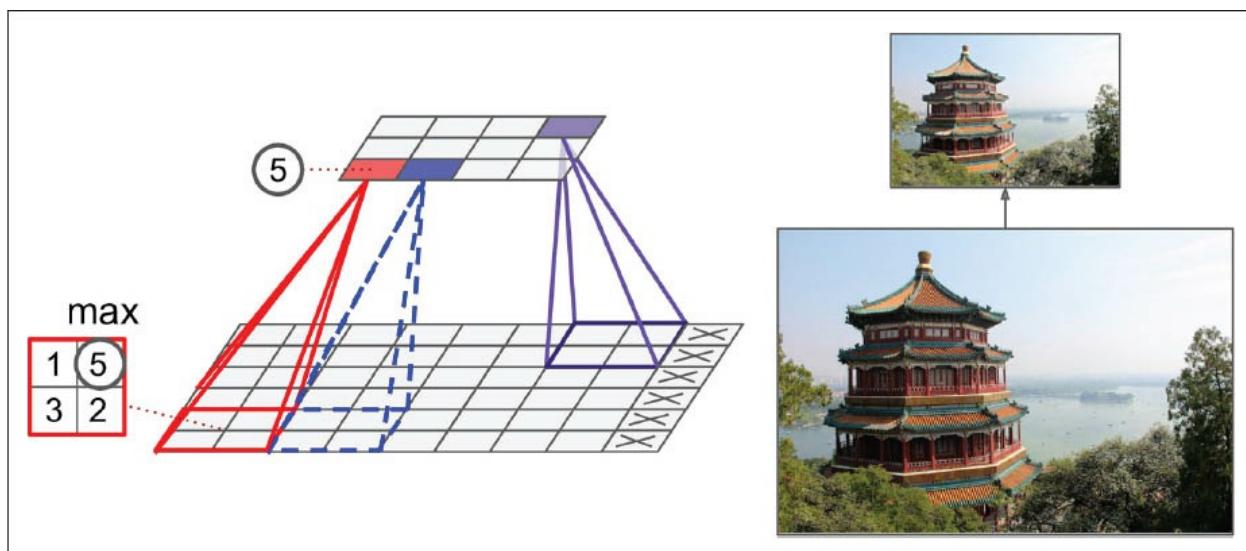


Abbildung 14-8: Max-Pooling Layer (Pooling-Kernel der Größe 2×2 , Schrittweite 2, kein Padding)

Ein Pooling Layer arbeitet normalerweise unabhängig für jeden Eingabekanal, daher ist die Ausgabetiefe die gleiche wie die Eingabetiefe.



Neben dem Verringern des Rechenaufwands, des Speicherbedarfs und der Anzahl der Parameter bringt ein Max-Pooling Layer auch noch eine gewisse Invarianz für kleine Verschiebungen mit, wie Sie in [Abbildung 14-9](#) sehen. Hier gehen wir davon aus, dass die hellen Pixel einen niedrigeren Wert als die dunklen Pixel haben und die drei Bilder (A, B, C) durch einen Max-Pooling Layer mit einem 2×2 -Kernel und einer Schrittweite von 2 laufen. Die Bilder B und C entsprechen Bild A, sind aber um einen bzw. zwei Pixel nach rechts verschoben. Wie Sie sehen können, sind die Ausgaben des Max-Pooling Layer für die Bilder A und B identisch. Das meinen wir mit Translationsinvarianz. Für Bild C ist die Ausgabe verschieden: Sie ist um ein Pixel nach rechts geschoben (aber es gibt immer noch eine Invarianz von 50%). Indem Sie in einem CNN alle paar Schichten einen Max-Pooling Layer einfügen, ist es möglich, auch im größeren Maßstab eine gewisse Translationsinvarianz zu erreichen. Zudem bietet das Max-Pooling eine gewisse Robustheit gegen Rotation und etwas Robustheit gegen Skalierung. Solche Invarianzen (auch wenn sie begrenzt sind) können nützlich sein, wenn die Vorhersage nicht von solchen Details abhängen soll, wie zum Beispiel in Klassifikationsaufgaben.

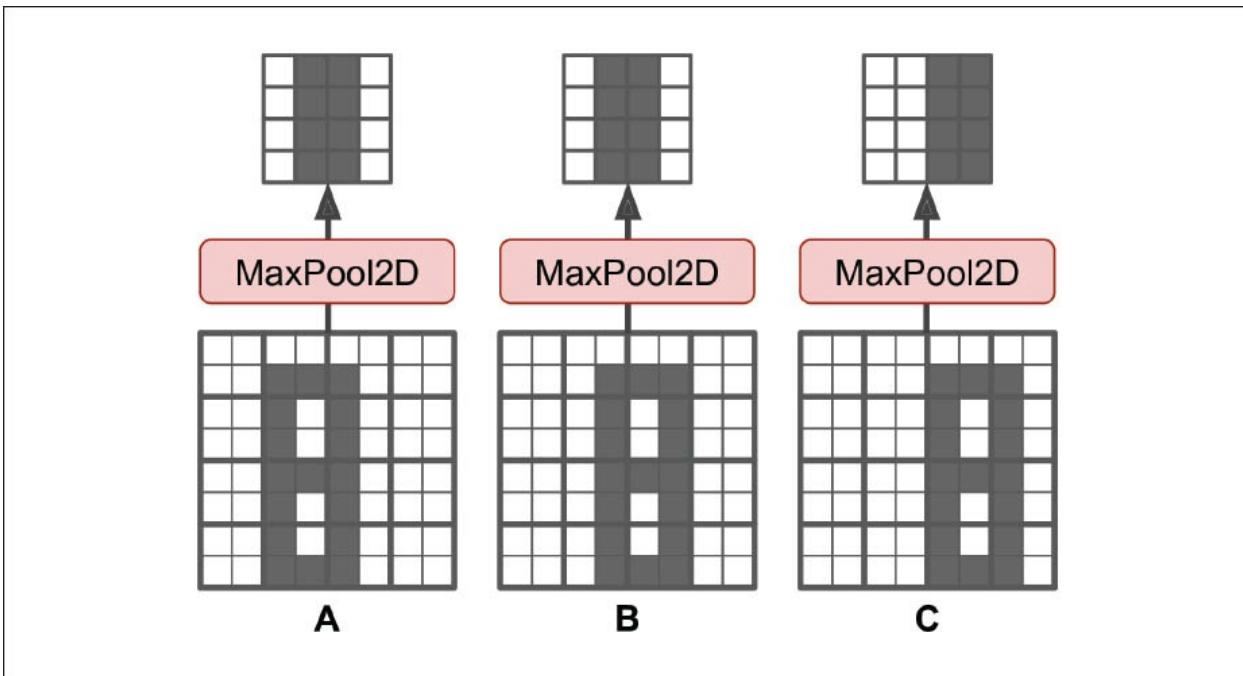


Abbildung 14-9: Invarianz gegenüber kleinen Translationen

Das Max-Pooling hat aber auch Nachteile. So ist es vor allem offensichtlich sehr destruktiv: Selbst mit einem kleinen 2×2 -Kernel und einer Schrittweite von 2 ist die Ausgabe in beide Richtungen zweimal kleiner (und die Fläche damit viermal kleiner), womit 75% der Eingabewerte verloren gehen. Und in manchen Anwendungen ist eine Invarianz nicht erwünscht. Nehmen wir die semantische Segmentierung (die Aufgabe, jedes Pixel in einem Bild dem Objekt zuzuordnen, zu dem es gehört, worauf wir später noch eingehen): Wurde die Eingabe um ein Pixel nach rechts verschoben, sollte offensichtlich auch die Ausgabe um ein Pixel nach rechts

verschoben sein. In diesem Fall ist das Ziel eine *Äquidistanz*, keine Invarianz: Eine kleine Änderung an den Eingabewerten sollte zu einer passenden kleinen Änderung an der Ausgabe führen.

Implementierung in TensorFlow

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 . Die Schrittweite entspricht standardmäßig der Kernelgröße, sodass dieser Layer eine Schrittweite von 2 (sowohl horizontal wie auch vertikal) nutzt. Standardmäßig kommt "valid"-Padding zum Einsatz (also keines):

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

Um einen *Average-Pooling Layer* zu erzeugen, verwenden Sie einfach AvgPool2D statt MaxPool2D. Wie zu erwarten, funktioniert er genauso wie ein Max-Pooling Layer, nur dass er den Mittelwert statt des Maximums ermittelt. Average-Pooling Layers waren sehr verbreitet, aber heutzutage kommen meist Max-Pooling Layers zu Einsatz, da sie im Allgemeinen eine bessere Leistung zeigen. Das mag überraschen, da das Berechnen des Mittelwerts normalerweise weniger Informationen verloren gehen lässt als das Berechnen des Maximums. Andererseits bewahrt Max-Pooling nur die stärksten Merkmale und verwirft alle bedeutungslosen, daher erhalten die nächsten Schichten ein saubereres Signal, mit dem sie arbeiten können. Zudem bietet Max-Pooling eine stärkere Translationsinvarianz als Average-Pooling, und es erfordert etwas weniger Rechenaufwand.

Beachten Sie, dass Max-Pooling und Average-Pooling entlang der Tiefendimension statt entlang der räumlichen Dimension ausgeführt werden können, was aber nicht so verbreitet ist. Damit kann das CNN lernen, auf verschiedenste Merkmale invariant zu reagieren. Es könnte zum Beispiel mehrere Filter lernen, von denen jeder eine andere Rotation des gleichen Musters erkennt (beispielsweise handgeschriebene Ziffern, siehe [Abbildung 14-10](#)), und der Max-Pooling Layer für die Tiefe stellt sicher, dass die Ausgabe unabhängig von der Rotation ist. Das CNN könnte genauso lernen, invariant auf anderes zu reagieren: Dicke, Helligkeit, Neigung, Farbe und so weiter.

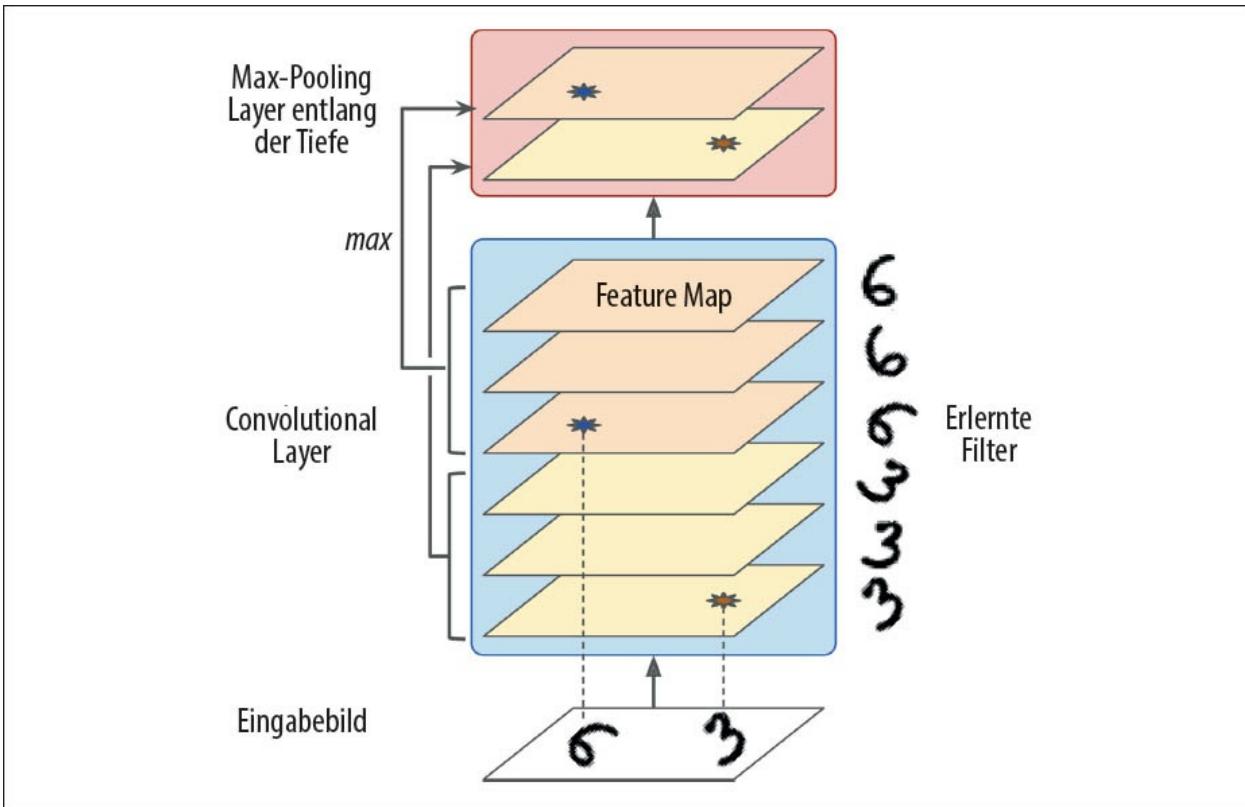


Abbildung 14-10: Max-Pooling entlang der Tiefe kann dem CNN helfen, Invarianzen zu lernen.

Keras besitzt keinen Max-Pooling Layer für die Tiefe, aber die Low-Level-Deep-Learning-API von TensorFlow hat einen: Verwenden Sie einfach die Funktion `tf.nn.max_pool()` und geben Sie Kernelgröße und Schrittweiten als 4-Tupel an (also Tupel der Größe 4). Die ersten drei Werte sollten dabei 1 sein – das legt fest, dass Kernelgröße und Schrittweite entlang der Batch-, Höhen- und Breitendimensionen 1 sein sollen. Der letzte Wert gibt die Kernelgröße und Schrittweite für die Tiefendimension an – zum Beispiel 3 (es muss ein Teiler der Eingabetiefe sein und funktioniert beispielsweise nicht, wenn die vorherige Schicht 20 Feature Maps ausgibt, da 20 kein Mehrfaches von 3 ist):

```
output = tf.nn.max_pool(images,
                        ksize=(1, 1, 1, 3),
                        strides=(1, 1, 1, 3),
                        padding="valid")
```

Wollen Sie dies als Schicht in Ihre Keras-Modelle aufnehmen, verpacken Sie die Funktion in eine Lambda-Schicht (oder erstellen Sie sich eine eigene Keras-Schicht):

```
depth_pool = keras.layers.Lambda(
    lambda x: tf.nn.max_pool(x, ksize=(1, 1, 1, 3), strides=(1, 1, 1, 3),
```

```
padding="valid"))
```

Ein letzter Typ eines Pooling Layer, dem Sie in modernen Architekturen häufiger begegnen werden, ist der *Global Average Pooling Layer*. Er funktioniert ganz anders, denn er berechnet nur die Mittelwerte jeder einzelnen Feature Map (wie ein Average-Pooling Layer mit einem Pooling-Kernel mit den gleichen räumlichen Dimensionen als Eingaben). Er gibt also eine einzelne Zahl pro Feature Map und Instanz aus. Auch wenn dies natürlich sehr destruktiv ist (die meisten Informationen in der Feature Map gehen verloren), kann er als Ausgabeschicht nützlich sein, wie wir später noch sehen werden. Um solch eine Schicht zu erstellen, verwenden Sie einfach die Klasse `keras.layers.GlobalAvgPool2D`:

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

Sie entspricht dieser einfachen Lambda-Schicht, die den Mittelwert über die räumlichen Dimensionen berechnet (Höhe und Breite):

```
global_avg_pool = keras.layers.Lambda(lambda x: tf.reduce_mean(x, axis=[1, 2]))
```

Sie kennen jetzt 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 aufeinandergestapelt (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 14-11](#)). 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).

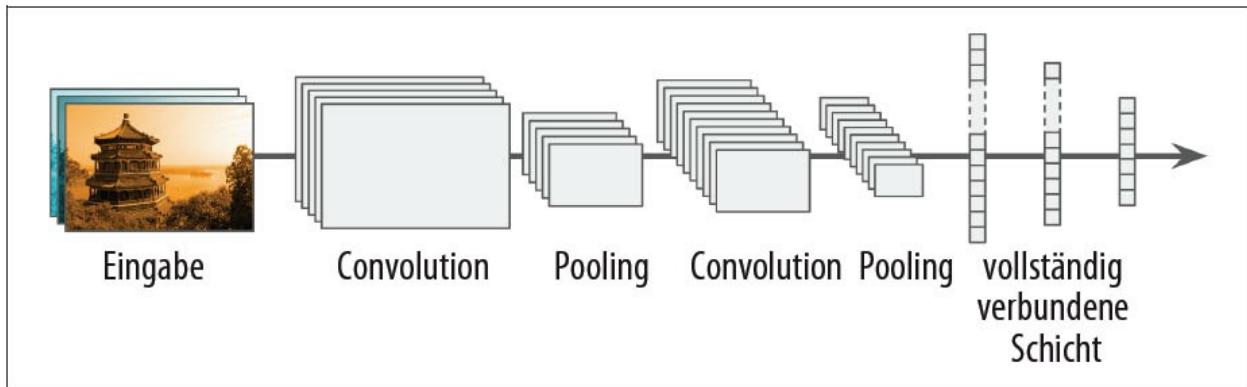


Abbildung 14-11: Typische Architektur eines CNN

Ein verbreiteter Fehler ist, zu große Convolution Kernels einzusetzen. Statt beispielsweise einen Convolutional Layer mit einem 5×5 -Kernel zu verwenden, stapeln Sie zwei Layers mit 3×3 -Kernels hintereinander: Es werden weniger Parameter und weniger Berechnungen gebraucht, und im Allgemeinen wird eine bessere Leistung gebracht. Eine Ausnahme bildet der erste Convolutional Layer: Er kann typischerweise einen großen Kernel besitzen (zum Beispiel 5×5), meist mit einer Schrittweite von 2 oder mehr – damit wird die räumliche Dimension des Bilds reduziert, ohne zu viele Informationen zu verlieren. Und da das Eingabebild im Allgemeinen drei Kanäle besitzt, wird das nicht zu teuer sein.

So können Sie ein einfaches CNN implementieren, mit dem Sie den Fashion-MNIST-Datensatz angehen (siehe [Kapitel 10](#)):

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                       input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])
```

Gehen wir dieses Modell durch:

- Die erste Schicht nutzt 64 ziemlich große Filter (7×7) und eine Schrittweite von 1, weil die Eingabebilder nicht sehr groß sind. Zudem wird `input_shape= [28, 28, 1]` gesetzt, weil die Bilder 28×28 Pixel und einen einzelnen Farbkanal haben (also Graustufenbilder sind).

- Als Nächstes haben wir einen Max-Pooling Layer mit einer Poolgröße von 2, also halbiert er jede räumliche Dimension.
- Dann wiederholen wir die gleiche Struktur zwei Mal: zwei Convolutional Layers, gefolgt von einem Max-Pooling Layer. Für größere Bilder könnten wir diese Struktur noch viel häufiger wiederholen (die Anzahl an Wiederholungen ist ein Hyperparameter, den Sie anpassen können).
- Beachten Sie, dass die Anzahl an Filtern wächst, wenn wir das CNN zu seiner Ausgabeschicht hochklettern (initial sind es 64, dann 128, dann 256): Es ist sinnvoll, dass sie wachsen, da die Anzahl an Low-Level-Merkmalen oft ziemlich gering ist (zum Beispiel kleine Kreise, horizontale Linien), es aber viele verschiedene Möglichkeiten gibt, sie zu High-Level-Merkmalen zu kombinieren. Es ist ein übliches Vorgehen, die Anzahl an Filtern nach jedem Pooling Layer zu verdoppeln: Da ein Pooling Layer jede räumliche Dimension halbiert, können wir es uns leisten, die Anzahl an Feature Maps in der nächsten Schicht zu verdoppeln, ohne Sorge haben zu müssen, dass uns die Anzahl der Parameter, die Speicherauslastung oder die Rechenlast explodiert.
- Danach folgt das vollständig verbundene Netz, bestehend aus zwei verborgenen Dense-Schichten und einer Dense-Ausgabeschicht. Beachten Sie, dass wir deren Eingaben flachklopfen müssen, da ein Dense-Netz ein eindimensionales Array mit Merkmalen für jede Instanz erwartet. Wir fügen auch noch zwei Drop-out-Schichten hinzu mit einer Drop-out-Rate von jeweils 50%, um Overfitting zu reduzieren.

Dieses CNN erreicht mit dem Testdatensatz über 92% Genauigkeit. Es ist nicht State of the Art, aber ziemlich gut und ganz klar besser als das, was wir in [Kapitel 10](#) mit Dense-Netzen erreicht haben.

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 sechs Jahren von über 26% auf weniger als 2,3% 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 1.000 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).

LeNet-5

Die LeNet-5-Architektur (<https://homl.info/lenet5>)¹⁰ ist vermutlich die am besten bekannte CNN-Architektur. Wie oben erwähnt, wurde sie von Yann LeCun im Jahr 1998 erstellt und im großen Stil zur Erkennung handgeschriebener Ziffern (MNIST) eingesetzt. Sie besteht aus den in [Tabelle 14-1](#) aufgeführten Schichten.

Tabelle 14-1: LeNet-5-Architektur

Schicht	Typ	Maps	Größe	Kernelgröße	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, deswegen schrumpft die Größe des Bilds beim Durchlaufen des Netzes immer weiter.
- Die Average-Pooling Layers sind etwas komplexer als gewöhnlich: Jedes Neuron berechnet den Mittelwert seiner Eingaben, 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 ([Seite 8](#)) im Originalartikel¹⁰.
- Die Ausgabeschicht ist ein wenig besonders: Anstatt die Matrixmultiplikation 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/exdb/lenet/index.html>) von Yann LeCun (im Abschnitt »LENET«) enthält großartige Demos, in denen LeNet-5 Ziffern klassifiziert.

AlexNet

Die CNN-Architektur *AlexNet* (<https://hml.info/80>)¹¹ 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 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 14-2](#) stellt diese Architektur vor.

Tabelle 14-2: AlexNet-Architektur

Schicht	Typ	Maps	Größe	Kernelgröße	Schrittweite	Padding	Aktivierung
Out	vollst. verbunden	–	1000	–	–	–	Softmax
F10	vollst. verbunden	–	4096	–	–	–	ReLU
F9	vollst. verbunden	–	4096	–	–	–	ReLU
S8	Max-Pooling	256	6×6	3×3	2	valid	–
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	–
C1	Convolution	96	55×55	11×11	4	valid	ReLU
In	Input	3 (RGB)	227×227	–	–	–	–

Um Overfitting zu vermeiden, verwendeten die Autoren zwei Regularisierungstechniken: Erstens wurde beim Trainieren auf Ausgaben der Schichten F8 und F9 Drop-out (siehe [Kapitel 11](#)) angewandt (mit einer Drop-out-Rate von 50%). Zweitens wurde *Data Augmentation* verwendet, indem die Trainingsbilder zufällig um unterschiedliche Beträge verschoben, horizontal gespiegelt und indem die Lichtverhältnisse verändert wurden.

Data Augmentation

Data Augmentation erhöht künstlich die Größe des Trainingsdatensatzes, indem es viele realistische Varianten jeder Trainingsinstanz generiert. Damit wird Overfitting verringert, und es wird so zu einer Regularisierungstechnik. Die generierten Instanzen sollten so realistisch wie möglich sein: Idealerweise sollte ein Mensch bei einem Bild aus dem angereicherten Trainingsdatensatz nicht dazu in der Lage sein, zu entscheiden, ob es generiert wurde oder nicht. Es reicht nicht, einfach weißes Rauschen hinzuzufügen – die Veränderungen sollten lernbar sein (weißes Rauschen ist es nicht).

Sie könnten zum Beispiel jedes Bild im Trainingsdatensatz ein wenig (aber unterschiedlich viel) verschieben, drehen und in der Größe verändern und die daraus entstehenden Bilder zum Trainingsdatensatz hinzufügen (siehe [Abbildung 14-12](#)). Das zwingt das Modell, auf Abweichungen in der Position, der Orientierung und der Größe der Objekte im Bild tolerant zu reagieren. Damit ein Modell toleranter auf unterschiedliche Beleuchtungssituationen reagiert, können Sie einfach viele Bilder mit unterschiedlichen Kontrasten erzeugen. Im Allgemeinen können Sie die Bilder auch horizontal spiegeln (außer bei Text und anderen asymmetrischen Objekten). Durch das Kombinieren dieser Transformationen können Sie die Größe Ihres Trainingsdatensatzes sehr gut erhöhen.

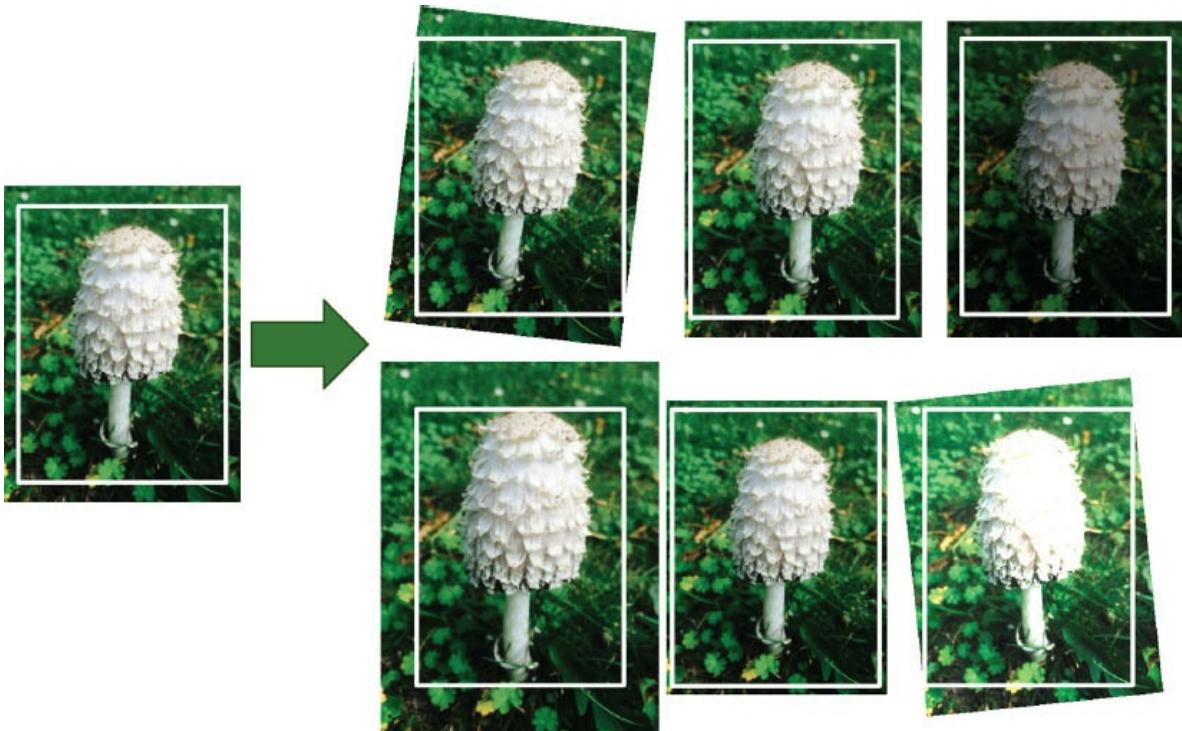


Abbildung 14-12: Neue Trainingsinstanzen aus bestehenden generieren

AlexNet führt unmittelbar nach dem ReLU-Schritt der Schichten C1 und C3 auch einen kompetitiven Normalisierungsschritt durch, die *Local Response Normalization* (LRN). Bei dieser Art Normalisierung hemmen 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. Das verbessert im Endeffekt die Verallgemeinerungsfähigkeit. [Formel 14-2](#) zeigt, wie LRN angewendet wird.

Formel 14-2: 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 hemmt 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 Funktion `tf.nn.local_response_normalization()` implementieren (die Sie in einer Lambda-Schicht verpacken können, wenn Sie sie in einem Keras-Modell einsetzen wollen).

Eine Variante von AlexNet namens *ZF Net* (<https://homl.info/zfnet>)¹² 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, der Schrittweite und so weiter).

GoogLeNet

Die GoogLeNet-Architektur (<https://homl.info/81>) wurde von Christian Szegedy et al. aus dem Google-Research-Team entwickelt¹³ 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 vorherige CNNs war (siehe Abbildung 14-14). Dies wurde durch Subnetze ermöglicht, die *Inception-Module*,¹⁴ 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 14-13 zeigt die Architektur eines Inception-Moduls. Die Notation » $3 \times 3 + 1(S)$ « bedeutet, dass die Schicht einen Kernel der Größe 3×3 , Schrittweite 1 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 Kernelgrößen (1×1 , 3×3 und 5×5), wodurch Muster in unterschiedlichen Größenordnungen erfasst werden. Jede einzelne Schicht verwendet Schrittweite 1 und "same"-Padding (sogar der Max-Pooling Layer), sodass alle Ausgaben die gleiche Höhe und Breite wie ihre 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 (die Achse ist die Tiefe).

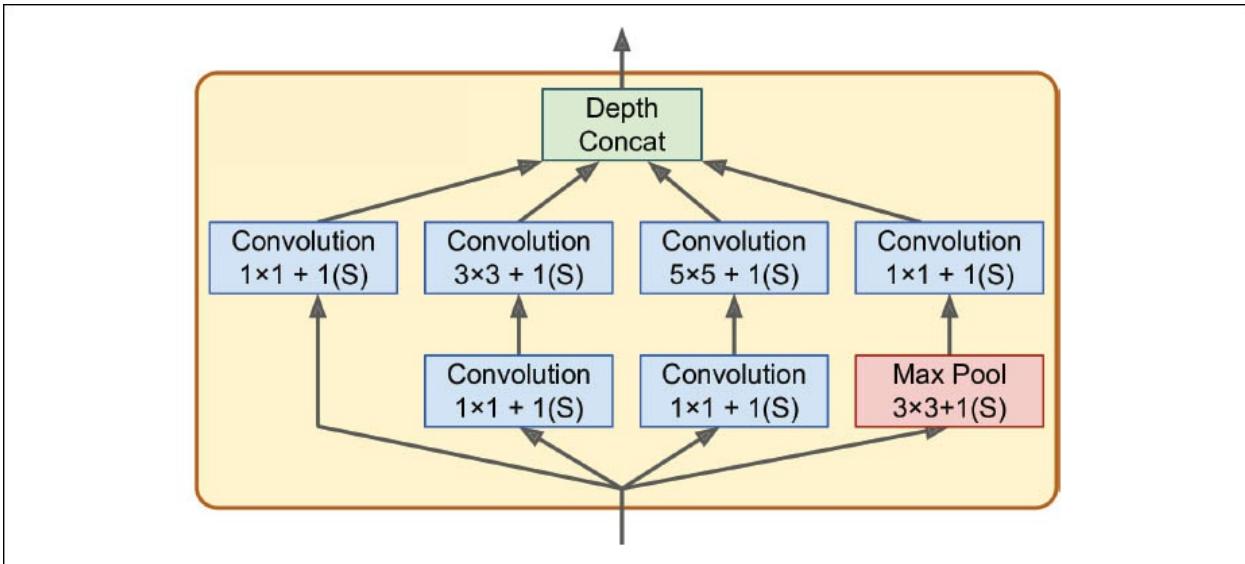


Abbildung 14-13: Inception-Modul

Sie fragen sich vielleicht, warum die Inception-Module Convolutional Layers mit Kernen der Größe 1×1 enthalten. Können diese Schichten überhaupt irgendwelche Merkmale erfassen, wenn sie nur genau ein Pixel betrachten? Diese Schichten sind für drei Dinge gut:

- Auch wenn sie keine räumlichen Muster erfassen können, sind sie dazu in der Lage, Muster entlang der Tiefendimension zu erkennen.
- Sie sind so konfiguriert, dass sie viel weniger Feature Maps ausgeben, als ihre Eingaben enthalten. Diese Schichten dienen als *Flaschenhals* zur Dimensionsreduktion. Das reduziert den Rechenaufwand und die Anzahl der Parameter, was das Training beschleunigt und die Generalisierung verbessert.
- Jedes Paar von Convolutional Layers ($[1 \times 1, 3 \times 3]$ und $[1 \times 1, 5 \times 5]$) verhält sich 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 dies, dass jeder hinzugefügte Inception-Layer sechs zusätzliche Hyperparameter mit sich bringt.

Betrachten wir nun die Architektur des GoogLeNet-CNN (siehe Abbildung 14-14). Die Anzahl der Feature Maps, ausgegeben von jedem Convolutional Layer und jedem Pooling Layer, wird vor der Kernelgröße angezeigt. Die Architektur 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). 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 14-13). Sämtliche Convolutional Layers verwenden die ReLU-Aktivierungsfunktion.

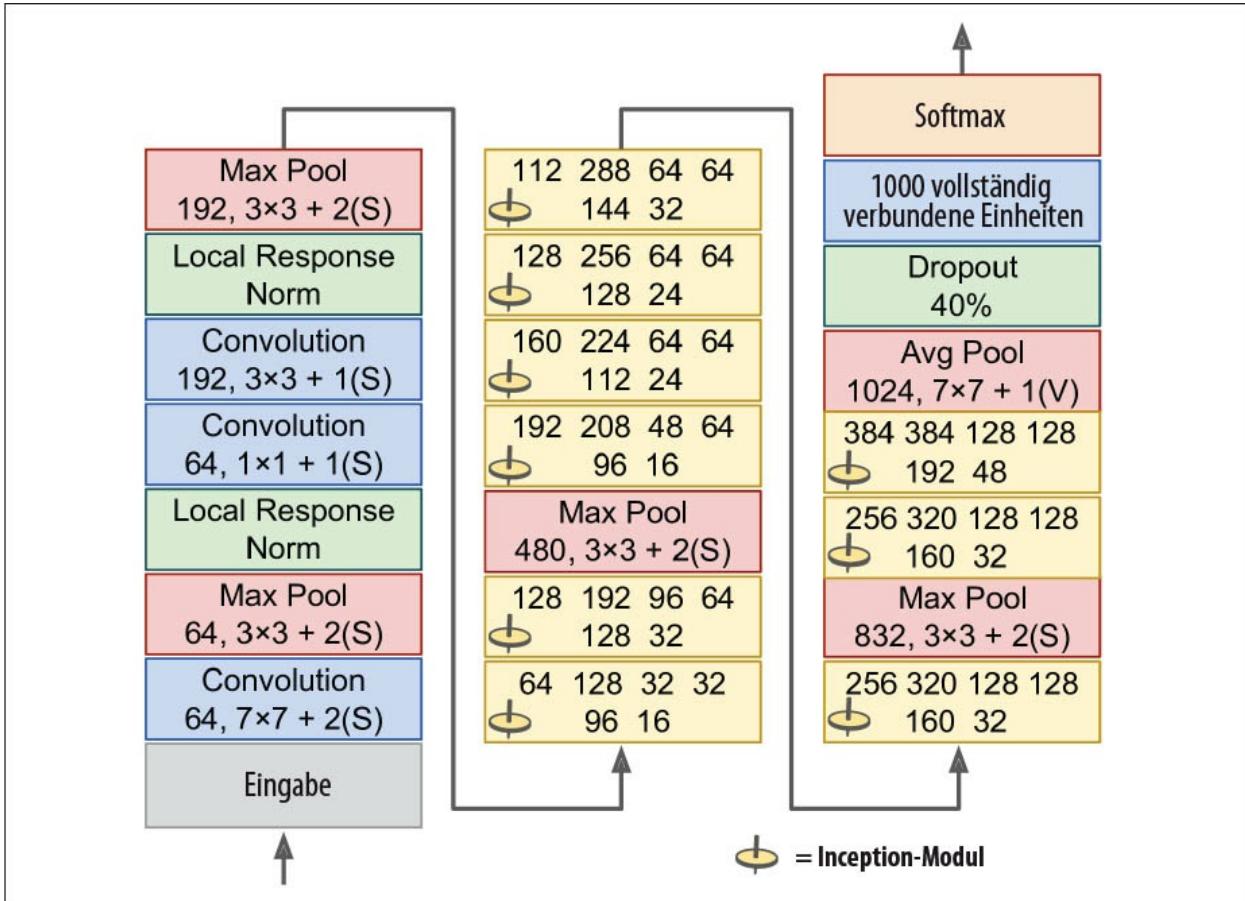


Abbildung 14-14: GoogLeNet-Architektur

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. Die erste Schicht nutzt einen großen Kernel, damit viele Informationen erhalten bleiben.
- Anschließend stellt der Local Response Normalization Layer sicher, dass die vorherigen 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, schlaueren Convolutional Layer veranschaulichen.
- Ein weiterer Local Response Normalization Layer stellt sicher, dass die vorherigen 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 Global Average Pooling Layer gibt den Mittelwert jeder Feature Map aus: Damit verschwinden alle verbleibenden räumlichen Informationen, was in Ordnung ist, weil gar nicht mehr so viel davon übrig geblieben ist. Tatsächlich wird normalerweise davon ausgegangen, dass die Eingabebilder von GoogLeNet 224×224 Pixel groß sind, daher sind die Feature Maps nach fünf Max-Pooling Layers, die jeweils Höhe und Breite durch zwei teilen, nur noch 7×7 Pixel groß. Zudem handelt es sich um eine Klassifikationsaufgabe, nicht um eine Lokalisierungsaufgabe, daher ist es egal, wo sich das Objekt befindet. Dank der Dimensionalitätsreduktion durch diese Schicht 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 1.000 Einheiten (da es 1.000 Kategorien gibt) und eine Softmax-Aktivierungsfunktion geben 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 dem 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.

Später wurde noch eine Reihe von Varianten der GoogLeNet-Architektur von Forschern bei Google vorgeschlagen, unter anderem Inception-v3 und Inception-v4, die etwas andere Inception-Module verwenden und eine noch bessere Performance erreichen.

VGGNet

Zweiter in der ILSVRC 2014 Challenge war VGGNet (<https://homl.info/83>)¹⁵, entwickelt von Karen Simonyan und Andrew Zisserman vom Visual Geometry Group (VGG) Research Lab an der Oxford University. Er hat eine sehr einfache und klassische Architektur mit zwei oder drei Convolutional Layers und einem Pooling Layer, dann wieder zwei oder drei Convolutional Layers und einem Pooling Layer und so weiter (bis zu 16 oder 19 Convolutional Layers – abhängig von der VGG-Variante), plus ein abschließendes Dense-Netz mit zwei verborgenen Schichten und der Ausgabeschicht. VGGNet nutzt nur 3×3 -Filter, davon aber viele.

ResNet

Kaiming He et al. gewannen die ILSVRC 2015 Challenge mit dem *Residual Network* (<https://homl.info/82>) (oder *ResNet*)¹⁶. Es erzielte eine erstaunliche Top-5-Fehlerquote unter 3,6%. Die Gewinnervariante nutzte ein extrem tiefes CNN mit 152 Schichten (andere Varianten hatten 34, 50 und 101 Schichten). Es bestätigte den allgemeinen Trend: Modelle werden tiefer und tiefer, haben aber immer weniger Parameter. 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 das 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 14-15).

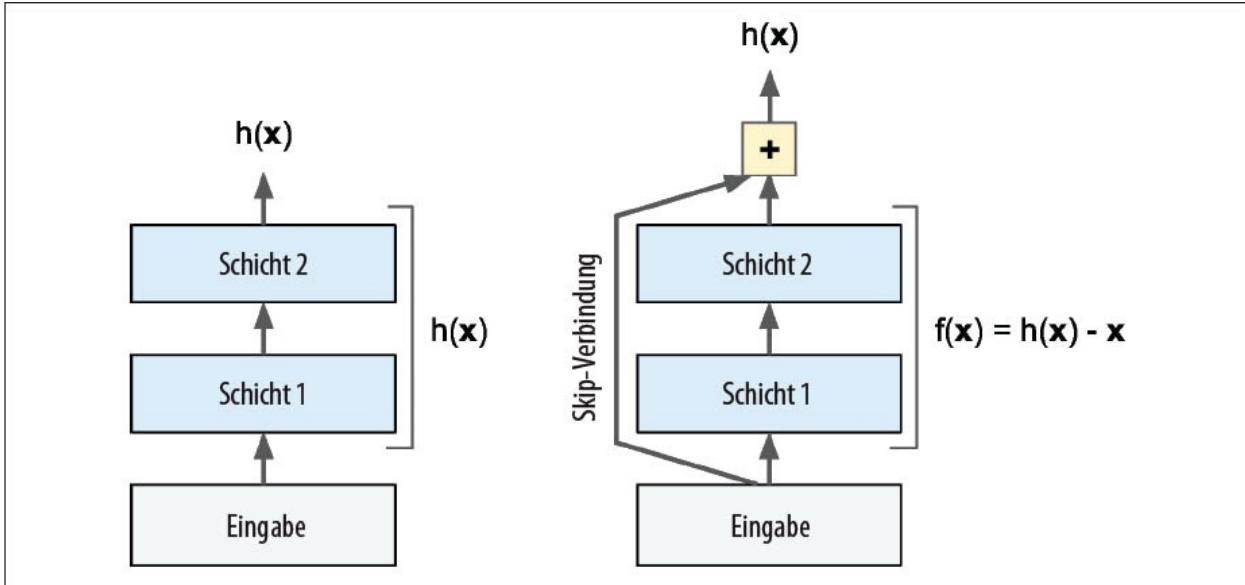


Abbildung 14-15: Residual Learning

Wenn Sie ein gewöhnliches neuronales Netz initialisieren, liegen dessen Gewichte um null, sodass das Netz Werte um null ausgibt. Fügen Sie eine Skip-Verbindung hinzu, 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.

Fügen Sie außerdem viele Skip-Verbindungen hinzu, kann das Netz auch dann Fortschritte erzielen, wenn mehrere Schritte noch nicht mit dem Lernen begonnen haben (siehe Abbildung 14-16). 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.

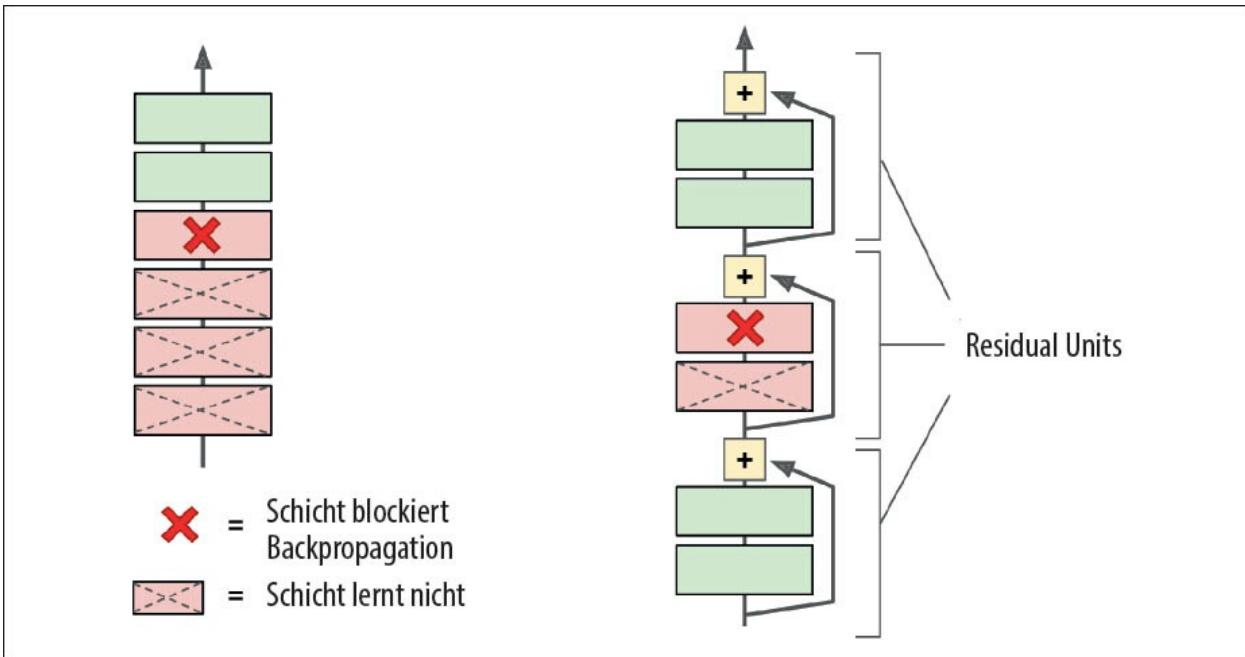


Abbildung 14-16: Gewöhnliches Deep-Learning-Netz (links) und Deep-Residual-Netz (rechts)

Betrachten wir nun die Architektur von ResNet (siehe [Abbildung 14-17](#)). Sie ist eigentlich überraschend einfach. Sie beginnt und endet genau so 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 (aber ohne Pooling Layer!) mit Batchnormalisierung (BN), der ReLU-Aktivierungsfunktion, Kernels der Größe 3×3 und Erhalt der räumlichen Dimensionen (Schrittweite 1, "same"-Padding).

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 in [Abbildung 14-17](#)). 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 14-18](#)).

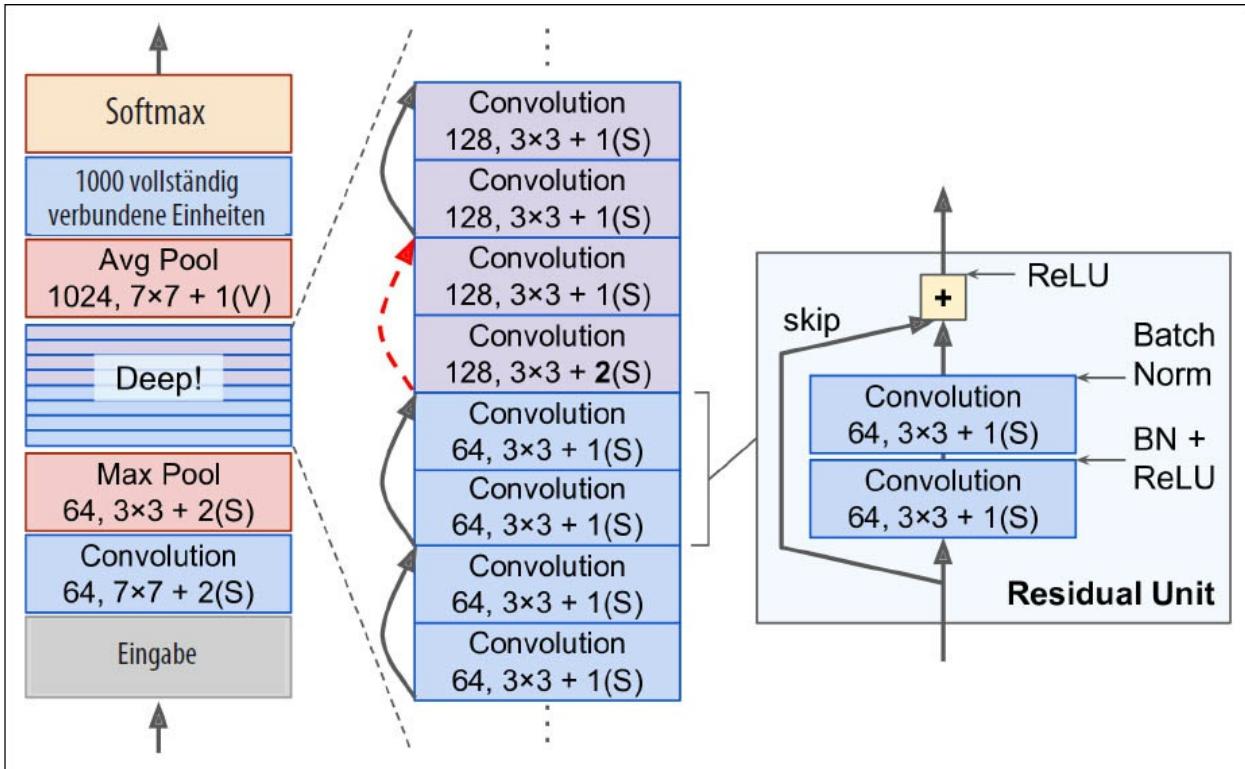


Abbildung 14-17: ResNet-Architektur

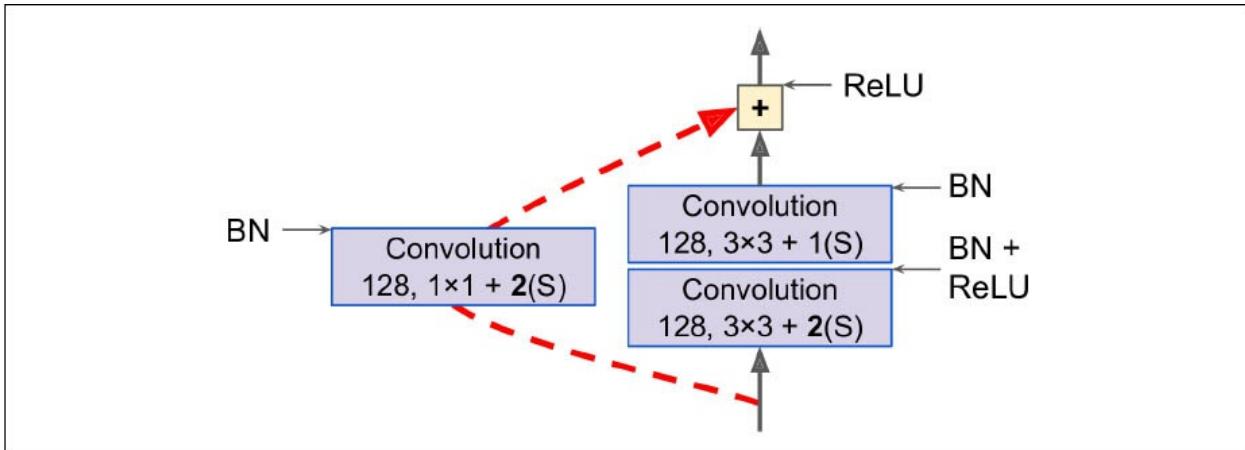


Abbildung 14-18: Skip-Verbindung beim Ändern der Größe und Tiefe von Feature Maps

ResNet-34 ist ein ResNet mit 34 Schichten (nur die Convolutional Layers und die vollständig verbundene Schicht werden gezählt).¹⁷ 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. Wir werden diese Architektur später noch implementieren.

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. ResNet-152 enthält 3 solcher RUs, die 256 Maps ausgeben, anschließend 8 RUs mit 512 Maps, stolze 36 RUs mit 1.024 Maps und schließlich 3 RUs mit 2.048 Maps.



Googles Inception-v4 (<https://homl.info/84>)-Architektur¹⁸ verbindet die Ideen von GoogLeNet und ResNet und erreicht eine Top-5-Fehlerrate nahe bei 3% bei der ImageNet-Klassifikation.

Xception

Als weitere Variante der GoogLeNet-Architektur lohnt Xception (<https://homl.info/xception>)¹⁹ einen Blick (der Name steht für *Extreme Inception*). Sie wurde im Jahr 2016 von François Chollet (dem Autor von Keras) vorgeschlagen und schlägt Inception-v3 bei einer sehr großen Bilderkennungsaufgabe deutlich (350 Millionen Bilder und 17.000 Kategorien). Wie Inception-v4 verbindet es die Ideen von GoogLeNet und ResNet, ersetzt aber die Inception-Module durch einen speziellen Schichttyp namens *Depthwise Separable Convolution Layer* (oder kurz *Separable Convolution Layer*²⁰). Diese Schichten wurden schon zuvor in manchen CNN-Architekturen eingesetzt, aber nicht in einer so zentralen Rolle wie bei der Xception-Architektur. Während normale Convolutional Layers Filter verwenden, die versuchen, räumliche Muster (zum Beispiel ein Oval) und kanalübergreifende Muster (zum Beispiel Mund + Nase + Augen = Gesicht) zu erkennen, trifft ein Separable Convolutional Layer die starke Annahme, dass räumliche Muster und kanalübergreifende Muster getrennt modelliert werden können (siehe Abbildung 14-19). Daher besteht er aus zwei Teilen: Der erste wendet einen einzelnen räumlichen Filter für jede Eingabe-Feature-Map an, dann sucht der zweite exklusiv nach kanalübergreifenden Mustern – es handelt sich dabei einfach um einen normalen Convolutional Layer mit 1×1 -Filtern.

Da Separable Convolutional Layers nur einen räumlichen Filter pro Eingabekanal haben, sollten Sie sie möglichst nicht nach Schichten verwenden, die zu wenig Kanäle haben, wie zum Beispiel die Eingabeschicht (ja, genau das ist in Abbildung 14-19 dargestellt, aber es dient hier nur der Illustration). Aus diesem Grund fängt die Xception-Architektur mit zwei normalen Convolutional Layers an, im Rest der Architektur kommen dann aber nur Separable Convolutions zum Einsatz (insgesamt 34), dazu ein paar Max-Pooling Layers und die üblichen abschließenden Schichten (ein Global Average Pooling Layer und eine Dense-Ausgabeschicht).

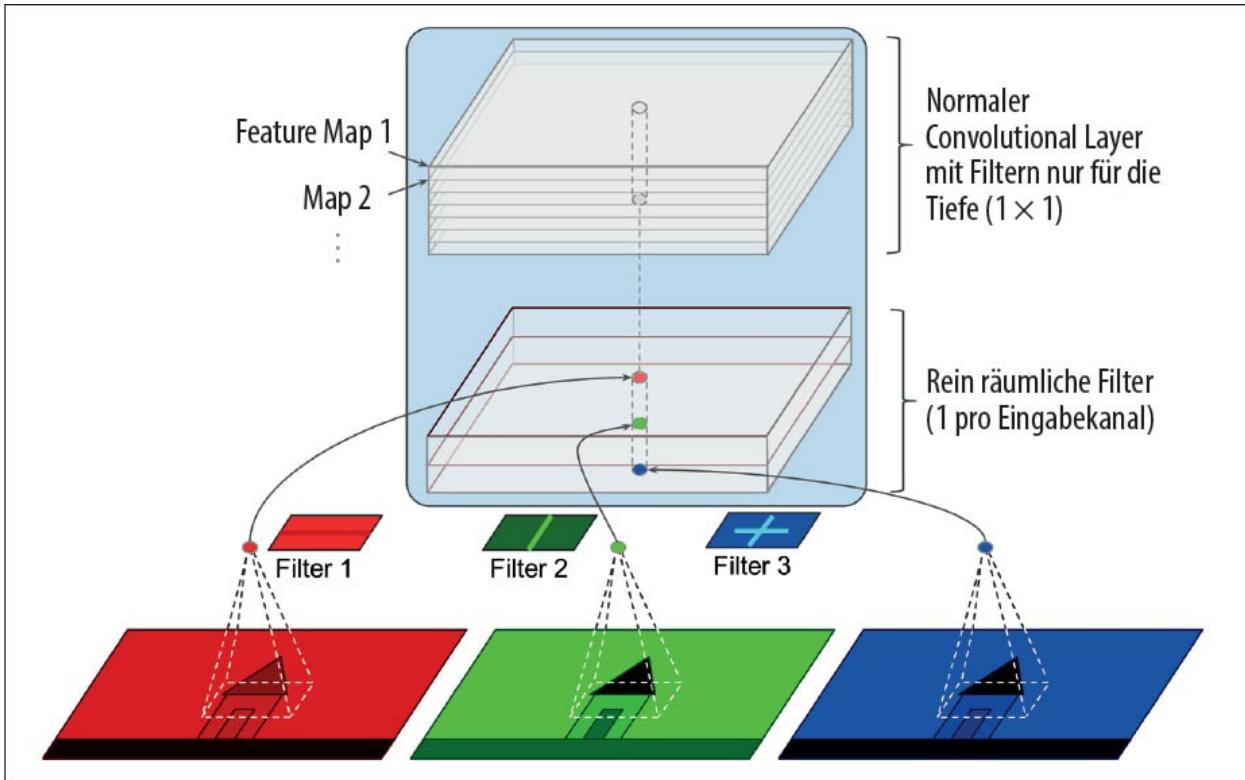


Abbildung 14-19: Depthwise Separable Convolutional Layer

Sie fragen sich vielleicht, warum Xception als Variante von GoogLeNet angesehen wird, da es doch überhaupt kein Inception-Modul mehr enthält. Nun, wie schon erklärt, enthält ein Inception-Modul Convolutional Layers mit 1×1 -Filtern, die nur Ausschau nach kanalübergreifenden Mustern halten. Aber die Convolutional Layers ganz oben sind normale Convolutional Layers, die sowohl nach räumlichen wie auch nach kanalübergreifenden Mustern suchen. Daher können Sie sich ein Inception-Modul als Vermittler zwischen einem normalen Convolutional Layer (der räumliche und kanalübergreifende Muster gemeinsam berücksichtigt) und einem Separable Convolutional Layer (der sie getrennt betrachtet) vorstellen. In der Praxis scheint es so zu sein, dass Separable Convolutional Layers im Allgemeinen besser funktionieren.

- ☞ Separable Convolutional Layers brauchen weniger Parameter, weniger Speicher und weniger Rechenaufwand als normale Convolutional Layers, und im Allgemeinen bieten sie sogar eine bessere Leistung, daher sollten Sie sich überlegen, standardmäßig auf sie zurückzugreifen (außer nach Schichten mit wenigen Kanälen).

Die ILSVRC 2016 Challenge wurde vom CUImage-Team der Chinese University of Hong Kong gewonnen. Das Team nutzte ein Ensemble unterschiedlichster Technologien, einschließlich eines ausgereiften Systems zur Objekterkennung namens GBD-Net (<https://homl.info/gbdnet>)²¹, um eine Top-5-Fehlerrate unter 3% zu erreichen. Auch wenn dieses Ergebnis unbestreitbar beeindruckend ist, steht die Komplexität der Lösung in Kontrast zur Einfachheit von ResNets. Zudem war ein Jahr später eine andere ziemlich einfache Architektur noch besser, wie wir jetzt sehen werden.

SENet

Die Gewinner-Architektur der ILSVRC 2017 Challenge war das Squeeze-and-Excitation Network (SENet) (<https://homl.info/senet>)²². Diese Architektur erweiternd bestehende Architekturen wie Inception Networks und ResNets und verbessert deren Performance. Damit konnte SENet den Wettbewerb mit einer erstaunlichen Top-Five-Fehlerrate von 2,25% gewinnen! Die erweiterten Versionen von Inception Networks und ResNets heißen *SE-Inception* und *SE-ResNet*. Die Leistungssteigerung beruht darauf, dass ein SENet ein kleines neuronales Netzwerk – *SE-Block* genannt – zu jeder Einheit der ursprünglichen Architektur hinzufügt (also zu jedem Inception-Modul oder jeder Residual Unit), wie in Abbildung 14-20 gezeigt wird.

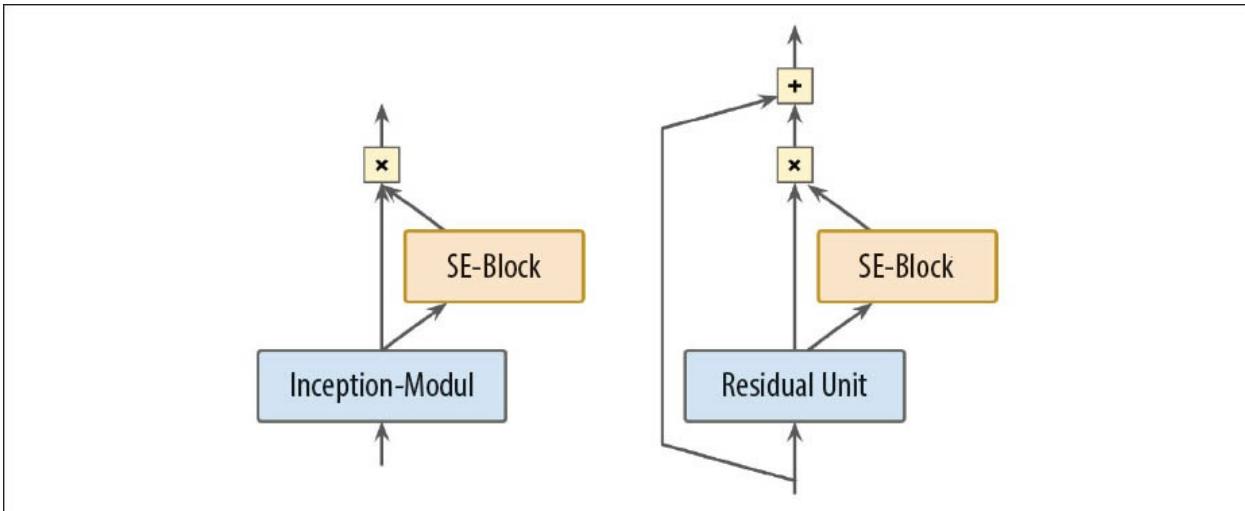


Abbildung 14-20: SE-Inception-Modul (links) und SE-ResNet Unit (rechts)

Ein SE-Block analysiert die Ausgabe der Einheit, mit der er verbunden ist, konzentriert sich ganz auf die Tiefendimension (er sucht nicht nach räumlichen Mustern) und lernt, welche Features normalerweise gemeinsam am aktivsten sind. Dann nutzt er diese Informationen, um die Feature Maps neu zu kalibrieren, wie Sie in Abbildung 14-21 sehen. Ein SE-Block kann beispielsweise lernen, dass Münden, Nasen und Augen in Bildern meist zusammen auftauchen – sehen Sie einen Mund und eine Nase, sollten Sie auch davon ausgehen, dass Augen zu sehen sind. Sieht der Block also eine starke Aktivierung in den Feature Maps für Mund und Nase, aber nur eine schwache in der für die Augen, wird er diese Feature Map verstärken (genauer – er reduziert irrelevante Feature Maps). Wurden die Augen mit etwas anderem verwechselt, wird dieses Rekalibrieren der Feature Map dabei helfen, die Mehrdeutigkeit aufzulösen.

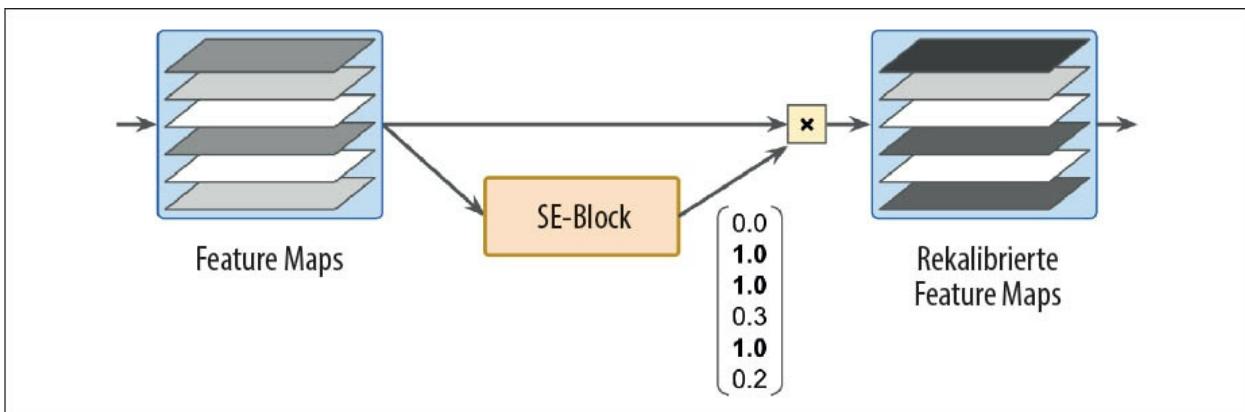


Abbildung 14-21: Ein SE-Block führt eine Rekalibrierung der Feature Map durch.

Ein SE-Block besteht aus nur drei Schichten: einem Global Average Pooling Layer, einer verborgenen Dense-Schicht mit ReLU-Aktivierungsfunktion und einer Dense-Ausgabeschicht mit der Sigmoid-Aktivierungsfunktion (siehe Abbildung 14-22).

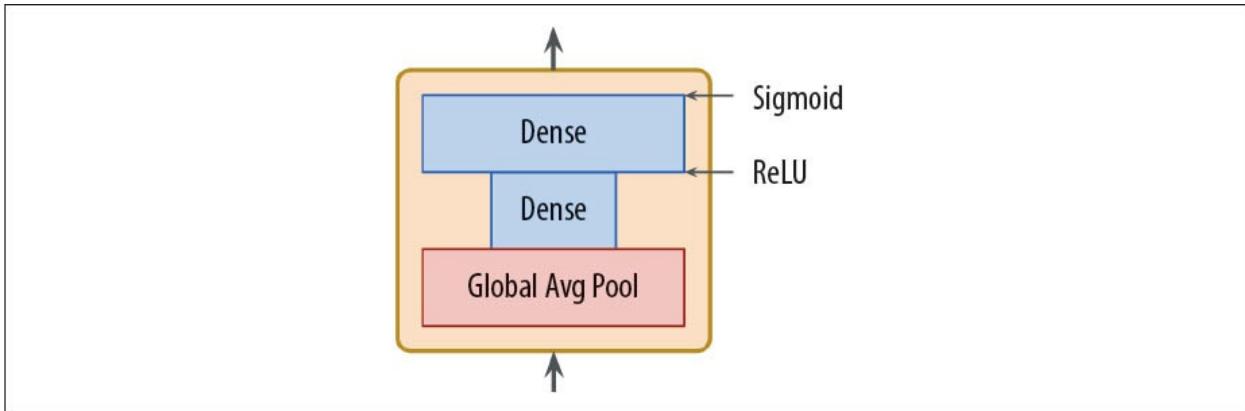


Abbildung 14-22: Architektur des SE-Blocks

Wie zuvor berechnet der Global Average Pooling Layer die mittlere Aktivierung für jede Feature Map. Besteht seine Eingabe beispielsweise aus 256 Feature Maps, wird er 256 Zahlen liefern, die die Gesamtreaktion jedes Filters darstellen. In der nächsten Schicht geschieht das »Squeeze«: Er hat deutlich weniger als 256 Neuronen – meist das 16-Fache weniger als die Anzahl an Feature Maps (zum Beispiel 16 Neuronen) –, sodass die 256 Zahlen in einen kleinen Vektor gequetscht werden (zum Beispiel 16 Dimensionen). Das ist eine niedrigdimensionale Vektordarstellung (also ein Embedding) der Verteilung der Merkmalsreaktionen. Dieser Flaschenhals-Schritt zwingt den SE-Block dazu, eine allgemeine Repräsentation der Merkmalskombinationen zu lernen (wir werden diesem Prinzip bei Autoencodern in Kapitel 17 erneut begegnen). Die Ausgabeschicht nimmt schließlich das Embedding und gibt einen Rekalibrierungsvektor mit einer Zahl pro Feature Map (hier also 256) aus, die jeweils zwischen 0 und 1 liegt. Die Feature Maps werden dann mit diesem Rekalibrierungsvektor multipliziert, sodass irrelevante Merkmale (mit einem niedrigen Rekalibrierungswert) herunterskaliert werden, während relevante Merkmale (mit einem Rekalibrierungswert nahe an 1) beibehalten werden.

Ein ResNet-34-CNN mit Keras implementieren

Die meisten bisher beschriebenen CNN-Architekturen lassen sich ziemlich einfach implementieren (auch wenn Sie normalerweise stattdessen ein vortrainiertes Netzwerk laden werden, wie wir noch sehen). Um den Prozess aufzuzeigen, wollen wir ein ResNet-34 mit Keras von Grund auf erstellen. Zuerst erzeugen wir eine Residual Unit-Schicht:

```
class ResidualUnit(keras.layers.Layer):

    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            keras.layers.Conv2D(filters, 3, strides=strides,
                               padding="same", use_bias=False),
            keras.layers.BatchNormalization(),
            self.activation,
            keras.layers.Conv2D(filters, 3, strides=1,
                               padding="same", use_bias=False),
            keras.layers.BatchNormalization()]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                keras.layers.Conv2D(filters, 1, strides=strides,
                                   padding="same", use_bias=False),
                keras.layers.BatchNormalization()]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
```

```

for layer in self.skip_layers:
    skip_Z = layer(skip_Z)
return self.activation(Z + skip_Z)

```

Wie Sie sehen, bildet dieser Code Abbildung 14-18 ziemlich nah ab. Im Konstruktor erzeugen wir alle erforderlichen Schichten: Die Hauptschichten sind die auf der rechten Seite des Diagramms, die Skip-Schichten die auf der linken Seite (nur erforderlich, wenn die Schrittweite größer als 1 ist). Dann lassen wir in der Methode `call()` die Eingaben durch die Hauptschichten und die Skip-Schichten (sofern vorhanden) laufen, addieren beide Ausgaben und wenden die Aktivierungsfunktion an.

Als Nächstes können wir das ResNet-34 mit einem Sequential-Modell bauen, da es sich wirklich nur um eine lange Folge von Schichten handelt (wir können jetzt mit der Klasse `ResidualUnit` jede Residual Unit als einzelne Schicht behandeln):

```

model = keras.models.Sequential()

model.add(keras.layers.Conv2D(64, 7, strides=2, input_shape=[224, 224, 3],
                           padding="same", use_bias=False))

model.add(keras.layers.BatchNormalization())

model.add(keras.layers.Activation("relu"))

model.add(keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"))

prev_filters = 64

for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))

    prev_filters = filters

model.add(keras.layers.GlobalAvgPool2D())

model.add(keras.layers.Flatten())

model.add(keras.layers.Dense(10, activation="softmax"))

```

Der einzige etwas knifflige Abschnitt in diesem Code ist die Schleife, die die `Residual Unit`-Schichten zum Modell hinzufügt: Wie schon erläutert, haben die ersten drei RUs 64 Filter, die nächsten vier dann 128 Filter und so weiter. Wir können die Schrittweite auf 1 setzen, wenn die Anzahl der Filter im Vergleich zur vorherigen RU gleich bleibt, ansonsten setzen wir sie auf 2. Dann fügen wir die `ResidualUnit` hinzu und aktualisieren schließlich `prev_filters`.

Es ist erstaunlich, dass wir mit weniger als 40 Zeilen Code das Modell nachbauen können, das die ILSVRC 2015 Challenge gewonnen hat! Das zeigt die Eleganz des ResNet-Modells und die Ausdrucksstärke der Keras-API. Es ist nicht viel schwerer, die anderen CNN-Architekturen zu implementieren. Aber Keras bringt viele dieser Architekturen schon mit, warum sollten wir sie dann nicht verwenden?

Vortrainierte Modelle aus Keras einsetzen

Im Allgemeinen müssen Sie keine Standardmodelle wie GoogLeNet oder ResNet selbst implementieren, da mit einer einzigen Codezeile vortrainierte Netze aus dem Paket `keras.applications` bereitgestellt werden können. So können Sie beispielsweise das ResNet-50-Modell – vorgenommen auf ImageNet – mit der folgenden Codezeile laden:

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

Das ist alles! Damit wird ein ResNet-50-Modell erstellt, und die Gewichte werden heruntergeladen, die mit dem ImageNet-Datensatz vorgenommen wurden. Um es zu verwenden, müssen Sie zuerst sicherstellen, dass die Bilder die richtige Größe besitzen. Ein ResNet-50-Modell erwartet Bilder mit 224×224 Pixeln (andere Modelle erwarten eventuell andere Größen, wie zum Beispiel 299×299 Pixel), daher verwenden wir die Funktion `tf.image.resize()` von TensorFlow, um die zuvor geladenen Bilder zu skalieren:

```
images_resized = tf.image.resize(images, [224, 224])
```

- ☞ Die Funktion `tf.image.resize()` behält das Seitenverhältnis nicht bei. Wenn das ein Problem ist, versuchen Sie, die Bilder vor dem Anpassen der Größe auf das passende Seitenverhältnis zu beschneiden. Beide Operationen können Sie in einem Aufruf mit `tf.image.crop_and_resize()` erledigen.

Die vorgenommenen Modelle gehen davon aus, dass die Bilder auf eine bestimmte Art und Weise vorverarbeitet wurden. In manchen Fällen müssen die Eingabewerte zwischen 0 und 1 oder -1 und 1 liegen. Jedes Modell stellt eine Funktion `preprocess_input()` bereit, die Sie zum Vorverarbeiten Ihrer Bilder anwenden können. Diese Funktionen nehmen an, dass die Pixelwerte im Bereich von 0 bis 255 liegen, daher müssen wir sie mit 255 multiplizieren (da wir sie zuvor auf den Bereich von 0 bis 1 skaliert haben):

```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
```

Jetzt können wir mit dem vorgenommenen Modell Vorhersagen treffen:

```
Y_proba = model.predict(inputs)
```

Wie immer ist die Ausgabe `Y_proba` eine Matrix mit einer Zeile pro Bild und einer Spalte pro Kategorie (in diesem Fall gibt es 1.000 Kategorien). Wollen Sie die besten K Vorhersagen

ausgeben lassen – einschließlich Kategorienname und der geschätzten Wahrscheinlichkeit für jede vorhergesagte Kategorie –, verwenden Sie die Funktion `decode_predictions()`. Für jedes Bild gibt sie ein Array mit den besten K Vorhersagen zurück, wobei jede Vorhersage als Array mit der Kategorienkennung²³, deren Name und dem zugehörigen Konfidenzwert geliefert wird:

```
top_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)

for image_index in range(len(images)):

    print("Bild #{}".format(image_index))

    for class_id, name, y_proba in top_K[image_index]:
        print("  {} - {:.12s} {:.2f}%".format(class_id, name, y_proba * 100))

    print()
```

Die Ausgabe sieht dann so aus:

Bild #0

n03877845 - palace	42.87%
n02825657 - bell_cote	40.57%
n03781244 - monastery	14.56%

Bild #1

n04522168 - vase	46.83%
n07930864 - cup	7.78%
n11939491 - daisy	4.87%

Die richtigen Kategorien (`monastery` und `daisy`) tauchen für beide Bilder unter den besten drei Ergebnissen auf. Das ist wirklich gut angesichts der Tatsache, dass das Modell aus etwa 1.000 Kategorien auswählen musste.

Wie Sie sehen, ist es sehr einfach, einen ziemlich guten Bildklassifizierer mit einem vortrainierten Modell zu erstellen. In `keras.applications` stehen noch andere Bilderkennungsmodelle bereit, zum Beispiel diverse ResNet-Varianten, GoogLeNetVarianten wie Inception-v3 und Xception, VGGNet-Varianten sowie MobileNet und MobileNetV2 (schlanke Modelle für den Einsatz in mobilen Anwendungen).

Aber was ist, wenn Sie eine Bilderkennung für Bildkategorien einsetzen wollen, die nicht Teil von ImageNet sind? In diesem Fall können Sie trotzdem von den vortrainierten Modellen profitieren und ein Transfer Learning durchführen.

Vortrainierte Modelle für das Transfer Learning

Wollen Sie einen Bildklassifizierer bauen, haben aber nicht ausreichend Trainingsdaten, ist es häufig eine gute Idee, die unteren Schichten eines vortrainierten Modells wiederzuverwenden, wie wir in [Kapitel 11](#) besprochen haben. Trainieren wir beispielsweise ein Modell zum Klassifizieren von Blumenbildern und verwenden wir dafür ein vortrainiertes Xception-Modell. Zuerst laden wir den Datensatz mithilfe von TensorFlow-Datasets (siehe [Kapitel 13](#)):

```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)

dataset_size = info.splits["train"].num_examples # 3670
class_names = info.features["label"].names # ["dandelion", "daisy", ...]
n_classes = info.features["label"].num_classes # 5

----
```

Beachten Sie, dass Sie Informationen über den Datensatz durch Setzen von `with_info=True` erhalten können. Hier bekommen wir die Größe des Datensatzes und die Namen der Kategorien. Leider gibt es nur ein "train"-Dataset und keines zum Testen oder Validieren, daher müssen wir den Trainingsdatensatz aufteilen. Das TF-Datasets-Projekt stellt dafür eine API zur Verfügung. Nehmen wir beispielsweise die ersten 10% des Datasets zum Testen, die nächsten 15% zum Validieren und die restlichen 75% zum Trainieren:

```
test_split, valid_split, train_split = tfds.Split.TRAIN.subsplit([10, 15, 75])

test_set = tfds.load("tf_flowers", split=test_split, as_supervised=True)
valid_set = tfds.load("tf_flowers", split=valid_split, as_supervised=True)
train_set = tfds.load("tf_flowers", split=train_split, as_supervised=True)
```

Als Nächstes müssen wir die Bilder vorverarbeiten. Das CNN erwartet Bilder mit 224×224 Pixeln, also müssen wir ihre Größe anpassen. Zudem müssen wir sie durch die Funktion `preprocess_input()` des Xception-Modells laufen lassen:

```
def preprocess(image, label):

    resized_image = tf.image.resize(image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)

    return final_image, label
```

Wenden Sie nun diese Vorverarbeitungsfunktion auf alle drei Datasets an, durchmischen Sie den

Trainingsdatensatz und fügen Sie Batching und Prefetching für alle drei hinzu:

```
batch_size = 32

train_set = train_set.shuffle(1000)

train_set = train_set.map(preprocess).batch(batch_size).prefetch(1)

valid_set = valid_set.map(preprocess).batch(batch_size).prefetch(1)

test_set = test_set.map(preprocess).batch(batch_size).prefetch(1)
```

Wollen Sie etwas Data Augmentation durchführen, ändern Sie die Vorverarbeitungsfunktion für den Trainingsdatensatz und fügen ein paar zufällige Transformationen zu den Trainingsbildern hinzu. Nutzen Sie beispielsweise `tf.image.random_crop()`, um die Bilder zufällig zu beschneiden, `tf.image.random_flip_left_right()`, um sie horizontal zu spiegeln und so weiter (siehe den Abschnitt »Pretrained Models for Transfer Learning« im Notebook).

Die Klasse `keras.preprocessing.image.ImageDataGenerator` macht es einfach, Bilder von der Festplatte zu laden und sie auf unterschiedlichste Weise anzureichern: Sie können jedes Bild verschieben, rotieren, umskalieren, horizontal oder vertikal spiegeln, verzerren oder beliebige andere Transformationsfunktionen anwenden. Das ist für einfache Projekte sehr bequem. Aber der Aufbau einer `tf.data`-Pipeline hat viele Vorteile – Sie können die Bilder effizient (zum Beispiel parallel) aus beliebigen Quellen und nicht nur von der Festplatte lesen, Sie können das Dataset nach Ihren Wünschen verändern, und wenn Sie eine Vorverarbeitungsfunktion schreiben, die auf `tf.image`-Operationen basiert, kann diese Funktion sowohl in der `tf.data`-Pipeline wie auch im in die Produktivumgebung deployten Modell genutzt werden (siehe [Kapitel 19](#)).

Als Nächstes wollen wir ein Xception-Modell laden, das mit ImageNet vortrainiert wurde. Wir schließen den oberen Teil des Netzes aus, indem wir `include_top=False` setzen – damit werden der Global Average Pooling Layer und die Dense-Ausgabeschicht weggelassen. Dann fügen wir unseren eigenen Globale Average Pooling Layer hinzu, der auf der Ausgabe des Basismodells basiert, gefolgt von einer Dense-Ausgabeschicht mit einer Einheit pro Klasse und der Softmax-Aktivierungsfunktion. Schließlich erstellen wir das Keras-Modell:

```
base_model = keras.applications.Xception(weights="imagenet",
                                             include_top=False)

avg = keras.layers.GlobalAveragePooling2D()(base_model.output)

output = keras.layers.Dense(n_classes, activation="softmax")(avg)

model = keras.Model(inputs=base_model.input, outputs=output)
```

Wie in [Kapitel 11](#) erläutert, ist es im Allgemeinen nicht verkehrt, die Gewichte der vortrainierten Schichten einzufrieren – zumindest zu Beginn des Trainings:

```
for layer in base_model.layers:
    layer.trainable = False
```



Da unser Modell die Schichten des Basismodells direkt verwendet und nicht das Objekt `base_model` selbst, würde es keine Auswirkungen haben, `base_model.trainable=False` zu setzen.

Schließlich können wir das Modell kompilieren und mit dem Training beginnen:

```
optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)

model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
               metrics=["accuracy"])

history = model.fit(train_set, epochs=5, validation_data=valid_set)
```



Das wird sehr langsam sein, sofern Sie keine GPU haben. Dann sollten Sie das Notebook dieses Kapitels in Colab mit einer GPU-Runtime ausführen (das ist kostenlos). Eine Anleitung dazu finden Sie unter <https://github.com/ageron/handson-ml2>.

Nachdem Sie das Modell für ein paar Epochen trainiert haben, sollte seine Validierungsgenauigkeit etwa 75 bis 80% erreicht haben und nicht mehr viel besser werden. Das heißt, dass die obersten Schichten nun ziemlich gut trainiert sind, wir alle Schichten wieder auftauen können (oder Sie könnten versuchen, nur die obersten aufzutauen) und mit dem Training fortfahren (vergessen Sie nicht, das Modell zu kompilieren, wenn Sie Schichten einfrieren oder auftauen). Dieses Mal nutzen wir eine viel niedrigere Lernrate, um zu verhindern, dass die vortrainierten Gewichte beschädigt werden:

```
for layer in base_model.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.001)

model.compile(...)

history = model.fit(...)
```

Es wird eine Weile dauern, aber dieses Modell sollte etwa 95% Genauigkeit beim Testdatensatz erreichen. Damit können Sie beginnen, erstaunliche Bildklassifizierer zu trainieren! Aber es gibt im Bereich Computer Vision noch viel mehr als nur Klassifikation. Wie wäre es beispielsweise, wenn Sie herausfinden, wo sich die Blume im Bild befindet? Darum kümmern wir uns jetzt.

Klassifikation und Lokalisierung

Das Lokalisieren eines Objekts in einem Bild kann als eine Regressionsaufgabe ausgedrückt werden (wie in [Kapitel 10](#) besprochen): Um eine Bounding Box (also einen Rahmen um das Objekt) vorherzusagen, sagt man häufig die horizontalen und vertikalen Koordinaten des

Objektmittelpunkts sowie dessen Höhe und Breite voraus. Wir haben also vier Zahlen, die vorherzusagen sind. Am Modell sind nicht viele Änderungen notwendig – wir müssen nur eine zweite Dense-Ausgabeschicht mit vier Einheiten hinzufügen (meist aufgesetzt auf den Global Average Pooling Layer), dann können wir sie mit dem MSE-Verlust trainieren:

```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                     include_top=False)

avg = keras.layers.GlobalAveragePooling2D()(base_model.output)

class_output = keras.layers.Dense(n_classes, activation="softmax")(avg)

loc_output = keras.layers.Dense(4)(avg)

model = keras.Model(inputs=base_model.input,
                     outputs=[class_output, loc_output])

model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2], # hängt davon ab, was wichtig ist
              optimizer=optimizer, metrics=["accuracy"])
```

Aber jetzt haben wir ein Problem. Der Flowers-Datensatz besitzt keine Bounding Boxes um die Blumen herum. Wir müssen sie also selbst hinzufügen. Das ist oft eine der schwersten und teuersten Aufgaben eines Machine-Learning-Projekts – die Labels erstellen. Es lohnt sich, nach den richtigen Tools Ausschau zu halten. Um Bilder mit Bounding Boxes zu versehen, können Sie ein Open-Source-Image-Labeling-Tool wie VGG Image Annotator, LabelImg, OpenLabeler oder ImgLab verwenden. Oder Sie greifen auf ein kommerzielles Tool wie LabelBox oder Supervisely zurück. Auch können Sie sich überlegen, Crowdsourcing-Plattformen wie Amazon Mechanical Turk einzusetzen, wenn Sie eine sehr große Menge von Bildern auszuzeichnen haben. Aber es ist auf jeden Fall oft viel Arbeit, eine Crowdsourcing-Plattform aufzusetzen, das Formular für die Worker zu erstellen, sie zu überwachen und sicherzustellen, dass die Qualität der von ihnen erzeugten Bounding Boxes gut ist. Überlegen Sie es sich also gut, ob das den Aufwand wert ist. Gibt es nur ein paar Tausend Bilder auszuzeichnen und haben Sie nicht vor, das regelmäßig zu tun, kann es sinnvoller sein, es selbst zu übernehmen. Adriana Kovashka et al. haben einen sehr nützlichen Artikel (<https://homl.info/crowd>) über das Crowdsourcing für Computer Vision geschrieben²⁴. Ich empfehle, diesen zu lesen, auch wenn Sie nicht vorhaben, Crowdsourcing einzusetzen.

Nehmen wir an, Sie haben die Bounding Boxes für jedes Bild im Flowers-Datensatz erzeugt (wir gehen hier davon aus, dass es eine einzelne Bounding Box pro Bild gibt). Dann müssen Sie ein Dataset erstellen, dessen Elemente Batches mit vorverarbeiteten Bildern zusammen mit ihren Kategorie-Labels und Bounding Boxes sind. Jedes Element sollte ein Tupel der Form (`images`, `(class_labels, bounding_boxes)`) sein. Dann sind Sie dazu bereit, Ihr Modell zu trainieren!

Die Bounding Boxes sollten normalisiert werden, sodass die horizontalen und vertikalen Koordinaten, aber auch Höhe und Breite, alle im Bereich von 0 bis 1 liegen. Zudem ist es üblich, die Wurzel von Höhe und Breite vorherzusagen, statt direkt Höhe und Breite anzugeben: So wird ein 10-Pixel-Fehler für eine große Bounding Box nicht so sehr bestraft wie ein 10-Pixel-Fehler für eine kleine Bounding Box.

Der MSE funktioniert als Kostenfunktion zum Trainieren des Modells oft ziemlich gut, aber er ist keine gute Metrik, um auszuwerten, wie gut das Modell Bounding Boxes vorhersagen kann. Die am häufigsten dafür eingesetzte Metrik ist die *Intersection over Union* (IoU): Dabei handelt es sich um die Fläche der Schnittmenge von vorhergesagter und anvisierter Bounding Box, geteilt durch deren Vereinigungsmenge (siehe Abbildung 14-23). In tf.keras ist sie implementiert durch die Klasse `tf.keras.metrics.MeanIoU`.



Abbildung 14-23: Intersection-over-Union-(IoU)-Metrik für Bounding Boxes

Das Klassifizieren und Lokalisieren eines einzelnen Objekts ist zwar nett, was aber sollen wir tun, wenn die Bilder mehrere Objekte enthalten (wie das im Flowers-Datensatz oft der Fall ist)?

Objekterkennung

Die Aufgabe, mehrere Objekte in einem Bild zu klassifizieren und zu lokalisieren, nennt man *Objekterkennung*. Bis vor ein paar Jahren nahm man meist ein CNN, das darauf trainiert war, einzelne Objekte zu klassifizieren und zu lokalisieren, um es dann über das Bild gleiten zu lassen (siehe Abbildung 14-24). In diesem Beispiel wurde das Bild in ein 6×8 -Raster aufgeteilt, und wir sehen ein CNN (das dicke schwarze Rechteck), das über alle 3×3 -Regionen gleitet. Sucht es oben links im Bild, findet es einen Teil der linken Rose, die es wieder entdeckt, wenn es einen Schritt nach rechts bewegt wurde. Im nächsten Schritt erkennt es einen Teil der obersten Rose, die es ebenfalls wieder entdeckt, wenn es sich einen weiteren Schritt bewegt hat. Sie würden dann mit dem Verschieben des CNN über das ganze Bild fortfahren und alle 3×3 -Regionen

anschauen. Da Objekte aber auch unterschiedliche Größen haben können, würden Sie das CNN für Regionen unterschiedlicher Größe nutzen. Sind Sie beispielsweise mit den 3×3 -Regionen fertig, würden Sie mit 4×4 -Regionen fortfahren.

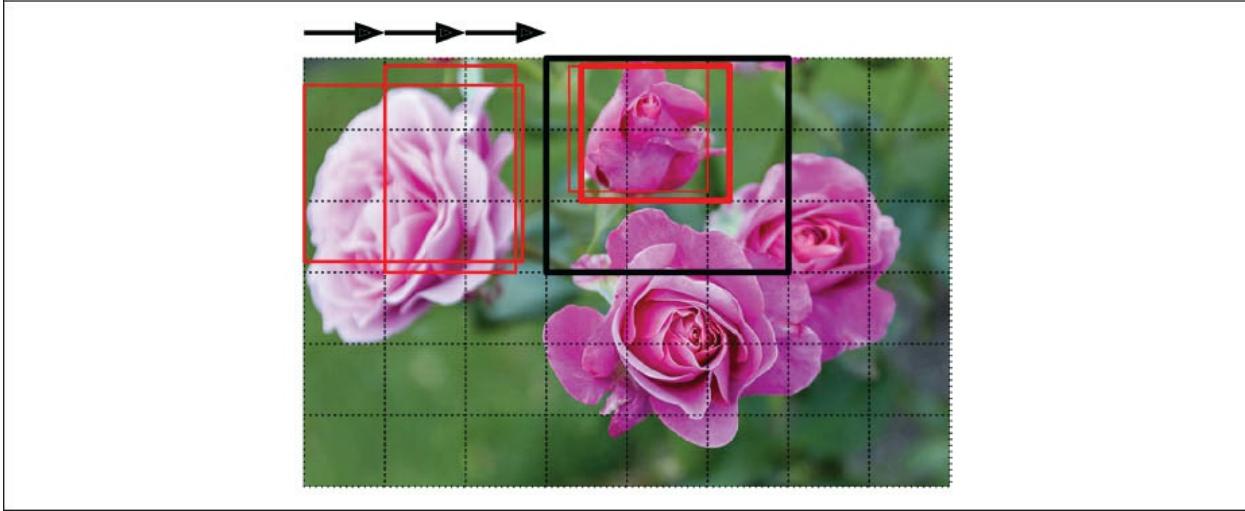


Abbildung 14-24: Mehrere Objekte durch das Verschieben eines CNN über ein Bild erkennen

Diese Technik ist ziemlich verständlich, aber wie Sie sehen, erkennt sie das gleiche Objekt mehrfach an etwas unterschiedlichen Positionen. Es ist etwas Nachverarbeitung notwendig, um all die überflüssigen Bounding Boxes wieder loszuwerden. Ein übliches Vorgehen dafür nennt sich *Non-Max Suppression*. Und so funktioniert es:

1. Zuerst müssen Sie Ihr CNN mit einer zusätzlichen *Objectness*-Ausgabe versehen, um die Wahrscheinlichkeit abzuschätzen, dass tatsächlich eine Blume im Bild vorhanden ist (alternativ könnten Sie eine Kategorie »Keine Blume« hinzufügen, aber das funktioniert meist nicht so gut). Sie muss die Sigmoid-Aktivierungsfunktion verwenden, und Sie können sie mithilfe des binären Kreuzentropie-Verlusts trainieren. Dann verwerfen Sie alle Bounding Boxes, für die der Objectness-Wert unter einem Grenzwert liegt – so werden Sie alle Bounding Boxes los, die gar keine Blume enthalten.
2. Finden Sie die Bounding Box mit dem höchsten Objectness-Wert und verwerfen Sie alle anderen Bounding Boxes, die sich stark mit ihr überlappen (zum Beispiel mit einem IoU größer als 60%). So ist beispielsweise die Bounding Box mit dem größten Objectness-Wert in [Abbildung 14-24](#) die mit dem dicken Rahmen um die oberste Rose (der Objectness-Wert wird durch die Dicke des Rahmens dargestellt). Die andere Bounding Box für die gleiche Rose überlappt stark mit der maximalen Bounding Box, daher löschen wir sie.
3. Wiederholen Sie Schritt 2, bis keine weiteren Bounding Boxes mehr zu verwerfen sind.

Dieser einfache Ansatz zur Objekterkennung funktioniert ziemlich gut, aber dafür muss das CNN viele Male laufen, daher ist er sehr langsam. Zum Glück gibt es eine viel schnellere Möglichkeit, ein CNN über Bilder laufen zu lassen – ein *Fully Convolutional Network* (FCN).

Fully Convolutional Networks

Die Idee hinter den FCNs wurde erstmals in einem Artikel (<https://homl.info/fcn>) aus dem Jahr

2015 von Jonathan Long et al.²⁵ zur semantischen Segmentierung vorgestellt (also der Aufgabe, jedes Pixel in einem Bild anhand der Kategorie des Objekts zu klassifizieren, zu dem es gehört). Die Autoren schrieben, dass Sie die Dense-Schichten oben in einem CNN durch Convolutional Layers ersetzen könnten. Um das zu verstehen, schauen wir uns ein Beispiel an. Stellen Sie sich vor, eine Dense-Schicht mit 200 Neuronen sitzt oben auf einem Convolutional Layer, der 100 Feature Maps ausgibt – jede mit einer Größe von 7×7 (das ist die Größe der Feature Map, nicht die Kernelgröße). Jedes Neuron berechnet eine gewichtete Summe aller $100 \times 7 \times 7$ Aktivierungen des Convolutional Layer (plus einen Bias-Term). Was passiert nun, wenn wir die Dense-Schicht durch einen Convolutional Layer mit 200 Filtern jeweils mit der Größe 7×7 und "valid"-Padding ersetzen? Diese Schicht wird 200 Feature Maps der Größe 1×1 ausgeben (da der Kernel genau die Größe der Eingabe-Feature-Maps besitzt und wir "valid"-Padding verwenden). Mit anderen Worten: Es werden 200 Zahlen ausgegeben – so wie die Dense-Schicht getan hat. Schauen Sie sich die von einem Convolutional Layer durchgeführten Berechnungen genauer an, werden Sie feststellen, dass diese Zahlen genau die sind, die auch die Dense-Schicht geliefert hat. Der einzige Unterschied ist, dass die Ausgabe der Dense-Schicht ein Tensor der Form [Batchgröße, 200] war, während der Convolutional Layer einen Tensor der Form [Batchgröße, 1, 1, 200] liefert.

- * Um eine Dense-Schicht in einen Convolutional Layer umzuwandeln, muss die Anzahl der Filter im Convolutional Layer gleich der Anzahl der Einheiten in der Dense-Schicht sein, die Filtergröße muss der Größe der Eingabe-Filter-Maps entsprechen, und Sie müssen "valid"-Padding verwenden. Die Schrittweite kann auf 1 oder größer gesetzt werden, wie wir gleich sehen werden.

Warum ist das wichtig? Nun, während eine Dense-Schicht eine bestimmte Eingabegröße erwartet (da es ein Gewicht pro Eingabmerkmal besitzt), verarbeitet ein Convolutional Layer problemlos Bilder beliebiger Größe²⁶ (allerdings erwartet er, dass seine Eingaben eine bestimmte Anzahl an Kanälen hat, da jeder Kernel einen anderen Satz Gewichte für jeden Eingabekanal besitzt). Weil ein FCN nur Convolutional Layers enthält (und Pooling Layers mit der gleichen Eigenschaft), kann es mit Bildern beliebiger Größe trainiert und ausgeführt werden!

Stellen Sie sich beispielsweise vor, wir hätten schon ein CNN zur Klassifikation und Lokalisation von Blumen trainiert. Das geschah mit Bildern mit 224×224 Pixeln und es liefert 10 Zahlen: Die Ausgaben 0 bis 4 sind durch die Softmax-Aktivierung gelaufen und geben die Kategorienwahrscheinlichkeiten aus (eine pro Kategorie); Ausgabe 5 durchläuft die logistische Aktivierungsfunktion und liefert den Objectness-Wert; die Ausgaben 6 bis 9 verwenden gar keine Aktivierungsfunktion, sie stehen für die Koordinaten der Bounding Box sowie deren Höhe und Breite. Wir können die Dense-Schichten dieses Modells jetzt in Convolutional Layers umwandeln. Tatsächlich müssen wir es nicht einmal neu trainieren – wir können einfach die Gewichte aus den Dense-Schichten in die Convolutional Layers kopieren! Alternativ könnten wir das CNN vor dem Training auch in ein FCN konvertiert haben.

Jetzt stellen Sie sich vor, der letzte Convolutional Layer vor der Ausgabeschicht (auch als Flaschenhalsschicht bezeichnet) gibt 7×7 -Feature Maps aus, wenn dem Netz ein 224×224 -Bild übergeben wird (siehe die linke Seite von Abbildung 14-25). Füttern wir das FCN mit einem 448

× 448-Bild (siehe die rechte Seite von Abbildung 14-25), wird die Flaschenhalsschicht nun Feature Maps der Größe 14×14 ausgeben.²⁷ Da die Dense-Ausgabeschicht durch einen Convolutional Layer mit 10 Filtern der Größe 7×7 , "valid"-Padding und Schrittweite 1 ersetzt wurde, wird die Ausgabe aus 10 Feature Maps bestehen, die jeweils die Größe 8×8 haben (denn $14 - 7 + 1 = 8$). Mit anderen Worten: Das FCN wird das gesamte Bild nur einmal verarbeiten und ein 8×8 -Raster zurückliefern, in dem jedes Feld 10 Zahlen enthält (5 Kategorienwahrscheinlichkeiten, 1 Objectness-Wert und 4 Koordinaten für die Bounding Box). Es ist genauso, als hätten wir das ursprüngliche CNN genutzt und es mit 8 Schritten pro Zeile und 8 Schritten pro Spalte über das Bild bewegt. Um sich das besser vorstellen zu können, gehen wir davon aus, dass das Bild in ein 14×14 -Raster zerteilt wurde und dann ein 7×7 -Fenster darübergleitet – es gibt dann $8 \times 8 = 64$ mögliche Positionen für das Fenster und damit 8×8 Vorhersagen. Aber das FCN ist *sehr viel* effizienter, da es sich das Bild nur einmal anschaut. Tatsächlich ist *You Only Look Once* (YOLO) der Name einer sehr beliebten Objekterkennungsarchitektur, die wir uns als Nächstes anschauen werden.

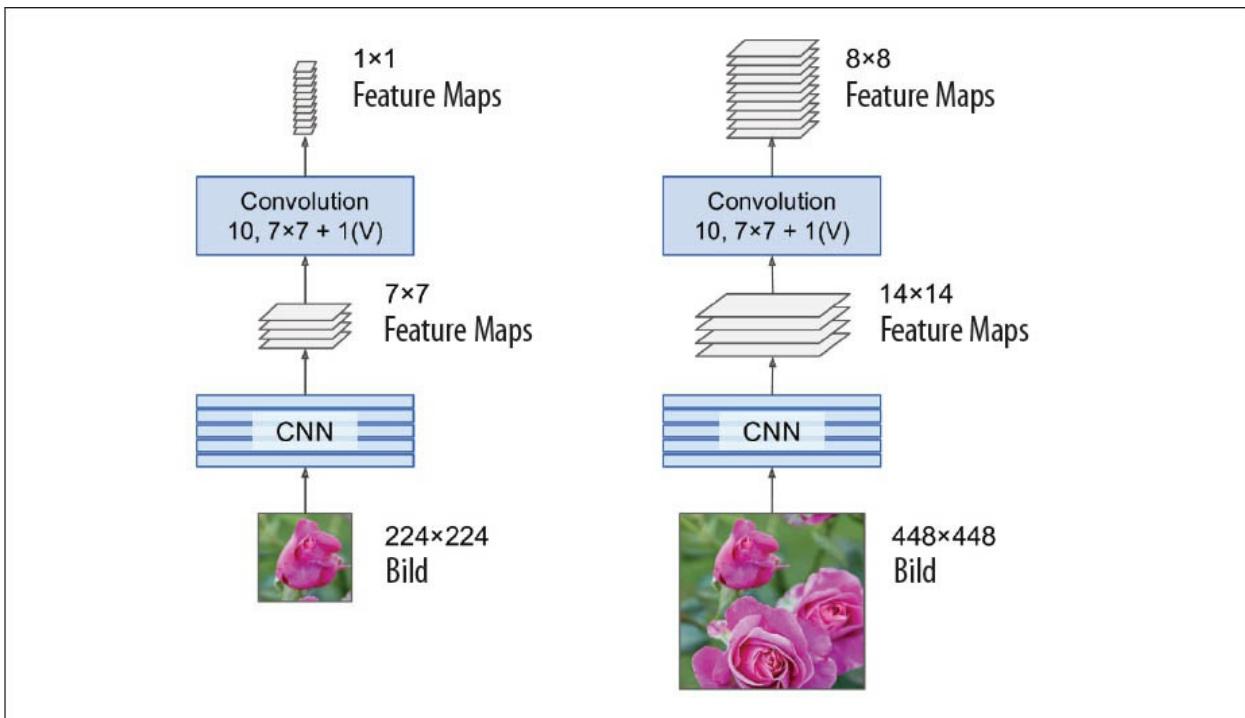


Abbildung 14-25: Das gleiche Fully Convolutional Network verarbeitet ein kleines Bild (links) und ein großes Bild (rechts).

You Only Look Once (YOLO)

YOLO ist eine außerordentlich schnelle und exakte Objekterkennungsarchitektur, die von Joseph Redmon et al. in einem Artikel (<https://homl.info/yolo>) aus dem Jahr 2015²⁸ vorgeschlagen und nachfolgend 2016 (<https://homl.info/yolo2>)²⁹ (YOLOv2) und 2018 (<https://homl.info/yolo3>)³⁰ (YOLOv3) verbessert wurde. Sie ist so schnell, dass sie in Echtzeit auf einem Video arbeiten kann, wie man in Redmons Demo (<https://homl.info/yolodemo>) sieht.

Die Architektur von YOLOv3 ähnelt der eben besprochenen, es gibt aber auch ein paar wichtige

Unterschiede:

- Sie gibt fünf Bounding Boxes für jedes Rasterfeld aus (statt nur eine), und jede Bounding Box bringt einen Objectness-Wert mit. Zudem gibt sie 20 Kategorienwahrscheinlichkeiten pro Rasterfeld aus, da sie mit dem PASCAL-VOC-Datensatz trainiert wurde, der 20 Kategorien besitzt. Da sind insgesamt 45 Zahlen pro Rasterfeld: 5 Bounding Boxes mit jeweils 4 Koordinaten, dazu 5 Objectness-Werte und 20 Kategorienwahrscheinlichkeiten.
- Statt die absoluten Koordinaten der Mittelpunkte der Bounding Boxes vorherzusagen, gibt YOLOv3 einen Offset relativ zu den Koordinaten des Rasterfelds aus, wobei $(0, 0)$ die obere linke und $(1, 1)$ die untere rechte Ecke des Felds sind. Für jedes Rasterfeld ist YOLOv3 darauf trainiert, nur Bounding Boxes vorherzusagen, deren Mittelpunkt in diesem Feld liegt (aber die Bounding Box selbst geht meist weit über das Rasterfeld hinaus). YOLOv3 wendet die logistische Aktivierungsfunktion auf die Koordinaten der Bounding Box an, um sicherzustellen, dass sie im Bereich zwischen 0 und 1 bleiben.
- Vor dem Training des neuronalen Netzes findet YOLOv3 fünf repräsentative Bounding-Box-Dimensionen, die als *Anchor Boxes* (oder *Bounding Box Priors*) bezeichnet werden. Dazu wendet es den K-Means-Algorithmus (siehe [Kapitel 9](#)) auf die Bounding Boxes des Trainingsdatensatzes an. Enthält das Trainingsbild beispielsweise viele Fußgänger, wird eine der Anchor Boxes sehr wahrscheinlich die Dimensionen eines typischen Fußgängers haben. Sagt dann das neuronale Netz fünf Bounding Boxes pro Rasterfeld voraus, sagt es tatsächlich voraus, wie viele jede der Anchor Boxes zu skalieren ist. Stellen Sie sich zum Beispiel vor, eine Anchor Box ist 100 Pixel hoch und 50 Pixel breit, und das Netzwerk sagt vielleicht einen vertikalen Skalierungsfaktor von 1,5 und einen horizontalen von 0,9 voraus (für eines der Rasterfelder). Das führt zu einer vorhergesagten Bounding Box mit 150×45 Pixeln. Genauer gesagt: Das Netz sagt den Logarithmus der vertikalen und horizontalen Skalierungsfaktoren voraus. Mit diesen Priors steigt die Wahrscheinlichkeit, dass das Netz Bounding Boxes mit passenden Dimensionen liefert, zudem wird das Training beschleunigt, weil das Netz schneller lernt, wie sinnvolle Bounding Boxes aussehen.
- Das Netz wird mit Bildern unterschiedlicher Größe trainiert: Alle paar Batches wählt es zufällig eine neue Größe (von 330×330 bis zu 608×608 Pixeln). Damit lernt das Netz, Objekte unterschiedlicher Skalierung zu erkennen. Zudem ist es so möglich, YOLOv3 mit unterschiedlichen Größen zu verwenden: Kleinere Bilder werden weniger genau, dafür aber schneller als große Bilder sein, und Sie können für Ihren Anwendungsfall selbst entscheiden, was Ihnen wichtiger ist.

Es gibt ein paar mehr Innovationen, an denen Sie vielleicht interessiert sind, wie zum Beispiel die Verwendung von Skip-Verbindungen, um einen Teil der räumlichen Auflösung wiederherzustellen, der im CNN verloren geht (wir werden dies gleich noch beim Blick auf die semantische Segmentierung behandeln). Im Artikel aus dem Jahr 2016 haben die Autoren das YOLO9000-Modell vorgestellt, das eine hierarchische Klassifikation verwendet: Es sagt eine Wahrscheinlichkeit für jeden Knoten in einer visuellen Hierarchie namens *WordTree* voraus. Damit wird es dem Modell ermöglicht, mit einer hohen Sicherheit vorherzusagen, ob ein Bild beispielsweise einen Hund enthält, auch wenn es nicht sicher ist, was für eine Rasse es ist.

Befassen Sie sich ruhig mit allen drei Artikeln: Sie sind gut lesbar und enthalten ausgezeichnete Beispiele dafür, wie Deep-Learning-Systeme schrittweise verbessert werden können.

Mean Average Precision (mAP)

Eine in der Objekterkennung sehr oft genutzte Metrik ist die *Mean Average Precision* (mAP). »Mean Average« klingt ein bisschen redundant, oder? Um diese Metrik zu verstehen, kehren wir zu den beiden Klassifikationsmetriken zurück, die wir in [Kapitel 3](#) behandelt haben: Relevanz und Sensitivität. Erinnern Sie sich an die Abhängigkeiten: Je höher die Relevanz, desto geringer ist die Sensitivität. Sie können dies in einer Relevanz-Sensitivitäts-Kurve sichtbar machen (siehe [Abbildung 3-5](#)). Um diese Kurve in einer einzigen Zahl zusammenzufassen, können wir die Fläche unter der Kurve berechnen (Area Under Curve, AUC). Aber bedenken Sie, dass die Relevanz-Sensitivitäts-Kurve ein paar Abschnitte besitzen kann, in denen die Relevanz tatsächlich mit wachsender Sensitivität steigt. Das kann insbesondere bei niedrigen Sensitivitätswerten der Fall sein (Sie sehen das oben links in [Abbildung 3-5](#)). Dies ist ein Teil der Motivation für die mAP-Metrik.

Angenommen, der Klassifizierer besitzt eine Relevanz von 90% bei einer Sensitivität von 10%, bei 20% Sensitivität aber 96%. Dann gibt es keinen Kompromiss, den man eingehen muss – es ist einfach sinnvoller, den Klassifizierer bei 20% Sensitivität zu verwenden, denn dann bekommen Sie gleichzeitig eine höhere Sensitivität und Relevanz. Statt sich also die Relevanz bei *genau* 10% Sensitivität anzuschauen, sollten wir nach der *maximalen* Relevanz suchen, die der Klassifizierer bei *mindestens* 10% Sensitivität bieten kann. Das wären 96%, nicht 90%. Daher ist eine Möglichkeit zum Abschätzen der Leistung des Modells, die maximale Relevanz zu berechnen, die Sie mit 0% Sensitivität, mit 10%, 20% und so weiter bis 100% bekommen können, und dann den Mittelwert dieser maximalen Relevanz zu ermitteln. Das ist die *Average-Precision-* (AP-)Metrik. Gibt es nun mehr als zwei Kategorien, können wir die AP für jede Kategorie und damit die Mean AP (mAP) berechnen. Das ist alles.

In einem System zur Objekterkennung gibt es eine zusätzliche Komplexitätsstufe: Was ist, wenn das System die korrekte Kategorie erkennt, aber die falsche Position (also die Bounding Box neben dem Objekt »gefunden« wird)? Sicherlich sollten wir das nicht als positive Vorhersage werten. Ein Ansatz ist, einen IoU-Grenzwert zu definieren: Wir können zum Beispiel überlegen, dass eine Vorhersage nur dann korrekt ist, wenn die IoU größer als (beispielsweise) 0,5 und die vorhergesagte Kategorie die richtige ist. Der zugehörige mAP wird im Allgemeinen als mAP@0.5 geschrieben (oder mAP@50%, manchmal auch einfach AP₅₀). In manchen Wettbewerben (wie der PASCAL VOC Challenge) wird das so gemacht. In anderen (wie der COCO Competition) wird der mAP für verschiedene IoU-Grenzwerte berechnet (0,50, 0,55, 0,60, ..., 0,95), und die abschließende Metrik ist der Mittelwert all dieser mAPs (geschrieben als mAP@[.50:.95] oder mAP[.50:0.05:.95]). Ja, das ist ein mittlerer mittlerer Mittelwert.

Eine Reihe von YOLO-Implementierungen, die mit TensorFlow gebaut wurden, finden sich auf GitHub. Schauen Sie sich insbesondere Zihao Zangs TensorFlow-2-Implementierung (<https://homl.info/yolotf2>) an. Andere Modelle zur Objekterkennung stehen im TensorFlow-Models-Projekt zur Verfügung, von denen viele vortrainierte Gewichte besitzen – manche wurden sogar nach TF Hub portiert, wie zum Beispiel SSD (<https://homl.info/ssd>)³¹ und Faster-RCNN (<https://homl.info/fasterrcnn>)³², die beide ziemlich beliebt sind. SSD ist zudem wie YOLO ein »Single Shot«-Erkennungsmodell. Faster R-CNN ist komplexer: Das Bild durchläuft zuerst ein CNN, die Ausgabe wird dann an ein *Region Proposal Network* (RPN) weitergereicht, das Bounding Boxes vorschlägt, die am wahrscheinlichsten ein Objekt enthalten. Schließlich läuft noch ein Klassifizierer für jede Bounding Box basierend auf der beschnittenen Ausgabe des CNN.

Die Wahl des Erkennungssystems hängt von vielen Faktoren ab: Geschwindigkeit, Genauigkeit, verfügbare vortrainierte Modelle, Trainingsdauer, Komplexität und so weiter. Die Artikel enthalten Tabellen mit Metriken, aber es gibt ziemlich viel Varianz in den Testumgebungen, und die Technologien entwickeln sich so schnell weiter, dass es schwierig ist, faire Vergleiche zu erstellen, die für die meisten Menschen nützlich sind und auch mehr als ein paar Monate gültig bleiben.

Wir können nun also Objekte lokalisieren, indem wir sie mit Bounding Boxes umgeben. Großartig! Aber vielleicht wollen Sie ein bisschen genauer sein. Schauen wir uns an, wie wir uns auf die Pixelebene begeben können.

Semantische Segmentierung

Bei der *semantischen Segmentierung* wird jedes Pixel aufgrund der Kategorie des Objekts klassifiziert, zu dem es gehört (zum Beispiel Straße, Auto, Fußgänger, Gebäude und so weiter), wie Sie in Abbildung 14-26 sehen können. Beachten Sie, dass verschiedene Objekte der gleichen Kategorie *nicht* unterschieden werden. So werden beispielsweise alle Räder auf der rechten Seite des segmentierten Bilds zu einem großen Pixelbereich. Das größte Problem bei dieser Aufgabe ist, dass Bilder beim Durchlaufen eines normalen CNN teilweise ihre räumliche Auflösung verlieren (aufgrund der Schichten mit Schrittweiten größer 1) – ein normales CNN findet daher eventuell heraus, dass irgendwo unten links im Bild eine Person ist, aber sehr viel genauer kann es nicht sein.

Wie bei der Objekterkennung gibt es viele verschiedene Ansätze, um dieses Problem anzugehen, von denen manche ziemlich komplex sind. Aber im Jahr 2015 wurde eine ziemlich einfache Lösung in dem Artikel von Jonathan Long et al. vorgeschlagen, den wir weiter oben schon erwähnt haben. Die Autoren beginnen mit einem vortrainierten CNN und verwandeln es in ein FCN. Das CNN wendet auf das Eingangsbild eine Gesamtschrittweite von 32 an (wenn Sie alle Schrittweiten größer 1 zusammenzählen), was bedeutet, dass die Ausgabe-Feature-Maps der letzten Schicht 32 Mal kleiner sind als das Eingabebild. Das ist natürlich zu grob, daher haben sie noch einen einzelnen *Upsampling Layer* hinzugefügt, der die Auflösung mit 32 multipliziert.



Abbildung 14-26: Semantische Segmentierung

Es gibt eine Reihe von Lösungsmöglichkeiten für das Upsampling (also das Vergrößern des Bilds), zum Beispiel bilineare Interpolation, aber das funktioniert nur bis zu einem Vergrößerungsfaktor von 4 oder 8 vernünftig. Stattdessen kommt ein *Transposed Convolutional Layer*³³ zum Einsatz: Dieser verhält sich so, als würde das Bild erst durch das Einfügen von leeren Zeilen und Spalten gestreckt und dann eine normale Convolution durchgeführt (siehe Abbildung 14-27). Alternativ stellen sich ihn manche als einen normalen Convolutional Layer vor, der gebrochene Schrittweiten einsetzt (zum Beispiel 1/2 in Abbildung 14-27). Der Transposed Convolutional Layer kann damit initialisiert werden, so etwas wie eine lineare Interpolation durchzuführen, aber da es sich um eine trainierbare Schicht handelt, wird er während des Trainings lernen, etwas Besseres zu liefern. In tf.keras können Sie die Schicht Conv2DTranspose verwenden.

- ▀ In einem Transposed Convolutional Layer definiert die Schrittweite, wie stark die Eingabe gestreckt wird, nicht die Größe der Filterschritte. Je größer also die Schrittweite, desto größer wird die Ausgabe (anders als bei Convolutional Layers oder Pooling Layers).

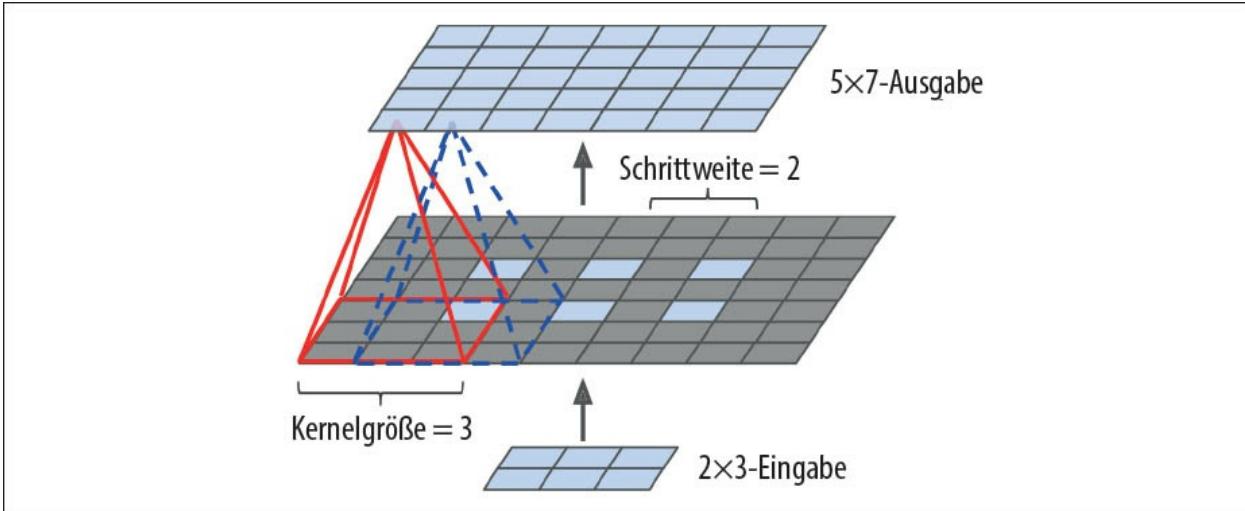


Abbildung 14-27: Upsampling mit einem Transposed Convolutional Layer

Convolution-Operationen in TensorFlow

TensorFlow bietet noch ein paar andere Arten von Convolutional Layers:

`keras.layers.Conv1D`

Erstellt einen Convolutional Layer für 1-D-Eingaben, wie zum Beispiel Zeitserien oder Text (Folgen von Buchstaben oder Wörtern), wie wir in [Kapitel 15](#) sehen werden.

`keras.layers.Conv3D`

Erstellt einen Convolutional Layer für 3-D-Eingaben, wie zum Beispiel dreidimensionalen PET-Scans.

`dilation_rate`

Setzen Sie den Hyperparameter `dilation_rate` eines Convolutional Layer auf einen Wert von 2 oder größer, erstellen Sie einen *À-Trous Convolutional Layer* (»à trous« ist französisch für »mit Löchern«). Das entspricht einem normalen Convolutional Layer mit einem Filter, der durch das Einfügen von Zeilen und Spalten mit Nullen (also Löchern) erweitert wird. So kann beispielsweise ein 1×3 -Filter (entsprechend `[[1, 2, 3]]`) durch eine *Dilation-Rate* von 4 so erweitert werden, dass er ein Dilated Filter mit `[[1, 0, 0, 2, 0, 0, 0, 3]]` wird. Damit bekommt der Convolutional Layer ein größeres Wahrnehmungsfeld ohne zusätzlichen Rechenaufwand und ohne zusätzliche Parameter.

`tf.nn.depthwise_conv2d()`

Kann genutzt werden, um einen *Depthwise Convolutional Layer* zu erstellen (aber Sie müssen die Variablen selbst anlegen). Es wird jeder Filter unabhängig auf jeden einzelnen Eingabekanal angewendet. Gibt es also f_n Filter und $f_{n'}$ Eingabekanäle, werden $f_n \times f_{n'}$ Feature Maps ausgegeben.

Diese Lösung ist okay, aber immer noch zu ungenau. Damit das besser funktioniert, haben die

Autoren Skip-Verbindungen von den unteren Schichten ergänzt: So wurde beispielsweise das Ausgabebild per Upsampling um einen Faktor 2 (statt 32) vergrößert und die Ausgabe einer unteren Schicht hinzugefügt, die diese doppelte Auflösung besitzt. Dann haben sie das Ergebnis mit einem Faktor von 16 upgesampelt, was zu einem Gesamt-Upsampling-Faktor von 32 führt (siehe Abbildung 14-28). Damit wurde ein Teil der räumlichen Auflösung wiederhergestellt, der in den früheren Pooling Layers verloren ging. In der besten Architektur haben die Autoren eine zweite, ähnliche Skip-Verbindung genutzt, um noch feinere Details von einer noch tieferen Schicht wiederherzustellen. Kurz gesagt, durchläuft die Ausgabe des ursprünglichen CNN die folgenden zusätzlichen Schritte: Upscale $\times 2$, Ausgabe einer tiefer liegenden Schicht (der passenden Größe) hinzufügen, Upscale $\times 2$, Ausgabe einer noch tiefer liegenden Schicht hinzufügen und schließlich ein Upscale $\times 8$. Es ist sogar möglich, die Auflösung über die Größe des ursprünglichen Bilds hinaus zu vergrößern – damit lässt sich die Auflösung eines Bilds erhöhen, was als *Super-Resolution* bezeichnet wird.

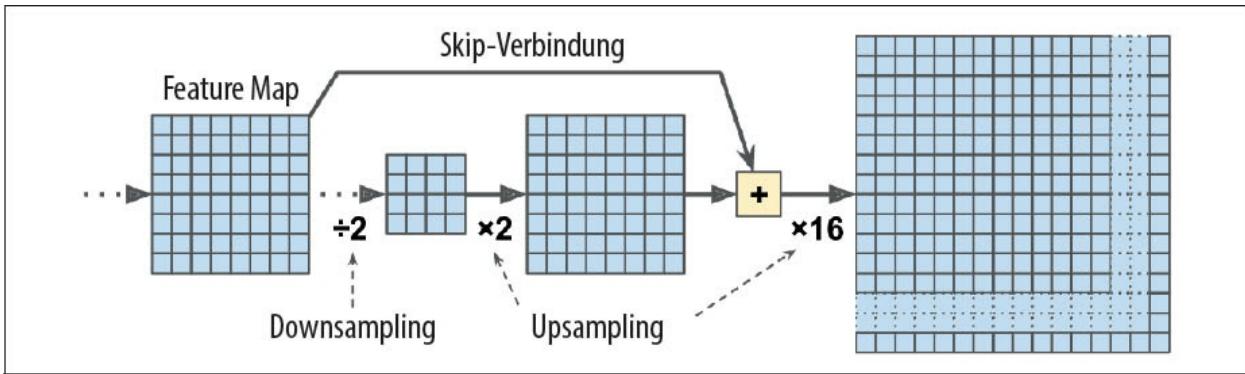


Abbildung 14-28: Skip Layers stellen einen Teil der räumlichen Auflösung aus tieferen Schichten wieder her.

Auch hier gibt es in vielen GitHub-Repositorien TensorFlow-Implementierungen zur semantischen Segmentierung (bisher aber nur TensorFlow 1), und Sie werden sogar vortrainierte *Instance-Segmentation*-Modelle im TensorFlow-Models-Projekt finden. Die Instance Segmentation ähnelt der semantischen Segmentierung, aber statt alle Objekte der gleichen Kategorie in einem großen Blob zusammenzufassen, wird jedes Objekt von den anderen unterschieden (so erkennt es beispielsweise jedes einzelne Fahrrad). Aktuell basieren die Instance-Segmentations-Modelle aus TensorFlow Models auf der *Mask-R-CNN*-Architektur, die in einem Artikel (<https://homl.info/maskrcnn>)³⁴ aus dem Jahr 2017 vorgestellt wurde: Es erweitert das Faster-R-CNN-Modell, indem es zusätzlich eine Pixelmaske für jede Bounding Box erzeugt. So erhalten Sie nicht nur eine Bounding Box für jedes Objekt und einen Satz von geschätzten Kategorienwahrscheinlichkeiten, sondern auch noch eine Pixelmaske, die Pixel in der Bounding Box lokalisiert, die zum Objekt gehören.

Wie Sie sehen, ist das Gebiet der Computer Vision umfangreich und entwickelt sich schnell weiter. Jedes Jahr entstehen neue Architekturen, die alle auf Convolutional Neural Networks basieren. Es ist erstaunlich, was für Fortschritte es in den letzten Jahren gegeben hat, und Forscher konzentrieren sich jetzt auf immer schwierigere Probleme, wie zum Beispiel *Adversarial Learning* (das versucht, das Netzwerk widerstandsfähiger gegen Bilder zu machen,

die versuchen, es auszutricksen), Explainability (verstehen, warum das Netz eine bestimmte Klassifikation vorgenommen hat), realistische *Bilderzeugung* (auf die wir in [Kapitel 17](#) wieder zurückkommen) und *Single-Shot Learning* (ein System, das ein Objekt erkennen kann, nachdem es dieses nur einmal gesehen hat). Manche erforschen sogar vollständig neue Architekturen, so wie Geoffrey Hints Capsule Networks (<https://homl.info/capsnet>)³⁵ (ich habe sie in ein paar Videos (<https://homl.info/capsnetvideos>) vorgestellt, wobei der zugehörige Code in einem Notebook vorhanden ist). Im nächsten Kapitel werden wir uns anschauen, wie wir sequenzielle Daten, wie zum Beispiel Zeitserien, mit rekurrenten neuronalen Netzen und Convolutional Neural Networks verarbeiten können.

Übungen

1. Was sind die Vorteile eines CNN gegenüber einem vollständig verbundenen DNN zur Bildklassifizierung?
2. Betrachten Sie ein aus drei Convolutional Layers zusammengesetztes CNN, wobei jeder Kernel die 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, ResNet, SENet und Xception zugrunde?
7. Was ist ein Fully Convolutional Network? Wie können Sie eine Dense-Schicht in einen Convolutional Layer umwandeln?
8. Was ist die größte technische Schwierigkeit bei der segmentischen Segmentierung?
9. Erstellen Sie Ihr eigenes CNN und versuchen Sie, die höchstmögliche Genauigkeit auf dem MNIST-Datensatz zu erzielen.
10. Transfer Learning zur Klassifikation großer Bilder – nutzen Sie folgende Schritte.
 - 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) oder einen bestehenden Datensatz (zum Beispiel ein TensorFlow-Dataset) klassifizieren.
 - b. Unterteilen Sie Ihren Datensatz in Trainings- und Testdaten.
 - c. Bauen Sie die Eingabepipeline einschließlich der passenden

Vorverarbeitungsoperationen und optional mit Data Augmentation.

d. Optimieren Sie ein vortrainiertes Modell mit diesem Datensatz.

11. Arbeiten Sie das Style Transfer Tutorial (<https://homl.info/styletuto>) von TensorFlow durch.

Es ist eine unterhaltsame Möglichkeit, mit Deep Learning Kunst zu erzeugen.

Lösungen zu diesen Übungen finden Sie in [Anhang A](#).

Verarbeiten von Sequenzen mit RNNs und CNNs

Der Schlagmann trifft den Ball. Der Outfielder fängt sofort an zu rennen und antizipiert die Flugbahn des Balls. Er verfolgt ihn, passt seine Bewegungen an und fängt 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 können (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, wie automatische Übersetzungen oder Sprache-zu-Text, sind.

In diesem Kapitel werden wir uns zunächst die grundlegenden Konzepte anschauen, auf denen RNNs aufbauen, und Sie werden erfahren, wie man sie per Backpropagation durch die Zeit trainiert. Dann werden wir sie einsetzen, um eine Zeitserie vorherzusagen. Danach werden wir die zwei größten Schwierigkeiten unter die Lupe nehmen, denen sich RNNs gegenübersehen:

- Instabile Gradienten (besprochen in [Kapitel 11](#)), die durch verschiedene Techniken abgemildert werden können – unter anderem Recurrent Dropout und Recurrent Layer Normalization.
- Ein (sehr) begrenztes Kurzzeitgedächtnis, das mithilfe von LSTM- und GRU-Zellen erweitert werden kann.

RNNs sind nicht die einzige Art neuronaler Netze, die sequenzielle Daten verarbeiten können. Für kleine Sequenzen kann auch ein normales Dense-Netzwerk ausreichen, und für sehr lange Sequenzen, wie zum Beispiel Audiosamples oder Text, können Convolutional Neural Networks ziemlich gut funktionieren. Wir werden beide Möglichkeiten besprechen und dieses Kapitel mit dem Implementieren eines *WaveNet* abschließen: Dabei handelt es sich um eine CNN-Architektur, die Sequenzen mit Zehntausenden Zeitschritten verarbeiten kann. In [Kapitel 16](#) werden wir die RNNs weiter fortführen und uns anschauen, wie man sie zur Verarbeitung natürlicher Sprache verwendet und welche aktuellen Architekturen es basierend auf Attention-Mechanismen gibt. Legen wir los!

Rekurrente Neuronen und Schichten

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 15-1](#) (links) gezeigt. Zu jedem Ausführungszeitpunkt t (auch *Frame* genannt) erhält dieses *rekurrente Neuron* die Eingaben $\mathbf{x}_{(t)}$ sowie seine eigene Ausgabe aus dem vorherigen Schritt $y_{(t-1)}$. Da es im ersten Zeitschritt keine vorherige Ausgabe gibt, wird sie im Allgemeinen auf 0 gesetzt. In [Abbildung 15-1](#) (rechts) stellen wir dieses winzige Netz entlang einer Zeitachse dar. Das nennt man *das Netz entlang der Zeitachse aufrollen* (es ist das gleiche Neuron, das einmal pro Zeitschritt dargestellt wird).

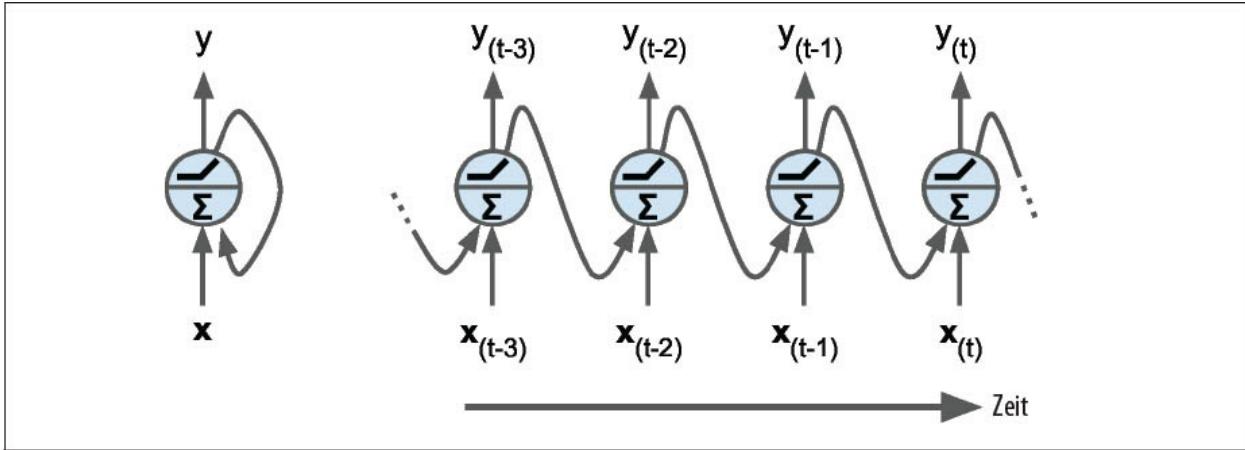


Abbildung 15-1: Ein rekurrentes Neuron (links), entlang der Zeitachse aufgerollt (rechts)

Sie können leicht eine ganze Schicht rekurrenter Neuronen erstellen. Bei jedem Schritt t erhält jedes Neuron sowohl den Eingabevektor $\mathbf{x}_{(t)}$ als auch den Ausgabevektor aus dem vorherigen Schritt $\mathbf{y}_{(t-1)}$, wie in [Abbildung 15-2](#) dargestellt. Sowohl die Ein- als auch die Ausgaben sind nun Vektoren (bei einem einzelnen Neuron war die Ausgabe noch ein Skalar).

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 . Betrachten wir die gesamte rekurrente Schicht statt nur ein rekurrentes Neuron, platzieren wir alle Gewichtsvektoren in zwei Gewichtsmatrizen \mathbf{W}_x und \mathbf{W}_y . Der Ausgabevektor der gesamten rekurrenten Schicht lässt sich wie in [Formel 15-1](#) gezeigt berechnen (\mathbf{b} ist der Bias-Term und $\phi(\cdot)$ ist die Aktivierungsfunktion, z.B. ReLU.¹).

Formel 15-1: Ausgabe eines einzelnen rekurrenten Neurons für einen einzelnen Datenpunkt

$$\mathbf{y}_{(t)} = \phi(\mathbf{W}_x^T \mathbf{x}_{(t)} + \mathbf{W}_y^T \mathbf{y}_{(t-1)} + \mathbf{b})$$

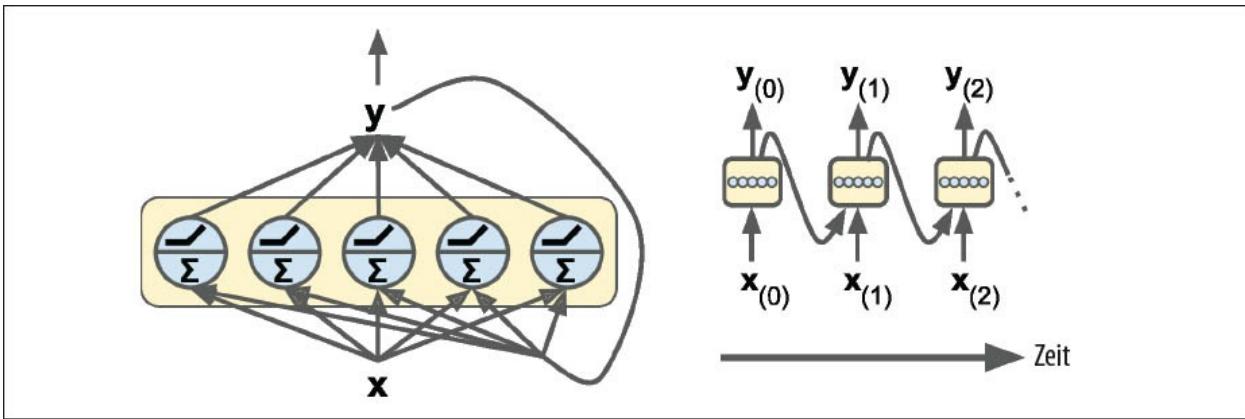


Abbildung 15-2: Eine Schicht rekurrenter Neuronen (links), entlang der Zeitachse aufgerollt (rechts)

Wie bei Feed-Forward-Netzen können wir die Ausgabe einer rekurrenten Schicht für ein ganzes Mini-Batch auf einen Schlag berechnen, indem wir alle Eingaben zum Zeitschritt t in eine Eingabematrix $\mathbf{X}_{(t)}$ stecken (siehe [Formel 15-2](#)).

Formel 15-2: Ausgabe einer Schicht rekurrenter Neuronen für alle Datenpunkte in einem Mini-Batch

$$\begin{aligned}\mathbf{Y}_{(t)} &= \phi(\mathbf{X}_{(t)} \mathbf{W}_x + \mathbf{Y}_{(t-1)} \mathbf{W}_y + \mathbf{b}) \\ &= \phi([\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}] \mathbf{W} + \mathbf{b}) \text{ mit } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}\end{aligned}$$

- $\mathbf{Y}_{(t)}$ 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 vorherigen Schritts ankommen.
- \mathbf{b} ist ein Vektor der Größe n_{Neuronen} mit den Bias-Termen jedes Neurons.
- 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 15-2](#)).
- Die Notation $[\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}]$ repräsentiert die horizontale Verkettung der Matrizen $\mathbf{X}_{(t)}$ und $\mathbf{Y}_{(t-1)}$.

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 $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \dots, \mathbf{X}_{(t)}$). Beim ersten Schritt $t = 0$ gibt es keine vorherigen Ausgaben, daher werden diese üblicherweise auf null gesetzt.

Gedächtniszellen

Da die Ausgabe eines rekurrenten Neurons zum Zeitpunkt t eine Funktion aller Eingaben der vorherigen 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, die nur kurze Muster lernen können (meist etwa zehn Schritte lang, aber das hängt von der Aufgabe ab). Wir werden uns im Verlauf dieses Kapitels aber noch komplexere und mächtigere Zellen ansehen, die längere Muster lernen können (etwa zehnmal länger, aber auch das hängt wieder von der Aufgabe ab).

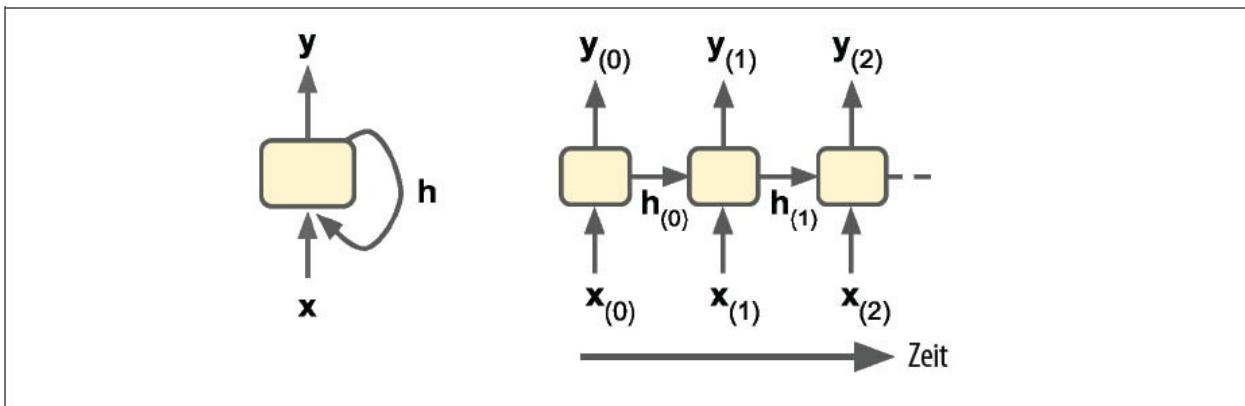


Abbildung 15-3: Der verborgene Zustand einer Zelle und ihre Ausgabe können unterschiedlich sein.

Allgemein ausgedrückt, ist der Zustand einer Zelle zum Zeitpunkt t , geschrieben $\mathbf{h}_{(t)}$ (das » h « steht für »hidden«), eine Funktion der Eingaben zu diesem Zeitpunkt und ihres Zustands zum vorherigen 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 vorherigen 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 15-3](#) zeigt.

Ein- und Ausgabesequenzen

Ein RNN kann gleichzeitig eine Sequenz von Eingabedaten aufnehmen und eine Sequenz von Ausgaben produzieren (siehe [Abbildung 15-4](#), Netz oben links). Diese Art von *Sequence-to-Sequence-Netz* ist 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 in [Abbildung 15-4](#)). Anders ausgedrückt, dieses Netz bildet 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 den gleichen Eingabevektor bei jedem Zeitschritt wieder und wieder zur Verfügung stellen und eine Sequenz ausgeben lassen (siehe das Netz unten links in Abbildung 15-4). Dies ist ein Vector-to-Sequence-Netz. Beispielsweise könnte die Eingabe ein Bild sein (oder die Ausgabe eines CNN) und die Ausgabe eine Beschreibung dieses Bilds.

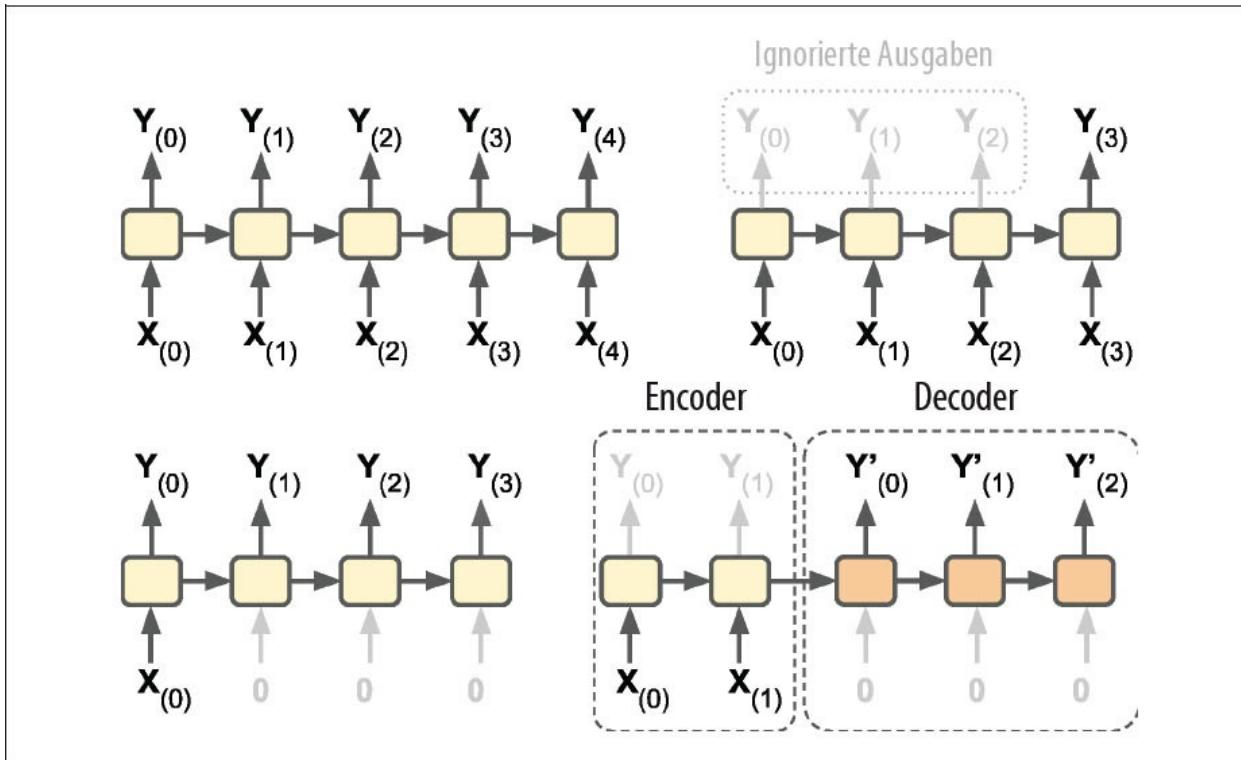


Abbildung 15-4: Sequence-to-Sequence (oben links), Sequence-to-Vector (oben rechts), Vector-to-Sequence (unten links), Encoder-Decoder (unten rechts)

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 in Abbildung 15-4). 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, und 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 Sequence-to-Sequence-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. Wir werden die Implementierung eines Encoder-Decoders in Kapitel 16 behandeln (wie sich zeigt, ist es ein bisschen komplexer, als in Abbildung 15-4 vorgeschlagen).

Dies klingt vielversprechend, aber wie trainieren Sie ein rekurrentes neuronales Netz?

RNNs trainieren

Um ein RNN zu trainieren, liegt das Geheimnis darin, es in der Zeit aufzurollen (so wie wir es gerade getan haben) und dann einfach eine normale Backpropagation einzusetzen (siehe Abbildung 15-5). Diese Strategie wird als *Backpropagation Through Time* (BPTT) bezeichnet.

Wie bei der normalen Backpropagation gibt es einen ersten Forward-Durchlauf durch das aufgerollte Netzwerk (dargestellt durch die gestrichelten Pfeile). Dann wird die Ausgabesequenz mit einer Kostenfunktion $C(Y_{(0)}, Y_{(1)}, \dots Y_{(T)})$ bewertet (wobei T der maximale Zeitschritt ist). Beachten Sie, dass diese Kostenfunktion manche Ausgaben ignorieren kann, wie in Abbildung 15-5 zu sehen (zum Beispiel werden in einem Sequence-to-Vector-RNN alle Ausgaben bis auf die allerletzte ignoriert). Die Gradienten dieser Kostenfunktion werden dann rückwärts durch das aufgerollte Netz propagiert (dargestellt durch die durchgehenden Pfeile). Schließlich werden die Modellparameter mit den während der BPTT berechneten Gradienten aktualisiert. Beachten Sie, dass die Gradienten rückwärts durch alle von der Kostenfunktion genutzten Ausgaben fließen und nicht nur durch die finale Ausgabe (so wird beispielsweise in Abbildung 15-5 die Kostenfunktion mit den letzten drei Ausgaben $Y_{(2)}$, $Y_{(3)}$ und $Y_{(4)}$ des Netzwerks berechnet, daher fließen die Gradienten durch diese drei Ausgaben, aber nicht durch $Y_{(0)}$ und $Y_{(1)}$). Und da die gleichen Parameter \mathbf{W} und \mathbf{b} bei jedem Zeitschritt verwendet werden, wird die Backpropagation das Richtige tun und über alle Zeitschritte aufsummieren.

Zum Glück kümmert sich tf.keras um die ganze Komplexität, also beginnen wir mit dem Programmieren!

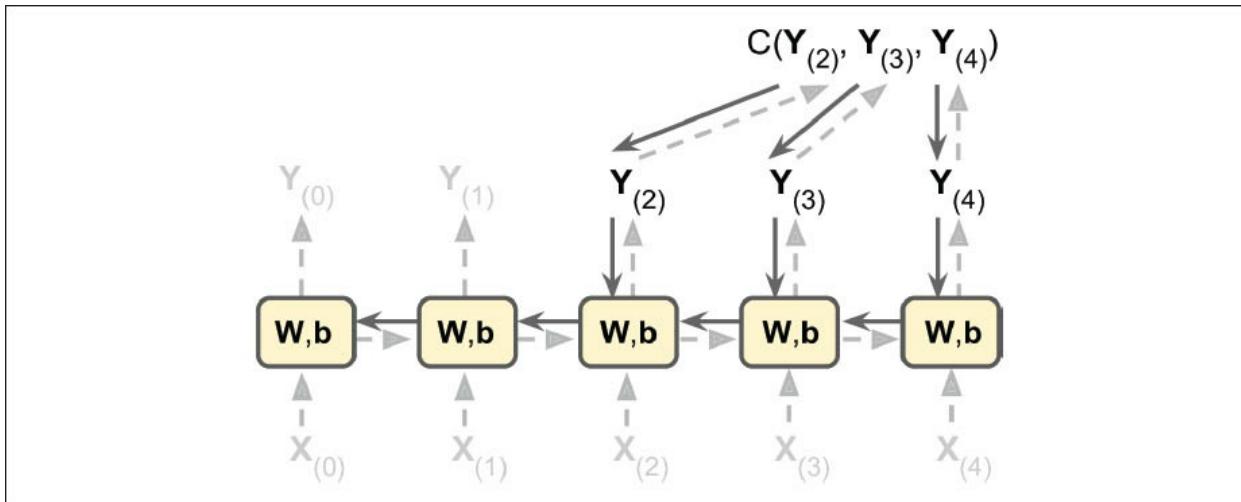


Abbildung 15-5: Backpropagation durch die Zeit

Eine Zeitserie vorhersagen

Stellen Sie sich vor, Sie untersuchen die Anzahl der aktiven Benutzer pro Stunde auf Ihrer Website, die tägliche Temperatur in Ihrer Stadt oder die finanzielle Situation Ihrer Firma – vierteljährlich mit verschiedenen Metriken gemessen. In all diesen Fällen handelt es sich bei den Daten um eine Sequenz eines oder mehrerer Werte pro Zeitschritt. Das wird als *Zeitserie*

bezeichnet. In den ersten beiden Beispielen gibt es einen einzelnen Wert pro Zeitschritt, daher handelt es sich um *univariate Zeitserien*, während es bei dem Finanzbeispiel mehrere Werte pro Zeitschritt gibt (zum Beispiel Umsatz, Verbindlichkeiten und so weiter), daher ist dies eine *multivariate Zeitserie*. Eine typische Aufgabe ist das Vorhersagen zukünftiger Werte, was als *Forecasting* bezeichnet wird. Eine andere häufige Aufgabe besteht darin, Lücken zu füllen – also fehlende Werte aus der Vergangenheit vorherzusagen (oder eher »hinterherzusagen«), was als *Imputation* bezeichnet wird. So zeigt Abbildung 15-6 beispielsweise drei univariate Zeitserien, die jeweils 50 Zeitschritte lang sind. Das Ziel ist hier, für jede der Serien den Wert für den nächsten Zeitschritt (dargestellt durch das X) vorherzusagen.

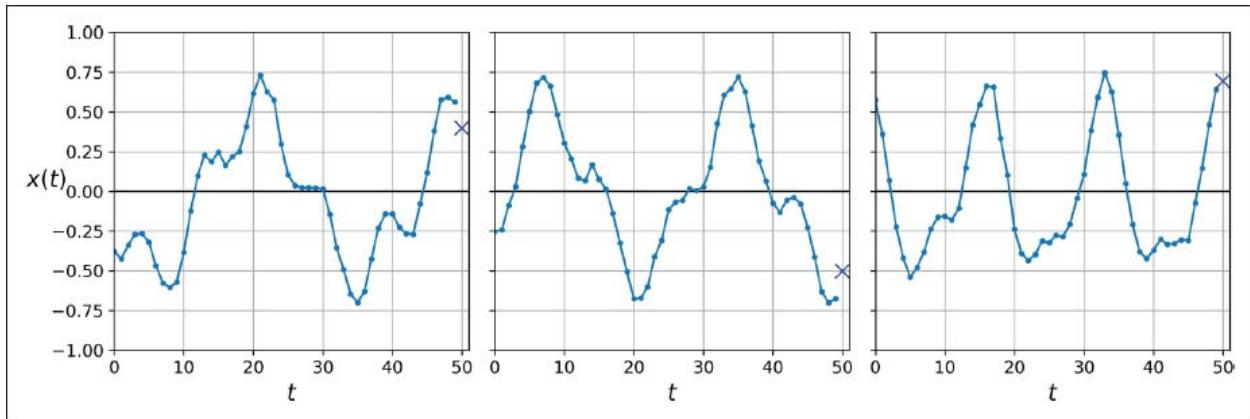


Abbildung 15-6: Zeitserien vorhersagen

Aus Gründen der Einfachheit nutzen wir eine Zeitserie, die durch die folgende Funktion `generate_time_series()` erzeugt wurde:

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)

    time = np.linspace(0, 1, n_steps)

    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10)) # Welle 1
    series += 0.2 * np.sin((time-offsets2) *(freq2 * 20 + 20))      # + Welle 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5)    # + Rauschen

    return series[..., np.newaxis].astype(np.float32)
```

Diese Funktion erstellt so viele Zeitserien wie angefordert (über das Argument `batch_size`) mit jeweils der Länge `n_steps`, und es gibt nur einen Wert pro Zeitschritt in jeder Serie (sie sind also alle univariat). Die Funktion gibt ein NumPy-Array der Form `[Batchgröße, Zeitschritte, 1]` zurück, wobei jede Serie die Summe von zwei Sinuswellen mit festen Amplituden, aber zufälligen Frequenzen und Phasen ist, ergänzt um ein bisschen Rauschen.

☞ Bei der Arbeit mit Zeitserien (und anderen Arten von Sequenzen, wie zum Beispiel Sätzen) werden die Eingangsmerkmale im Allgemeinen als 3-D-Arrays der Form `[Batchgröße,`

Zeitschritte, Dimensionalität] repräsentiert, wobei Dimensionalität für univariate Zeitserien 1 ist und für multivariate Zeitserien einen höheren Wert hat.

Erstellen wir jetzt einen Trainings-, einen Validierungs- und einen Testdatensatz mit dieser Funktion:

```
n_steps = 50  
series = generate_time_series(10000, n_steps + 1)  
  
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]  
  
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]  
  
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

X_train enthält 7.000 Zeitserien (seine Form ist also [7000, 50, 1]), während X_valid 2.000 (von der 7.000. bis zur 8.999. Zeitserie) und X_test 1.000 Zeitserien (von der 9.000. bis zur 9.999. Zeitserie) enthält. Da wir für jede Zeitserie einen einzelnen Wert vorhersagen wollen, sind die Ziele Spaltenvektoren (y_train hat beispielsweise die Form [7000, 1]).

Grundlegende Metriken

Bevor wir mit dem Einsatz von RNNs beginnen, ist es meist keine schlechte Idee, ein paar grundlegende Metriken zu haben, denn ansonsten denken wir, unser Modell funktioniere großartig, während es in Wirklichkeit schlechter arbeitet als einfache Modelle. So ist beispielsweise der einfachste Ansatz, den letzten Wert in jeder Serie vorherzusagen. Das nennt sich *naives Forecasting* und lässt sich manchmal nur erstaunlich schwer verbessern. In diesem Fall erhalten wir einen mittleren quadratischen Fehler von etwa 0,020:

```
>>> y_pred = X_valid[:, -1]  
>>> np.mean(keras.losses.mean_squared_error(y_valid, y_pred))  
0.020211367
```

Ein anderer einfacher Ansatz ist das Verwenden eines vollständig verbundenen Netzes. Da es eine flache Liste mit Merkmalen für jede Eingabe erwartet, müssen wir eine Flatten-Schicht hinzufügen. Nutzen wir ein einfaches lineares Regressionsmodell, sodass jede Vorhersage eine lineare Kombination der Werte in der Zeitserie ist:

```
model = keras.models.Sequential([  
    keras.layers.Flatten(input_shape=[50, 1]),  
    keras.layers.Dense(1)  
)
```

Kompilieren wir dieses Modell mit dem MSE-Verlust und dem Standard-Adam-Optimierer, fitten es mit dem Trainingsdatensatz für 20 Epochen und evaluieren es mit dem Validierungsdatensatz, erhalten wir einen MSE von etwa 0,004. Das ist viel besser als der naive Ansatz!

Ein einfaches RNN implementieren

Schauen wir, ob wir das mit einem einfachen RNN schlagen können:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

Das ist wirklich das einfachste RNN, das Sie bauen können. Es besteht nur aus einer einzigen Schicht mit einem einzigen Neuron, wie wir in [Abbildung 15-1](#) gesehen haben. Wir müssen (anders als im vorherigen Modell) die Länge der Eingabesequenzen nicht spezifizieren, da ein rekurrentes neuronales Netz eine beliebige Zahl von Zeitschritten verarbeiten kann (darum haben wir die erste Eingabedimension auf None gesetzt). Standardmäßig nutzt die SimpleRNN-Schicht als Aktivierungsfunktion den hyperbolischen Tangens. Sie funktioniert genau so, wie wir das zuvor gesehen haben: Der initiale Status $h_{(init)}$ wird auf 0 gesetzt und an ein einzelnes rekurrentes Neuron übergeben – zusammen mit dem Wert des ersten Zeitschritts $x_{(0)}$. Das Neuron berechnet eine gewichtete Summe dieser Werte und wendet die tanh-Aktivierungsfunktion auf das Ergebnis an, was zur ersten Ausgabe y_0 führt. In einem einfachen RNN ist diese Ausgabe zugleich der neue Status h_0 . Dieser neue Status wird zusammen mit dem nächsten Eingabewert $x_{(1)}$ an das gleiche rekurrente Neuron gegeben, und der Prozess wiederholt sich bis zum letzten Zeitschritt. Dann gibt die Schicht einfach den letzten Wert y_{49} aus. All dies wird genau so für jede Zeitserien durchgeführt.



Standardmäßig geben rekurrente Schichten in Keras nur das abschließende Ergebnis aus. Um eine Ausgabe pro Zeitschritt zu erreichen, müssen Sie `return_sequences=True` setzen, wie wir noch sehen werden.

Kompilieren, fitten und evaluieren Sie dieses Modell (wie zuvor trainieren wir für 20 Epochen mit Adam), werden Sie feststellen, dass sein MSE nur 0,014 erreicht. Es ist also besser als der naive Ansatz, schlägt aber kein einfaches lineares Modell. Beachten Sie, dass ein lineares Modell für jedes Neuron einen Parameter pro Eingabe und Zeitschritt hat, dazu ein Bias-Term (in dem von uns genutzten einfachen linearen Modell sind das zusammen 51 Parameter). Im Gegensatz dazu gibt es für jedes rekurrente Neuron in einem einfachen RNN nur einen Parameter pro Eingabe und Dimension des verborgenen Status (in einem einfachen RNN ist das lediglich die Anzahl rekurrenter Neuronen in der Schicht), dazu ein Bias-Term. In diesem einfachen RNN sind das insgesamt drei Parameter.

Trends und Saisonalität

Es gibt viele andere Modelle zum Vorhersagen von Zeitserien, wie zum Beispiel *Weighted-Moving-Average*-Modelle oder *Autoregressive-Integrated-Moving-Average*-(ARIMA-)Modelle. Für manche davon müssen Sie zuerst Trends und Saisonalität entfernen. Untersuchen Sie beispielsweise die Zahl der aktiven Benutzer auf Ihrer Website und wächst diese jeden Monat um 10%, müssten Sie diesen Trend aus der Zeitserie entfernen. Ist das Modell einmal trainiert und beginnt mit den Vorhersagen, müssten Sie den Trend wieder hinzufügen, um die endgültigen Vorhersagen zu erhalten. Oder wenn Sie versuchen, die Menge an Sonnencreme vorherzusagen, die jeden Monat verkauft wird, werden Sie vermutlich eine starke Saisonalität beobachten: Da sie sich jeden Sommer gut verkauft, wird jedes Jahr ein einfaches Muster wiederholt. Sie müssten diese Saisonalität aus der Zeitserie entfernen, zum Beispiel durch das Berechnen des Unterschieds des Werts bei jedem Zeitschritt zum Vorjahreswert (diese Technik wird als *Differencing* bezeichnet). Auch hier müssten Sie nach dem Trainieren des Modells beim Treffen von Vorhersagen das saisonale Muster erneut addieren, um die eigentlichen Vorhersagen zu erhalten.

Beim Einsatz von RNNs ist dieser Schritt im Allgemeinen nicht notwendig, aber es kann manchmal zu Verbesserungen führen, da das Modell den Trend oder die Saisonalität nicht erlernen muss.

Unser einfaches RNN war offensichtlich nicht gut genug für eine gute Leistung. Versuchen wir daher, mehr rekurrente Schichten hinzuzufügen!

Deep RNNs

Häufig stapelt man mehrere Schichten mit Zellen, wie in [Abbildung 15-7](#) zu sehen. Damit erhalten Sie ein *Deep RNN*.

Das Implementieren eines Deep RNN mit tf.keras ist ziemlich einfach – Sie stapeln einfach rekurrente Schichten. In diesem Beispiel nutzen wir drei SimpleRNN-Schichten (aber wir könnten auch beliebige andere rekurrente Schichten nehmen, wie zum Beispiel eine LSTM- oder GRU-Schicht, auf die wir noch kommen werden):

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])
```

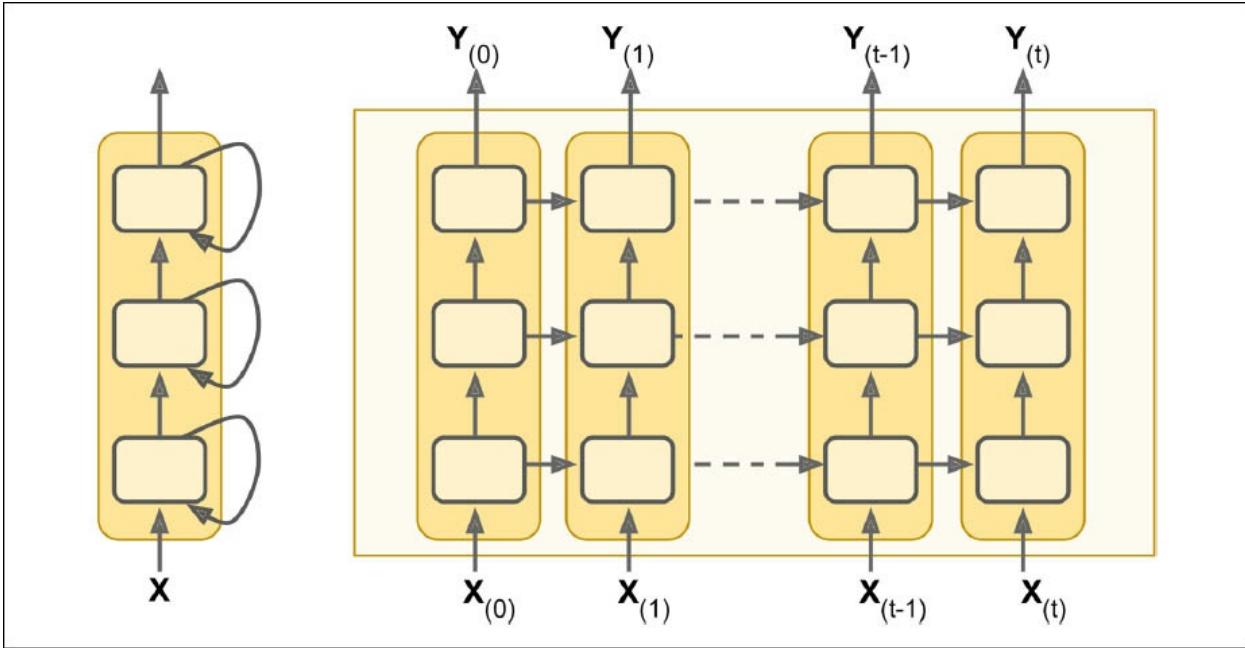


Abbildung 15-7: Deep RNN (links), in der Zeit aufgerollt (rechts)

- Achten Sie darauf, für alle rekurrente Schichten (außer der letzten, wenn Ihnen nur die letzte Ausgabe wichtig ist) `return_sequences=True` zu setzen. Wenn Sie das nicht tun, wird ein 2-D-Array (mit der Ausgabe des letzten Zeitschritts) statt eines 3-D-Arrays (mit den Ausgaben aller Zeitschritte) ausgegeben, und die nächste rekurrente Schicht wird sich beschweren, dass Sie ihr die Sequenzen nicht im erwarteten 3-D-Format anreichen.

Kompilieren, fitten und evaluieren Sie dieses Modell, werden Sie feststellen, dass ein MSE von 0,003 erreicht wird. Endlich haben wir das lineare Modell geschlagen!

Beachten Sie, dass die letzte Schicht nicht ideal ist: Sie muss eine einzelne Einheit haben, weil wir eine univariate Zeitserie vorhersagen wollen, was bedeutet, dass wir einen einzelnen Ausgabewert pro Zeitschritt haben müssen. Aber eine einzelne Einheit bedeutet, dass es sich beim verborgenen Status um nur eine einzelne Zahl handelt. Das ist nicht viel und vermutlich nicht so nützlich – sehr wahrscheinlich wird das RNN vor allem die verborgenen Status der anderen rekurrenten Schichten nutzen, um all die Informationen zu übertragen, die es von Zeitschritt zu Zeitschritt weiterreichen will, und der verborgene Status der letzten Schicht wird kaum zum Einsatz kommen. Und da eine `SimpleRNN`-Schicht standardmäßig die tanh-Aktivierungsfunktion nutzt, müssen die Vorhersagewerte im Bereich von -1 bis 1 liegen. Aber was ist, wenn Sie eine andere Aktivierungsfunktion nutzen wollen? Aus diesen beiden Gründen kann es sinnvoll sein, die Ausgabeschicht durch eine `Dense`-Schicht zu ersetzen: Sie würde etwas schneller laufen, die Genauigkeit wäre in etwa gleich, und sie würde es uns erlauben, eine beliebige Ausgabefunktion zu wählen. Nehmen Sie diese Änderung vor, sollten Sie auch darauf achten, `return_sequences=True` aus der zweiten (jetzt letzten) rekurrenten Schicht zu entfernen:

```
model = keras.models.Sequential([
    # ... (other layers)
])
```

```

    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)

])

```

Trainieren Sie dieses Modell, werden Sie sehen, dass es schneller konvergiert und eine genauso gute Leistung bringt. Zudem könnten Sie die Ausgabe-Aktivierungsfunktion ändern, wenn Sie das möchten.

Mehrere Zeitschritte vorhersagen

Bisher haben wir nur den Wert für den nächsten Zeitschritt vorhergesagt, aber wir könnten auch problemlos einen Wert mehrere Zeitschritte im Voraus vorhersagen, indem wir die Ziele entsprechend anpassen (um zum Beispiel den Wert zehn Schritte im Voraus vorherzusagen, ändern Sie einfach die Ziele auf den Wert zehn Schritte voraus statt nur einen Schritt voraus). Aber wie können wir alle nächsten zehn Werte vorhersagen?

Die eine Option ist, das schon trainierte Modell zu verwenden, es den nächsten Wert vorhersagen zu lassen, diesen Wert zu den Eingaben hinzuzufügen (und so zu tun, als wäre dieser vorhergesagte Wert tatsächlich eingetreten), das Modell erneut zu nutzen, um den folgenden Wert vorherzusagen, und so weiter:

```

series = generate_time_series(1, n_steps + 10)

X_new, Y_new = series[:, :n_steps], series[:, n_steps:]

X = X_new

for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[ :, np.newaxis, :]

    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]

```

Wie Sie sich vielleicht denken können, wird die Vorhersage für den nächsten Schritt im Allgemeinen genauer sein als für darauffolgende Zeitschritte, da sich die Fehler akkumulieren können (wie Sie in [Abbildung 15-8](#) sehen). Evaluieren Sie dieses Vorgehen mit dem Validierungsdatensatz, werden Sie einen MSE von etwa 0,029 erhalten. Das ist viel höher als bei den vorherigen Modellen, aber es ist auch eine viel schwierigere Aufgabe, daher ist dieser Vergleich nicht so sinnvoll. Es ist hilfreicher, diese Leistung mit naiven Vorhersagen (also nur vorherzusagen, dass die Zeitserie für zehn Zeitschritte konstant bleiben wird) oder mit einem einfachen linearen Modell zu vergleichen. Der naive Ansatz ist furchtbar (er liefert einen MSE

von etwa 0,223), aber das lineare Modell führt zu einem MSE von etwa 0,0188: Das ist viel besser als der Einsatz unseres RNN, um die Zukunft Schritt für Schritt vorherzusagen, außerdem lässt es sich viel schneller trainieren und ausführen. Aber wollen Sie bei komplexeren Aufgaben nur ein paar Schritte im Voraus vorhersagen, kann dieser Ansatz trotzdem gut funktionieren.

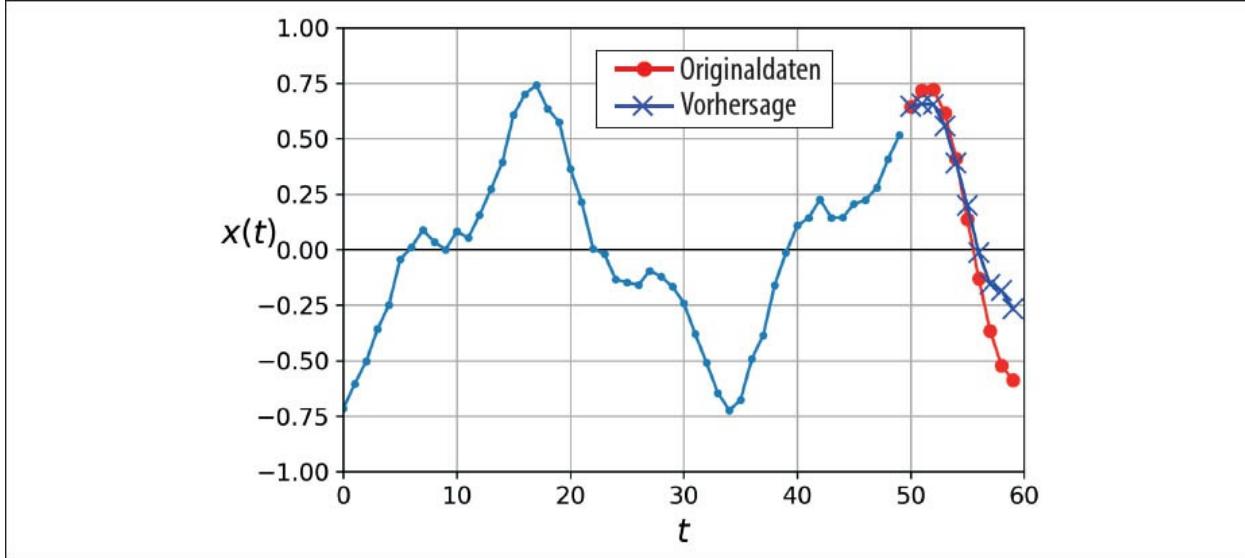


Abbildung 15-8: Zehn Schritte im Voraus vorhersagen, jeweils einen Schritt nach dem anderen

Die zweite Möglichkeit besteht darin, ein RNN so zu trainieren, dass es alle zehn nächsten Werte auf einmal vorhersagt. Wir können immer noch ein Sequence-to-Vector-Modell nutzen, aber es wird nicht nur einen, sondern zehn Werte ausgeben. Wir müssen jedoch zuerst die Ziele so ändern, dass es Vektoren mit den nächsten zehn Werten sind:

```
series = generate_time_series(10000, n_steps + 10)

X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

Jetzt müssen wir die Ausgabeschicht statt einer nun zehn Einheiten enthalten lassen:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
```

Nach dem Trainieren dieses Modells können Sie sehr einfach die nächsten zehn Werte auf

einmal vorhersagen lassen:

```
Y_pred = model.predict(X_new)
```

Dieses Modell funktioniert gut: Der MSE für die nächsten zehn Zeitschritte beträgt ungefähr 0,008. Das ist viel besser als das lineare Modell. Aber da geht noch mehr: Tatsächlich können wir das Modell nicht nur auf die Vorhersage der nächsten zehn Schritte im letzten Schritt trainieren, sondern bei jedem einzelnen Schritt. Mit anderen Worten – wir verwandeln dieses Sequence-to-Vector-RNN in ein Sequence-to-Sequence-RNN. Der Vorteil dieser Technik ist, dass der Verlust einen Term für die Ausgabe des RNN bei jedem einzelnen Schritt enthalten wird und nicht nur die Ausgabe beim letzten Zeitschritt. Das bedeutet, dass es viel mehr Fehlergradienten geben wird, die durch das Modell fließen, und sie müssen nicht nur durch die Zeit fließen – sie fließen auch von der Ausgabe jedes Zeitschritts. Das stabilisiert und beschleunigt das Training.

Zum Verständnis: Bei Zeitschritt 0 gibt das Modell einen Vektor mit der Vorhersage für die Zeitschritte 1 bis 10 aus, bei Zeitschritt 1 für die Zeitschritte 2 bis 11 und so weiter. So muss jedes Ziel eine Sequenz der gleichen Länge wie die der Eingabesequenz sein und einen zehndimensionalen Vektor in jedem Schritt enthalten. Bereiten wir diese Zielsequenzen vor:

```
Y = np.empty((10000, n_steps, 10)) # jedes Ziel ist eine Sequenz von 10-D-Vektoren

for step_ahead in range(1, 10 + 1):
    Y[:, :, step_ahead - 1] = series[:, step_ahead:step_ahead + n_steps, 0]

Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```

- Es mag Sie überraschen, dass die Ziele Werte enthalten, die in den Eingaben auftauchen (es gibt viel Überlappung zwischen X_train und Y_train). Ist das nicht geschummelt? Zum Glück nicht so richtig: Bei jedem Zeitschritt kennt das Modell nur die vergangenen Zeitschritte, daher kann es nicht vorausspicken. Das wird als *kausales* Modell bezeichnet.

Um das Modell in ein Sequence-to-Sequence-Modell umzuwandeln, müssen wir in allen rekurrenten Schichten `return_sequences=True` setzen (auch in der letzten) und die Dense-Ausgabeschicht bei jedem Zeitschritt anwenden. Keras bietet eine `Time Distributed`-Schicht genau dafür an: Sie verpackt eine Schicht (zum Beispiel eine Dense-Schicht) und wendet sie bei jedem Zeitschritt auf ihre Eingabesequenz an. Das tut sie effizient, indem sie die Form der Eingaben so anpasst, dass jeder Zeitschritt als eigene Instanz behandelt wird (sie wandelt die Eingabeform also von $[Batchgröße \times Zeitschritte, Eingabedimensionen]$ nach $[Batchgröße, Zeitschritte, Ausgabedimensionen]$ um; in diesem Beispiel ist die Anzahl der

Ausgabedimensionen 10, da die Dense-Schicht zehn Einheiten besitzt).² Dies ist das aktualisierte Modell:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Die Dense-Schicht unterstützt Sequenzen als Eingaben (und sogar höher dimensionale Eingaben): Sie behandelt sie wie TimeDistributed(Dense(...)), sie wird also nur auf die letzte Eingabedimension angewendet (unabhängig über alle Zeitschritte). Somit könnten wir die letzte Schicht einfach durch Dense(10) ersetzen. Aus Gründen der Verständlichkeit werden wir aber weiterhin TimeDistributed(Dense(10)) verwenden, weil so klarer wird, dass die Dense-Schicht unabhängig bei jedem Zeitschritt angewendet wird und dass das Modell eine Sequenz und nicht nur einen einzelnen Vektor ausgibt.

Alle Ausgaben werden während des Trainings benötigt, aber nur die Ausgabe beim letzten Zeitschritt ist für die Vorhersagen und die Evaluierung notwendig. Wenn wir also beim Training auch auf den MSE über alle Ausgaben bauen, werden wir zur Evaluierung eine eigene Metrik einsetzen, um den MSE nur über die Ausgabe beim letzten Zeitschritt zu berechnen.

```
def last_time_step_mse(Y_true, Y_pred):
    return keras.metrics.mean_squared_error(Y_true[:, -1], Y_pred[:, -1])

optimizer = keras.optimizers.Adam(lr=0.01)

model.compile(loss="mse", optimizer=optimizer, metrics=[last_time_step_mse])
```

Wir erhalten einen Validierungs-MSE von etwa 0,006, was um 25% besser als das vorherige Modell ist. Sie können diesen Ansatz mit dem ersten kombinieren: Sagen Sie einfach die nächsten zehn Werte mit diesem RNN voraus, dann verbinden Sie diese Werte mit der Eingangszeitserie, nutzen das Modell erneut, um die nächsten zehn Werte vorherzusagen, und wiederholen den Prozess so lange, wie es für Sie erforderlich ist. Mit diesem Ansatz können Sie beliebig lange Sequenzen erzeugen. Langfristige Voraussagen mögen damit nicht sehr genau sein, aber es kann reichen, wenn es Ihr Ziel ist, originelle Musik oder Text zu erzeugen, wie wir in [Kapitel 16](#) sehen werden.

Beim Vorhersagen von Zeitserien ist es oft nützlich, Fehlerbalken zusammen mit Ihren Voraussagen zu erhalten. Eine effiziente Technik dafür ist MC-Drop-out, das in [Kapitel 11](#) vorgestellt wurde: Fügen Sie eine MC-Drop-out-Schicht in jeder Gedächtniszelle hinzu, die Teile der Eingaben und des verborgenen Status verwirft. Nach dem Training nutzen Sie zum



Vorhersagen einer neuen Zeitserie das Modell viele Male und berechnen Mittelwert und Standardabweichung der Vorhersagen bei jedem Zeitschritt.

Einfache RNNs können ziemlich gut beim Vorhersagen von Zeitserien oder anderen Arten von Sequenzen sein, aber sie funktionieren nicht gut mit langen Zeitserien oder langen Sequenzen. Schauen wir uns an, woran das liegt und was wir dagegen tun können.

Arbeit mit langen Sequenzen

Um ein RNN mit langen Sequenzen zu trainieren, müssen wir es über viele Zeitschritte laufen lassen und das aufgerollte RNN damit zu einem sehr tiefen Netz machen. Wie bei jedem Deep Neural Network kann es mit dem Problem instabiler Gradienten kämpfen, das in [Kapitel 11](#) behandelt wurde: Es kann ewig dauern, bis es trainiert ist, oder das Training ist instabil. Zudem wird ein RNN, wenn es lange Sequenzen verarbeitet, die ersten Eingaben der Sequenz teilweise vergessen. Schauen wir uns beide Probleme an und beginnen wir mit den instabilen Gradienten.

Gegen instabile Gradienten kämpfen

Viele der Tricks, die wir in Deep Networks genutzt haben, um das Problem der instabilen Gradienten abzumildern, können auch für RNNs eingesetzt werden: eine gute Parameterinitialisierung, schnellere Optimierer, Drop-out und so weiter. Aber nicht sättigende Aktivierungsfunktionen (zum Beispiel ReLU) helfen hier eventuell nicht so viel – tatsächlich führen sie möglicherweise sogar dazu, dass das RNN während des Trainings noch instabiler zu machen. Warum? Nun, stellen Sie sich vor, dass das Gradientenverfahren die Gewichte so aktualisiert, dass die Ausgaben beim ersten Zeitschritt leicht erhöht werden. Da die gleichen Gewichte bei jedem Zeitschritt zum Einsatz kommen, werden die Ausgaben beim zweiten Zeitschritt ebenfalls leicht erhöht, genauso beim dritten und so weiter, bis die Ausgaben explodieren – und eine nicht sättigende Aktivierungsfunktion verhindert das nicht. Sie können das Risiko verringern, indem Sie eine geringere Lernrate einsetzen, aber Sie können auch einfach eine sättigende Aktivierungsfunktion wie den hyperbolischen Tangens nutzen (das erklärt, warum er der Standard ist). Auf die gleiche Art und Weise können die Gradienten selbst durch die Decke gehen. Stellen Sie fest, dass das Training instabil ist, sollten Sie die Größe der Gradienten überwachen (zum Beispiel mithilfe von TensorBoard) und eventuell auf Gradient Clipping zurückgreifen.

Zudem kann die Batchnormalisierung bei RNNs nicht so effizient genutzt werden wie bei tiefen Feed-Forward-Netzen. Tatsächlich können Sie es zwischen den Zeitschritten gar nicht verwenden, sondern nur zwischen rekurrenten Schichten. Genauer gesagt, ist es technisch gesehen möglich, eine BN-Schicht zu einer Gedächtniszelle hinzuzufügen (wie wir gleich sehen werden), sodass sie bei jedem Zeitschritt angewendet wird (sowohl auf die Eingaben für diesen Zeitschritt wie auch auf den verborgenen Status des vorherigen Schritts). Aber es wird bei jedem Zeitschritt die gleiche BN-Schicht genutzt – mit den gleichen Parametern, unabhängig von der tatsächlichen Größe und dem Offset der Eingaben und des verborgenen Status. In der Praxis

führt das nicht zu guten Ergebnissen, wie es César Laurent et al. in einem Artikel (<https://homl.info/rmmnm>)³ aus dem Jahr 2015 gezeigt haben: Die Autoren stellten fest, dass BN nur dann etwas besser war, wenn es auf die Eingaben, aber nicht auf die verborgenen Status angewendet wurde. Anders gesagt: Es war etwas besser als nichts beim Anwenden auf rekurrente Schichten (also vertikal in Abbildung 15-7), aber nicht innerhalb der rekurrenten Schichten (also horizontal). In Keras erreichen Sie das einfach durch Hinzufügen einer Schicht BatchNormalization vor jeder rekurrenten Schicht. Erwarten Sie aber nicht zu viel davon.

Eine andere Form der Normalisierung funktioniert mit RNNs meist besser – die *Layer Normalization*. Diese Idee wurde von Jimmy Lei Ba et al. in einem Artikel (<https://homl.info/layernorm>)⁴ aus dem Jahr 2016 vorgestellt, und sie ähnelt der Batchnormalisierung, aber statt zwischen den Batchdimensionen zu normalisieren, normalisiert sie zwischen den Merkmalsdimensionen. Ein Vorteil ist, dass die erforderlichen Statistiken bei jedem Schritt unabhängig von jeder Instanz berechnet werden können. Das heißt auch, dass sie sich während des Trainings und beim Testen gleich verhält (im Gegensatz zu BN) und dass sie keine exponentiellen gleitenden Mittelwerte nutzen muss, um die Merkmalsstatistiken über alle Instanzen im Trainingsdatensatz abzuschätzen. Wie BN lernt die Layer Normalization einen Größenwert und einen Offset-Parameter für jede Eingabe. In einem RNN wird sie normalerweise direkt nach der linearen Kombination der Eingaben und der verborgenen Status eingesetzt.

Verwenden wir nun `tf.keras`, um die Layer Normalization innerhalb einer einfachen Gedächtniszelle zu implementieren. Dazu müssen wir eine eigene Gedächtniszelle definieren. Das ist wie eine normale Schicht, nur erwartet ihre Methode `call()` zwei Argumente: die `inputs` zum aktuellen Zeitschritt und die verborgenen `states` aus dem vorherigen Zeitschritt. Beachten Sie, dass es sich beim Argument `states` um eine Liste mit einem oder mehreren Tensoren handelt. Bei einer einfachen RNN-Zelle enthält sie nur einen einzelnen Tensor mit den Ausgaben des vorherigen Zeitschritts, aber andere Zellen können auch mehrere Status-Tensoren haben (so hat zum Beispiel eine `LSTMCell` einen langfristigen und einen kurzfristigen Status, wie wir bald sehen werden). Eine Zelle muss zudem die Attribute `state_size` und `output_size` besitzen. In einem einfachen RNN entsprechen beide der Anzahl an Einheiten. Der folgende Code implementiert eine eigene Gedächtniszelle, die sich wie eine `SimpleRNNCell` verhält, allerdings bei jedem Zeitschritt eine Layer Normalization anwendet:

```

        self.layer_norm = keras.layers.LayerNormalization()

        self.activation = keras.activations.get(activation)

    def call(self, inputs, states):

        outputs, new_states = self.simple_rnn_cell(inputs, states)

        norm_outputs = self.activation(self.layer_norm(outputs))

        return norm_outputs, [norm_outputs]

```

Der Code ist recht klar.⁵ Unsere Klasse `LNSimpleRNNCell` erbt von der Klasse `keras.layers.Layer`, so wie jede andere eigene Schicht. Der Konstruktor übernimmt die Anzahl an Einheiten und die gewünschte Aktivierungsfunktion, setzt die Attribute `state_size` und `output_size` und erstellt dann eine `SimpleRNNCell` ohne Aktivierungsfunktion (weil wir die Layer Normalization nach den linearen Operationen, aber vor der Aktivierungsfunktion durchführen wollen). Dann erstellt der Konstruktor die `LayerNormalization`-Schicht und schnappt sich schließlich die gewünschte Aktivierungsfunktion. Die Methode `call()` wendet zunächst die einfache RNN-Zelle an, die eine lineare Kombination der aktuellen Eingaben und des vorherigen verborgenen Status berechnet, dann gibt sie das Ergebnis zweimal zurück (in einer `SimpleRNNCell` entsprechen die Ausgaben einfach dem verborgenen Status – `new_states[0]` ist gleich `outputs`, daher können wir `new_states` im Rest der Methode `call()` problemlos ignorieren). Als Nächstes wendet die Methode `call()` die Layer Normalization an, gefolgt von der Aktivierungsfunktion. Schließlich gibt sie die Ausgabe zweimal zurück (einmal als Ausgaben, einmal als neuer verborgener Status). Um diese eigene Zelle einzusetzen, müssen wir nur eine `keras.layers.RNN`-Schicht erstellen und ihr eine Zellinstanz übergeben:

```

model = keras.models.Sequential([
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True,
                     input_shape=[None, 1]),
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

```

Genauso könnten Sie eine eigene Zelle erstellen, um einen Drop-out zwischen jedem Zeitschritt anzuwenden. Aber es gibt einen einfacheren Weg: Alle rekurrenten Schichten (außer `keras.layers.RNN`) und alle von Keras angebotenen Zellen besitzen die Hyperparameter `dropout` und `recurrent_dropout`: Der erste definiert die Drop-out-Rate, die auf die Eingaben anzuwenden ist (bei jedem Zeitschritt), die zweite die Drop-out-Rate für die verborgenen Status (auch bei jedem Zeitschritt). Sie müssen also keine eigene Zelle erstellen, um bei jedem Zeitschritt in einem RNN einen Drop-out anzuwenden.

Mit diesen Techniken können Sie das Problem der instabilen Gradienten abschwächen und ein RNN sehr viel effizienter trainieren. Schauen wir uns nun an, wie wir mit dem Problem des Kurzzeitgedächtnisses umgehen.

Das Problem des Kurzzeitgedächtnisses

Aufgrund der Transformationen, die die Daten auf ihrem Weg durch ein RNN durchlaufen, gehen bei jedem Zeitschritt manche Informationen verloren. Nach einiger Zeit enthält der Status des RNN keinerlei Spuren der ursprünglichen Eingaben mehr. Das kann ein Showstopper sein. Stellen Sie sich die Fischdame Dory⁶ vor, die versucht, einen langen Satz zu übersetzen – wenn sie ihn fertig gelesen hat, weiß sie nicht mehr, womit er begann. Um dieses Problem anzugehen, wurden diverse Zelltypen mit Langzeitgedächtnis vorgestellt. Sie haben sich als so erfolgreich erwiesen, dass die einfachen Zellen nicht mehr sehr häufig zum Einsatz kommen. Schauen wir uns zuerst die beliebteste dieser Langzeitgedächtniszellen an: die LSTM-Zelle.

LSTM-Zellen

Die *Long-Short-Term-Memory-* (LSTM)-Zelle wurde 1997 von Sepp Hochreiter und Jürgen Schmidhuber vorgestellt (<https://homl.info/93>)⁷ und von verschiedenen Forschern über die Jahre nach und nach verbessert, wie zum Beispiel von Alex Graves (<https://homl.info/graves>), Haşim Sak (<https://homl.info/94>)⁸ und Wojciech Zaremba (<https://homl.info/95>)⁹. Stellen Sie sich die LSTM-Zelle als Black Box vor, können Sie sie fast so wie eine einfache Zelle einsetzen, nur dass sie eine bessere Leistung zeigen wird. Das Training wird schneller konvergieren, und sie wird langfristige Abhängigkeiten in den Daten erkennen. In Keras können Sie einfach statt der SimpleRNN-Schicht eine LSTM-Schicht nutzen:

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Alternativ können Sie die allgemeinere Schicht `keras.layers.RNN` einsetzen und ihr eine `LSTMCell` als Argument mitgeben:

```
model = keras.models.Sequential([
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True,
                    input_shape=[None, 1]),
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
```

])

Die LSTM-Schicht nutzt allerdings eine optimierte Implementierung bei der Ausführung auf einer GPU (siehe [Kapitel 19](#)), daher ist es im Allgemeinen sinnvoller, diese zu verwenden (die RNN-Schicht ist vor allem dann nützlich, wenn Sie eigene Zellen definieren, wie wir es weiter oben getan haben).

Wie arbeitet eine LSTM-Zelle? Die Architektur einer LSTM-Zelle ist in [Abbildung 15-9](#) dargestellt.

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: $\mathbf{h}_{(t)}$ und $\mathbf{c}_{(t)}$ (»c« steht für »cell«). Sie können sich $\mathbf{h}_{(t)}$ als das Kurzzeitgedächtnis und $\mathbf{c}_{(t)}$ als das Langzeitgedächtnis vorstellen.

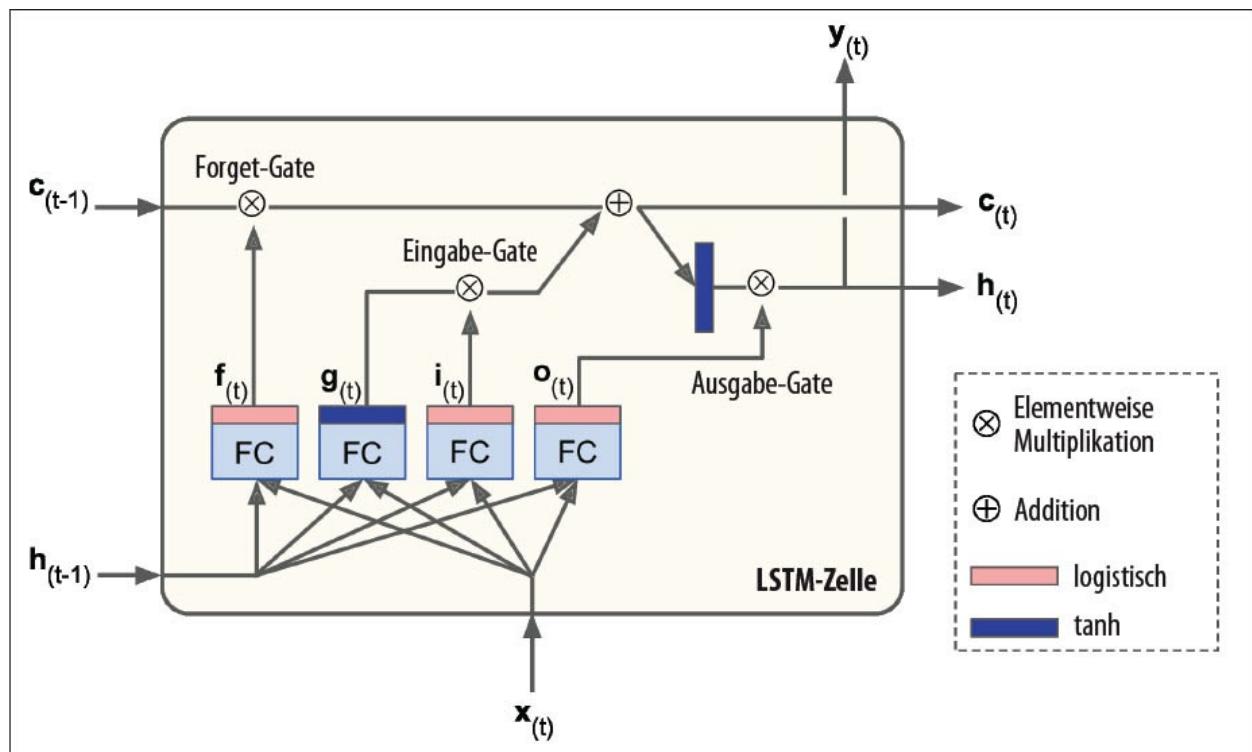


Abbildung 15-9: LSTM-Zelle

Öffnen wir nun die Black Box! 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 $\mathbf{c}_{(t-1)}$ 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 $\mathbf{c}_{(t)}$ 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 $\mathbf{h}_{(t)}$ (der die 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 $\mathbf{x}_{(t)}$ und der Zustand des Kurzzeitgedächtnisses $\mathbf{h}_{(t-1)}$ vier einzelnen vollständig verbundenen Schichten übergeben. Diese dienen unterschiedlichen Zwecken:

- Die wichtigste Schicht gibt $\mathbf{g}_{(t)}$ aus. Sie hat die gewöhnliche Aufgabe, die aktuellen Eingaben $\mathbf{x}_{(t)}$ und den vorherigen Zustand (des Kurzzeitgedächtnisses) $\mathbf{h}_{(t-1)}$ zu analysieren. In einer einfachen Zelle gibt es außer dieser Schicht nichts, ihre Ausgabe wird direkt an $\mathbf{y}_{(t)}$ und $\mathbf{h}_{(t)}$ übergeben. Im Gegensatz dazu wird die Ausgabe dieser Schicht in einer LSTM-Zelle nicht direkt nach außen geleitet. Stattdessen werden ihre wichtigsten Teile im Langzeitgedächtnis gespeichert (und der Rest wird verworfen).
- 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 $\mathbf{f}_{(t)}$ kontrolliert).
 - Das *Input Gate* steuert, welche Teile von $\mathbf{g}_{(t)}$ ins Langzeitgedächtnis übergehen sollen (von $\mathbf{i}_{(t)}$ kontrolliert).
 - Schließlich steuert das *Output Gate*, welche Teile des Langzeitgedächtnisses ausgelesen und bei diesem Schritt ausgegeben werden sollen – sowohl an $\mathbf{h}_{(t)}$ als auch $\mathbf{y}_{(t)}$ (von $\mathbf{o}_{(t)}$ 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 15-3](#) fasst zusammen, wie sich der Zustand des Langzeitgedächtnisses, des Kurzzeitgedächtnisses und die Ausgabe eines Zeitschritts für einen einzelnen Datenpunkt berechnen lassen (die Gleichungen für einen ganzen Mini-Batch sehen sehr ähnlich aus).

Formel 15-3: LSTM-Berechnungen

$$\mathbf{i}_{(t)} = \sigma(\mathbf{W}_{xi}^T \mathbf{x}_{(t)} + \mathbf{W}_{hi}^T \mathbf{h}_{(t-1)} + \mathbf{b}_i)$$

$$\mathbf{f}_{(t)} = \sigma(\mathbf{W}_{xf}^T \mathbf{x}_{(t)} + \mathbf{W}_{hf}^T \mathbf{h}_{(t-1)} + \mathbf{b}_f)$$

$$\mathbf{o}_{(t)} = \sigma(\mathbf{W}_{xo}^T \mathbf{x}_{(t)} + \mathbf{W}_{ho}^T \mathbf{h}_{(t-1)} + \mathbf{b}_o)$$

$$\mathbf{g}_{(t)} = \tanh(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \mathbf{h}_{(t-1)} + \mathbf{b}_g)$$

$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}$$

$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)})$$

- \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 normalen LSTM-Zelle betrachten die Gate-Controller nur die Eingabe $\mathbf{x}_{(t)}$ und den Zustand des Kurzzeitgedächtnisses aus dem vorherigen 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 vorgestellt (<https://homl.info/96>)¹⁰. Sie schlugen eine LSTM-Variante mit zusätzlichen Verbindungen vor, den *Peephole-Verbindungen*: Der Zustand des Langzeitgedächtnisses aus dem vorherigen Schritt $\mathbf{c}_{(t-1)}$ wird dabei den Eingaben des Forget Gate und des Input Gate hinzugefügt, und der aktuelle Zustand des Langzeitgedächtnisses $\mathbf{c}_{(t)}$ zur Eingabe des Output Gate. Das verbessert oft die Leistung, aber nicht immer, und es gibt keine klaren Muster dafür, welche Aufgaben besser mit oder ohne sie funktionieren – Sie müssen es bei Ihrer Aufgabe ausprobieren, um zu sehen, ob es hilft.

In Keras basiert die LSTM-Schicht auf der `keras.layers.LSTMCell`-Zelle, die keine Peepholes unterstützt. Aber die experimentelle `tf.keras.experimental.PeepholeLSTMCell` tut es, daher können Sie eine `keras.layers.RNN`-Schicht erzeugen und ihrem Konstruktor eine `PeepholeLSTMCell` übergeben.

Es gibt viele weitere Varianten der LSTM-Zelle. Eine besonders beliebte ist die GRU-Zelle, die wir uns als Nächstes anschauen werden.

GRU-Zellen

Die *Gated-Recurrent- Unit-(GRU-)*Zelle (siehe Abbildung 15-10) wurde von Kyunghyun Cho et al. in einem Artikel (<https://homl.info/97>) aus dem Jahr 2014 vorgeschlagen¹¹, in dem auch das oben erwähnte Encoder-Decoder-Netz erstmalig erwähnt wurde.

Eine GRU-Zelle ist eine vereinfachte Variante der LSTM-Zelle, die aber genauso gut zu funktionieren scheint¹² (was ihre wachsende Beliebtheit erklärt). Die wichtigsten Vereinfachungen sind:

- Beide Zustandsvektoren sind zu einem einzigen Vektor $\mathbf{h}(t)$ vereinigt.
- Ein einzelner Gate-Controller $\mathbf{z}(t)$ steuert sowohl das Forget Gate als auch das Input Gate. Falls der Gate-Controller eine 1 ausgibt, ist das Forget Gate offen ($= 1$) und das Input Gate geschlossen ($1 - 1 = 0$). 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 $r_{(t)}$, der steuert, welcher Teil des vorherigen Zustands der Hauptschicht zur Verfügung gestellt wird ($g_{(t)}$).

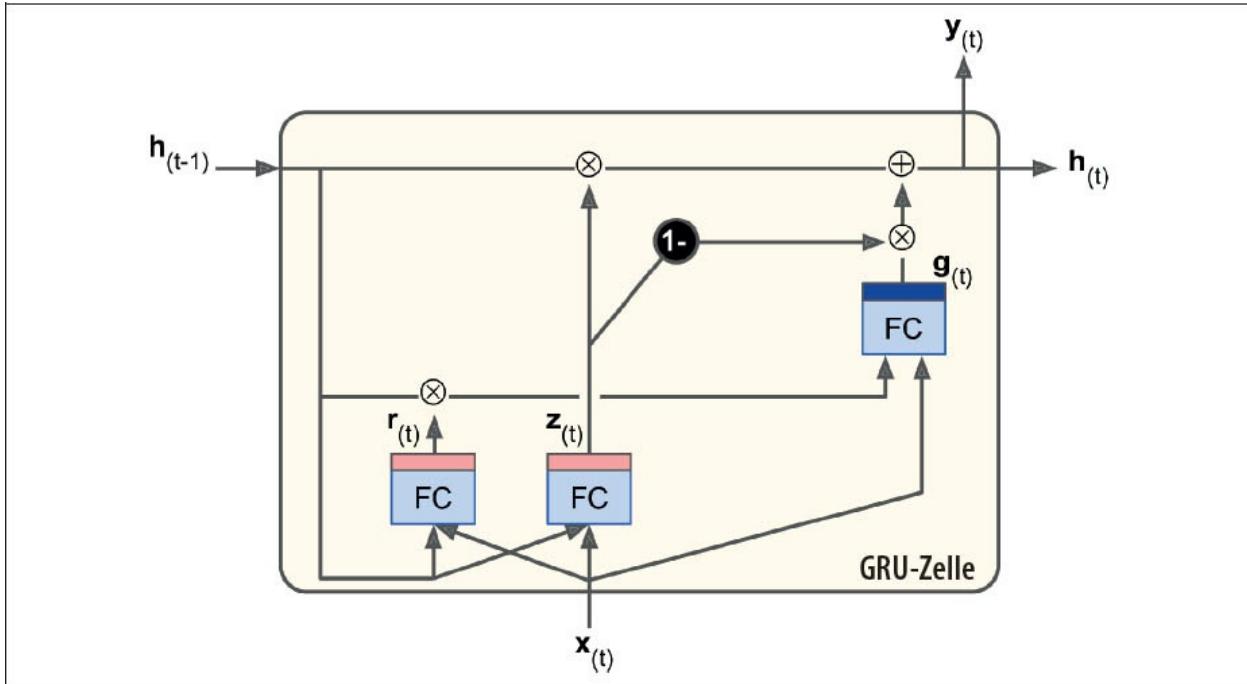


Abbildung 15-10: GRU-Zelle

[Formel 15-4](#) fasst zusammen, wie sich der Zustand der Zelle bei einem Schritt für einen einzelnen Datenpunkt berechnen lässt.

Formel 15-4: GRU-Berechnungen

$$z_{(t)} = \sigma(\mathbf{W}_{xz}^T x_{(t)} + \mathbf{W}_{hz}^T h_{(t-1)} + \mathbf{b}_z)$$

$$r_{(t)} = \sigma(\mathbf{W}_{xr}^T x_{(t)} + \mathbf{W}_{hr}^T h_{(t-1)} + \mathbf{b}_r)$$

$$g_{(t)} = \tanh(\mathbf{W}_{xg}^T x_{(t)} + \mathbf{W}_{hg}^T (r_{(t)} \otimes h_{(t-1)}) + \mathbf{b}_g)$$

$$h_{(t)} = z_{(t)} \otimes h_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)}$$

Keras bietet eine `keras.layers.GRU`-Schicht an (die auf der Gedächtniszelle `keras.layers.GRUCell` basiert). Um sie zu nutzen, müssen Sie nur `SimpleRNN` oder `LSTM` durch `GRU` ersetzen.

LSTM- und GRU-Zellen gehören zu den Hauptgründen für den Erfolg von RNNs. Während sie deutlich längere Sequenzen als einfache RNNs angehen können, haben sie aber immer noch ein recht begrenztes Kurzzeitgedächtnis, und es fällt ihnen schwer, langfristige Muster in Sequenzen mit 100 Zeitschritten oder mehr zu lernen, wie zum Beispiel bei Audiosamples, Langzeitserien oder langen Sätzen. Eine Möglichkeit, dies zu lösen, ist das Verkürzen der Eingabesequenzen,

zum Beispiel durch eindimensionale Convolutional Layers.

Eindimensionale Convolutional Layers zum Verarbeiten von Sequenzen nutzen

In [Kapitel 14](#) haben Sie gesehen, dass ein zweidimensionaler Convolutional Layer viele recht kleine Kernels (oder Filter) über ein Bild schiebt und damit viele zweidimensionale Feature Maps erzeugt (eine pro Kernel). Genauso schiebt ein eindimensionaler Convolutional Layer viele Kernels über eine Sequenz und erzeugt eine eindimensionale Feature Map pro Kernel. Jeder Kernel lernt, ein einzelnes, sehr kurzes sequenzielles Muster zu erkennen (nicht länger als die Kernelgröße). Verwenden Sie zehn Kernels, besteht die Ausgabe der Schicht aus zehn eindimensionalen Sequenzen (alle mit der gleichen Länge), oder Sie betrachten sie als einzelne zehndimensionale Sequenz. Das heißt, Sie können ein neuronales Netz aus einer Mischung von rekurrenten Schichten und eindimensionalen Convolutional Layers (oder sogar eindimensionalen Pooling Layers) bauen. Verwenden Sie einen eindimensionalen Convolutional Layer mit einer Schrittweite von 1 und "same"-Padding, wird die Ausgabesequenz die gleiche Länge wie die Eingabesequenz haben. Nutzen Sie aber "valid"-Padding oder eine Schrittweite größer als 1, wird die Ausgabesequenz kürzer als die Eingabesequenz sein – stellen Sie dann sicher, dass Sie die Ziele entsprechend anpassen. Das folgende Modell entspricht beispielsweise dem weiter oben genutzten, nur dass es mit einem eindimensionalen Convolutional Layer beginnt, der die Eingabefolge um einen Faktor von 2 herunterrechnet (durch eine Schrittweite von 2). Die Kernelgröße ist größer als die Schrittweite, sodass alle Eingaben zum Berechnen der Ausgabe der Schicht genutzt werden und das Modell damit lernen kann, die nützlichen Informationen zu behalten und die unnötigen Details zu verwerfen. Durch das Verkürzen der Sequenzen kann der Convolutional Layer eventuell dabei helfen, die GRU-Schichten längere Muster erkennen zu lassen. Beachten Sie, dass wir auch die ersten drei Zeitschritte in den Zielen verkürzen (da die Kernelgröße 4 ist, basiert die erste Ausgabe des Convolutional Layer auf den Eingabezeitschritten 0 bis 3) und die Ziele um einen Faktor von 2 herunterrechnen müssen:

```
model = keras.models.Sequential([
    keras.layers.Conv1D(filters=20, kernel_size=4, strides=2, padding="valid",
                        input_shape=[None, 1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train[:, 3::2], epochs=20,
                      validation_data=(X_valid, Y_valid[:, 3::2]))
```

Trainieren und evaluieren Sie dieses Modell, werden Sie feststellen, dass es das bisher beste Modell ist. Der Convolutional Layer hilft wirklich. Tatsächlich ist es sogar möglich, nur eindimensionale Convolutional Layers einzusetzen und die rekurrenten Schichten ganz zu verwerfen!

WaveNet

In einem Artikel (<https://homl.info/wavenet>)¹³ aus dem Jahr 2016 haben Aaron van der Oord und andere DeepMind-Forscher eine Architektur namens *WaveNet* vorgestellt. Sie haben eindimensionale Convolutional Layers gestapelt und dabei die Dilation Rate (wie weit die Eingaben der Neuronen voneinander entfernt sind) in jeder Schicht verdoppelt. Der erste Convolutional Layer erhält einen Einblick auf nur zwei Zeitschritte zugleich, während der nächste vier Zeitschritte sieht (sein Rezeptionsfeld ist vier Zeitschritte lang), der nächste acht Zeitschritte und so weiter (siehe Abbildung 15-11). So lernen die unteren Schichten kurzfristige Muster, während die höheren Schichten langfristige Muster lernen. Dank der doppelten Dilation Rate kann das Netz extrem lange Sequenzen sehr effizient verarbeiten.

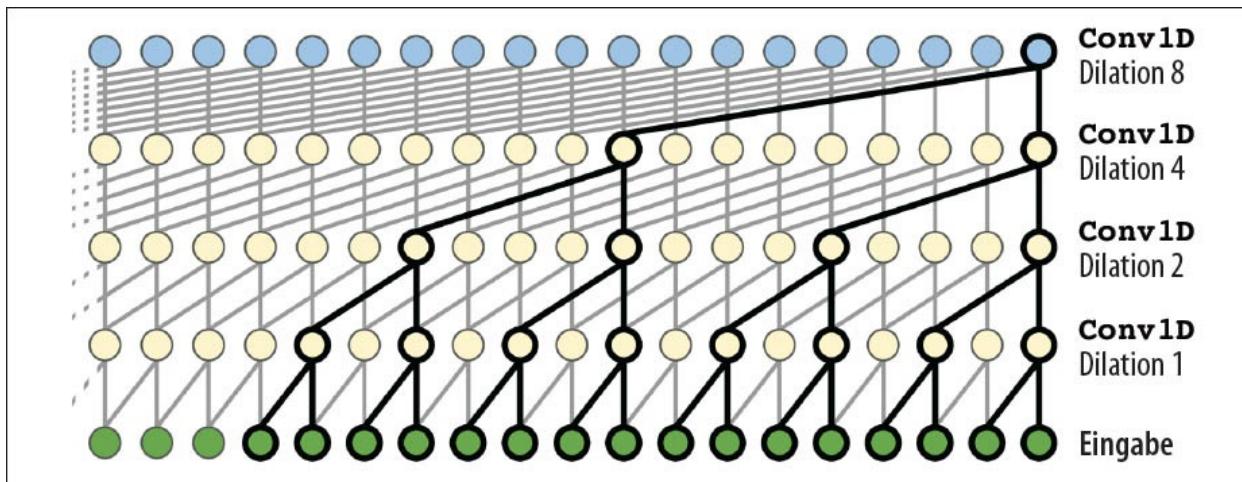


Abbildung 15-11: Architektur von WaveNet

Im WaveNet-Artikel haben die Autoren zehn Convolutional Layers gestapelt mit Dilatation Rates von 1, 2, 4, 8, ..., 256, 512, dann haben sie eine weitere Gruppe von zehn identischen Schichten gestapelt (wieder mit Dilatation Rates von 1, 2, 4, 8, ..., 256, 512) und nochmals eine identische Gruppe mit zehn Schichten. Diese Architektur haben sie damit begründet, dass sich ein einzelner Stack mit zehn Convolutional Layers mit diesen Dilatation Rates wie ein sehr effizienter Convolutional Layer mit einer Kernelgröße von 1024 verhält, nur dass er viel schneller und leistungsfähiger ist und deutlich weniger Parameter nutzt, weshalb sie drei solcher Blöcke gestapelt haben. Auch haben sie die Eingabesequenzen mit einer Reihe von Nullen, die der Dilatation Rate entspricht, vor jeder Schicht nach links gepaddet, um im ganzen Netz die gleiche Sequenzlänge zu erreichen. So kann man ein vereinfachtes WaveNet implementieren, das die gleichen Sequenzen wie zuvor verarbeiten:¹⁴

```
model = keras.models.Sequential()
```

```

model.add(keras.layers.InputLayer(input_shape=[None, 1]))

for rate in (1, 2, 4, 8) * 2:

    model.add(keras.layers.Conv1D(filters=20, kernel_size=2, padding="causal",
                                 activation="relu", dilation_rate=rate))

model.add(keras.layers.Conv1D(filters=10, kernel_size=1))

model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])

history = model.fit(X_train, Y_train, epochs=20,
                     validation_data=(X_valid, Y_valid))

```

Dieses Sequential-Modell beginnt mit einer expliziten Eingabeschicht (das ist einfacher, als zu versuchen, nur auf der ersten Schicht `input_shape` zu setzen), dann fährt es mit einem eindimensionalen Convolutional Layer fort, der "casual"-Padding verwendet: Damit wird sichergestellt, dass der Convolutional Layer bei seinen Vorhersagen nicht in die Zukunft spickt (es entspricht dem Padden der Eingaben mit der richtigen Menge an Nullen auf der linken Seite und dem Einsatz des "valid"- Padding). Dann fügen wir ähnliche Schichtenpaare mit steigenden Dilation Rates hinzu: 1, 2, 4, 8 und erneut 1, 2, 4, 8. Schließlich hängen wir noch die Ausgabeschicht an – ein Convolutional Layer mit zehn Filtern der Größe 1 und ohne Aktivierungsfunktion. Dank der Padding Layers gibt jeder Convolutional Layer eine Sequenz mit der gleichen Länge wie die der Eingabesequenzen aus, sodass die Ziele, die wir während des Trainings verwenden, die vollständigen Sequenzen sein können: Wir müssen sie nicht beschneiden oder herunterrechnen.

Die letzten beiden Modelle liefern die bisher beste Leistung beim Vorhersagen unserer Zeitserien! Im WaveNet-Artikel haben die Autoren State-of-the-Art-Leistungen für diverse Audioaufgaben erreicht (daher der Name der Architektur), Text-to-Speech-Aufgaben und das Erzeugen unglaublich realistischer Stimmen für verschiedene Sprachen eingeschlossen. Sie haben das Modell auch verwendet, um Musik zu erzeugen mit immer jeweils einem Audiosample. Das Ganze ist noch beeindruckender, wenn Sie sich klarmachen, dass eine einzige Sekunde Audiodaten Zehntausende Zeitschritte enthalten kann – selbst LSTMs und GRUs können nicht mit solch langen Sequenzen arbeiten.

In [Kapitel 16](#) werden wir uns weiter mit RNNs beschäftigen, und wir werden sehen, wie wir damit verschiedene Aufgaben im Bereich der natürlichen Sprache angehen können.

Übungen

1. Welche Anwendungen für ein Sequence-to-Sequence-RNN fallen Ihnen ein? Wie sieht es mit einem Sequence-to-Vector-RNN aus? Und einem Vector-to- Sequence-RNN?
2. Wie viele Dimensionen müssen die Eingaben für eine RNN-Schicht haben? Wofür steht jede Dimension? Wie ist es mit den Ausgaben?

3. Wenn Sie ein tiefes Sequence-to-Sequence-RNN bauen wollen – welche RNN-Schichten sollten `return_sequences=True` gesetzt haben? Wie ist es bei Sequence-to-Vector-RNNs?
4. Stellen Sie sich vor, Sie hätten eine univariate Zeitserie mit einem Eintrag pro Tag und wollten die nächsten sieben Tage vorhersagen. Welche RNN-Architektur sollten Sie verwenden?
5. Was sind die größten Schwierigkeiten, die beim Trainieren von RNNs auftreten können? Wie können Sie damit umgehen?
6. Können Sie die Architektur der LSTM-Zelle skizzieren?
7. Warum sollten Sie eindimensionale Convolutional Layers in einem RNN verwenden?
8. Welche Architektur für neuronale Netze könnten Sie zum Klassifizieren von Videos verwenden?
9. Trainieren Sie ein Klassifikationsmodell für den SketchRNN-Datensatz, der bei den TensorFlow-Datasets zur Verfügung steht.
10. Laden Sie das Bach Chorales Dataset (<https://homl.info/bach>) herunter und entpacken Sie es. Es besteht aus 382 Chorälen, die Johann Sebastian Bach komponiert hat. Jeder Choral ist 100 bis 640 Zeitschritte lang, und jeder Zeitschritt besteht aus vier Ganzzahlen, die jeweils einem Notenindex auf einem Klavier entsprechen (ausgenommen Wert 0, der bedeutet, dass keine Note gespielt wird). Trainieren Sie ein Modell – rekurrent, Convolutional oder beides –, das den nächsten Zeitschritt (vier Noten) vorhersagen kann, wenn es eine Sequenz von Zeitschritten für einen Choral erhält. Dann verwenden Sie dieses Modell, um Bach-ähnliche Musik zu erzeugen – jeweils eine Note nach der anderen: Sie können dazu dem Modell den Anfang eines Chorals mitgeben und es bitten, den nächsten Zeitschritt vorherzusagen, diesen dann an die Eingabesequenz anzuhängen, das Modell nach der nächsten Note zu fragen und so weiter. Schauen Sie sich auch Googles Coconet- Modell (<https://homl.info/coconet>) an, das für ein kleines Google-Doodle über Bach verwendet wurde.

Lösungen zu diesen Übungen finden Sie in [Anhang A](#).

Natürliche Sprachverarbeitung mit RNNs und Attention

Als sich Alan Turing 1950 seinen berühmten Turing-Test (<https://homl.info/turingtest>)¹ ausdachte, war es sein Ziel, die Fähigkeit einer Machine zu menschlicher Intelligenz zu bewerten. Er hätte viele Dinge testen können, wie zum Beispiel die Fähigkeit, Katzen auf Bildern zu erkennen, Schach zu spielen, Musik zu komponieren oder aus einem Labyrinth zu entkommen, aber interessanterweise entschied er sich für eine linguistische Aufgabe. Genauer gesagt, konzipierte er einen *Chatbot*, der seinen Gesprächspartner denken lässt, er sei ein Mensch.² Dieser Test hat seine Schwächen: Ein Satz hartcodierter Regeln kann arglose oder naive Menschen täuschen (zum Beispiel könnte die Maschine vage vordefinierte Antworten als Reaktion auf bestimmte Schlüsselwörter liefern, sie könnte so tun, als ob sie Spaß macht oder betrunken ist, um mit seltsamen Antworten durchzukommen, oder sie könnte schwierigen Fragen ausweichen, indem sie mit Gegenfragen reagiert), und viele Aspekte menschlicher Intelligenz werden vollständig ignoriert (zum Beispiel die Fähigkeit, nonverbale Kommunikation wie Gesichtsausdrücke zu interpretieren oder eine manuelle Aufgabe zu erlernen). Aber der Test hebt die Tatsache hervor, dass Beherrschung von Sprache unbestreitbar die größte kognitive Fähigkeit des *Homo sapiens* ist. Können wir eine Maschine bauen, die natürliche Sprache lesen und schreiben kann?

Ein oft verfolgter Ansatz für Aufgaben mit natürlicher Sprache ist die Verwendung rekurrenter neuronaler Netze. Daher werden wir uns weiter mit RNNs befassen (die in [Kapitel 15](#) vorgestellt wurden) und dabei mit einem *Character-RNN* beginnen, das darauf trainiert wird, den nächsten Buchstaben in einem Satz vorherzusagen. Damit können wir interessante Texte erzeugen, und dabei lernen wir, wie wir ein TensorFlow-Dataset für eine sehr lange Sequenz bauen. Zuerst werden wir ein *zustandsloses RNN* einsetzen (das bei jeder Iteration mit zufälligen Textabschnitten lernt und dabei keine Information über den Rest des Texts besitzt), dann werden wir ein *zustandsbehaftetes RNN* bauen (das den verborgenen Status zwischen Trainingsiterationen beibehält und dort mit dem Lesen fortfährt, wo es aufgehört hat, wodurch es längere Muster erlernen kann). Als Nächstes werden wir ein RNN zur Sentimentanalyse bauen (beispielsweise zum Lesen von Filmbewertungen und dem Ermitteln der Gefühle des Kritikers zum Film), wobei wir dieses Mal Sätze als Wortfolgen statt als Zeichenfolgen betrachten. Schließlich werden wir zeigen, wie RNNs zum Bauen einer Encoder-Decoder-Architektur genutzt werden können, die eine neuronale maschinelle Übersetzung (Neural Machine Translation, NMT) durchführen. Dazu werden wir die seq2seq-API des TensorFlow-Addons-Projekts einsetzen.

Im zweiten Teil dieses Kapitels werden wir uns *Attention-Mechanismen* anschauen. Es handelt

sich dabei um Komponenten neuronaler Netze, die lernen, den Teil der Eingaben auszuwählen, auf die sich das restliche Modell bei jedem Zeitschritt konzentrieren soll. Als Erstes erfahren wir, wie wir die Performance einer RNN-basierten Encoder-Decoder-Architektur mithilfe von Attention steigern, dann werden wir RNNs ganz fallen lassen und uns eine sehr erfolgreiche rein Attention-basierte Architektur namens *Transformer* anschauen. Schließlich werden wir uns noch mit den wichtigsten Fortschritten der Jahre 2018 und 2019 für NLP befassen, einschließlich der unglaublich leistungsfähigen Sprachmodelle GPT-2 und BERT, die beide auf Transformers beruhen.

Beginnen wir mit einem einfachen und netten Modell, das wie Shakespeare schreiben kann (nun, zumindest so ungefähr).

Shakespearesche Texte mit einem Character-RNN erzeugen

In einem berühmten Blogpost (<https://homl.info/charrnn>) aus dem Jahr 2015 mit dem Titel »The Unreasonable Effectiveness of Recurrent Neural Networks« hat Andrej Karpathy gezeigt, wie man ein RNN so trainiert, dass es das nächste Zeichen in einem Satz vorhersagt. Dieses *Char-RNN* kann eingesetzt werden, um Romantexte zu generieren – jeweils ein Zeichen nach dem anderen. Hier ein kleines Beispiel des Texts, der von einem Char-RNN erzeugt wurde, nachdem es mit dem vollständigen Werk von Shakespeare trainiert worden war:

PANDARUS:

Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Nicht wirklich ein Meisterwerk, aber trotzdem beeindruckend, dass das Modell dazu in der Lage war, Wörter, Grammatik, korrekte Satzzeichen und mehr zu erlernen – alles, indem es gelernt hat, das nächste Zeichen in einem Satz vorherzusagen. Schauen wir uns an, wie wir Schritt für Schritt ein Char-RNN bauen, und beginnen wir mit dem Erstellen des Datensatzes.

Den Trainingsdatensatz erstellen

Als Erstes wollen wir das gesamte Werk von Shakespeare herunterladen, wobei wir die praktische Keras-Funktion `get_file()` verwenden und die Daten aus Andres Karpathys Char-RNN-Projekt (<https://github.com/karpathy/char-rnn>) nutzen:

```
shakespeare_url = "https://homl.info/shakespeare" # Shortcut-URL  
  
filepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)  
  
with open(filepath) as f:  
  
    shakespeare_text = f.read()
```

Als Nächstes müssen wir jedes Zeichen als Integer codieren. Eine Möglichkeit besteht darin, wie

in [Kapitel 13](#) eine eigene Vorverarbeitungsschicht zu erstellen. Aber in diesem Fall wird es einfacher sein, die Keras-Klasse `Tokenizer` zu verwenden. Zuerst müssen wir einen Tokenizer an den Text anpassen: Er wird alle Zeichen aus dem Text finden und jedes davon auf eine andere Zeichen-ID abbilden – von 1 bis zur Anzahl der unterschiedlichen Zeichen (er beginnt nicht mit 0, daher können wir diesen Wert zum Maskieren verwenden, wie wir später noch sehen werden):

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)

tokenizer.fit_on_texts(shakespeare_text)
```

Wir setzen `char_level=True`, um eine Codierung auf Zeichenebene zu erhalten – standardmäßig wird auf Wortebene codiert. Beachten Sie, dass dieser Tokenizer den Text standardmäßig in Kleinbuchstaben umwandelt (aber Sie können `lower=False` setzen, wenn Sie das nicht wollen). Jetzt kann der Tokenizer einen Satz (oder eine Liste mit Sätzen) in eine Liste mit Zeichen-IDs und wieder zurück codieren, und wir erfahren, wie viele verschiedene Zeichen es gibt und wie viele Zeichen der Text enthält:

```
>>> tokenizer.texts_to_sequences(["First"])

[[20, 6, 9, 8, 3]]

>>> tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])

['f i r s t']

>>> max_id = len(tokenizer.word_index) # unterschiedliche Zeichen

>>> dataset_size = tokenizer.document_count # Gesamtzahl an Zeichen
```

Codieren wir den vollständigen Text, sodass jedes Zeichen durch seine ID repräsentiert wird (wir ziehen 1 ab, um IDs von 0 bis 38 statt von 1 bis 39 zu erhalten):

```
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

Bevor wir weitermachen, müssen wir den Datensatz in einen Trainingsdatensatz, einen Validierungsdatensatz und einen Testdatensatz unterteilen. Wir können nicht einfach alle Zeichen im Text durchmischen – wie also teilen wir einen sequenziellen Datensatz auf?

Wie ein sequenzieller Datensatz aufgeteilt wird

Es ist sehr wichtig, ein Überlappen zwischen Trainingsdatensatz, Validierungsdatensatz und Testdatensatz zu vermeiden. Wir können beispielsweise die ersten 90% des Texts als Trainingsdatensatz, die nächsten 5% als Validierungsdatensatz und die letzten 5% als Testdatensatz verwenden. Es wäre ebenfalls sinnvoll, eine Lücke zwischen diesen Datensätzen zu lassen, um zu vermeiden, dass ein Absatz über zwei Datensätze gespannt ist.

Beim Umgang mit Zeitserien würden Sie im Allgemeinen entlang der Zeitachse unterteilen: So

könnten Sie beispielsweise die Jahre 2000 bis 2012 für den Trainingsdatensatz, die Jahre 2013 bis 2015 für den Validierungsdatensatz und die Jahre 2016 bis 2018 für den Testdatensatz verwenden. Aber in manchen Fällen werden Sie auch entlang anderer Dimensionen unterteilen wollen, die Ihnen eine längere Zeitspanne zum Trainieren ermöglichen. Haben Sie zum Beispiel Daten über die finanzielle Lage von 10.000 Firmen für die Jahre 2000 bis 2018, wollen Sie vielleicht diese Daten für verschiedene Firmen aufteilen. Es ist sehr wahrscheinlich, dass viele der Firmen stark miteinander korreliert sind (zum Beispiel gehen eventuell ganze Wirtschaftssektoren gemeinsam nach oben oder unten), und wenn Sie korrelierte Firmen im Trainingsdatensatz und im Testdatensatz haben, wird Ihr Testdatensatz nicht sehr nützlich sein, da das Messen des Generalisierungsfehlers optimistisch beeinflusst ist.

Daher ist es oft sicherer, entlang der Zeitachse zu teilen – dabei wird aber implizit davon ausgegangen, dass die Muster, die das RNN in der Vergangenheit (im Trainingsdatensatz) lernen kann, auch in Zukunft existieren werden. Mit anderen Worten: Wir gehen davon aus, dass die Zeitserie *stationär* ist (zumindest im weitesten Sinne).³ Das kann man für viele Zeitserien vernünftigerweise annehmen (zum Beispiel sollten chemische Reaktionen in Ordnung sein, da sich die Gesetze der Chemie nicht jeden Tag ändern), aber für viele andere auch nicht (so sind beispielsweise Finanzmärkte notorisch nicht stationär, da Muster verschwinden, sobald die Händler sie erkennen und auszunutzen versuchen). Um sicherzustellen, dass die Zeitserie tatsächlich ausreichend stationär ist, können Sie die Modellfehler für den Validierungsdatensatz über die Zeit ausgeben: Verhält sich das Modell viel besser im ersten Teil des Validierungsdatensatzes als im letzten Teil, ist die Zeitserie eventuell nicht stationär genug, und Sie tun gut daran, das Modell mit einer kürzeren Zeitspanne zu trainieren.

Kurz gesagt, das Aufteilen einer Zeitserie in einen Trainings-, einen Validierungs- und einen Testdatensatz ist keine triviale Aufgabe, und es hängt stark von der Aufgabe ab, wie Sie es am besten tun.

Kehren wir nun zu Shakespeare zurück. Nehmen wir die ersten 90% des Texts für den Trainingsdatensatz (wir behalten den Rest für den Validierungs- und Testdatensatz) und erstellen wir ein `tf.data.Dataset`, das Zeichen für Zeichen aus diesem Datensatz liefert:

```
train_size = dataset_size * 90 // 100  
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

Den sequenziellen Datensatz in mehrere Fenster unterteilen

Der Trainingsdatensatz besteht nun aus einer Sequenz von über einer Million Zeichen, daher können wir das neuronale Netz nicht einfach damit trainieren: Das RNN würde einem Deep Net mit über einer Million Schichten entsprechen, und wir hätten eine einzelne (sehr lange) Instanz, um es zu trainieren. Stattdessen werden wir die Methode `window()` des Datasets verwenden, um diese lange Zeichensequenz in viele kleinere Textfenster zu unterteilen. Jede Instanz im Datensatz wird ein ziemlich kurzer Substring des gesamten Texts sein, und das RNN wird auf die Länge dieser Substrings verkleinert. Das nennt sich *Truncated Backpropagation Through*

Time. Rufen wir die Methode `window()` auf, um ein Dataset mit kurzen Textfenstern zu erstellen:

```
n_steps = 100  
window_length = n_steps + 1 # Ziel: Eingabe um 1 Zeichen verschoben  
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```



Sie können versuchen, `n_steps` anzupassen. Es ist einfacher, RNNs mit kürzeren Eingabesequenzen zu trainieren, aber natürlich wird das RNN keine Muster lernen können, die länger als `n_steps` sind – machen Sie es also nicht zu klein.

Standardmäßig erstellt die Methode `window()` nicht überlappende Fenster, aber um den größtmöglichen Trainingsdatensatz zu erhalten, verwenden wir `shift=1`, damit das erste Fenster die Zeichen 0 bis 100, das zweite 1 bis 101 und so weiter enthält. Um sicherzustellen, dass alle Fenster genau 101 Zeichen lang sind (was es uns erlaubt, Batches ohne Padding zu erstellen), setzen wir `drop_remainder=True` (ansonsten würden die letzten 100 Fenster 100 Zeichen, 99 Zeichen und so weiter bis hin zu 1 Zeichen enthalten).

Die Methode `window()` erstellt ein Dataset, das Fenster enthält, die jeweils wiederum als Dataset repräsentiert werden. Es handelt sich um ein *Nested Dataset*, was einer Liste mit Listen entspricht. Das ist nützlich, wenn Sie jedes Fenster durch einen Aufruf seiner Dataset-Methoden umwandeln wollen (zum Beispiel um sie zu durchmischen oder Batches daraus zu erstellen). Aber wir können ein Nested Dataset nicht direkt zum Training verwenden, da unser Modell Tensoren statt Datasets als Eingaben erwartet. Also müssen wir die Methode `flat_map()` aufrufen: Sie konvertiert ein Nested Dataset in ein *Flat Dataset* (eines, das keine anderen Datasets enthält). Stellen Sie sich beispielsweise vor, `{1, 2, 3}` repräsentiere ein Dataset mit der Sequenz aus den Tensoren 1, 2 und 3. Klopfen Sie das Nested Dataset `[[1, 2], [3, 4, 5, 6]]` flach, erhalten Sie das Flat Dataset `[1, 2, 3, 4, 5, 6]`. Zudem können Sie der Methode `flat_map()` eine Funktion als Argument mitgeben, mit der Sie jedes Dataset im Nested Dataset umwandeln können, bevor es flachgeklopft wird. Übergeben Sie beispielsweise die Funktion `lambda ds: ds.batch(2)`, wird das Nested Dataset `[[1, 2], [3, 4, 5, 6]]` umgewandelt in `[[[1, 2], [3, 4]], [[5, 6]]]` – es wird zu einem Dataset mit Tensoren der Größe 2. Also können wir jetzt unser Dataset flachklopfen:

```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

Beachten Sie, dass wir `batch(window_length)` für jedes Fenster aufrufen: Da alle Fenster genau die gleiche Länge haben, erhalten wir einen einzelnen Tensor für jedes von ihnen. Jetzt besteht das Dataset aus aufeinanderfolgenden Fenstern mit jeweils 101 Zeichen. Da die Gradientenmethode am besten funktioniert, wenn die Instanzen im Trainingsdatensatz unabhängig und gleichförmig verteilt sind (siehe [Kapitel 4](#)), müssen wir diese Fenster durchmischen. Dann können wir die Fenster batchen und die Eingaben (die ersten 100 Zeichen)

vom Ziel (dem letzten Zeichen) separieren:

```
batch_size = 32  
  
dataset = dataset.shuffle(10000).batch(batch_size)  
  
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
```

Abbildung 16-1 fasst die bisher besprochenen Vorbereitungsschritte für den Datensatz zusammen (mit Fenstern der Länge 11 statt 101 und einer Batchgröße von 3 statt 32).

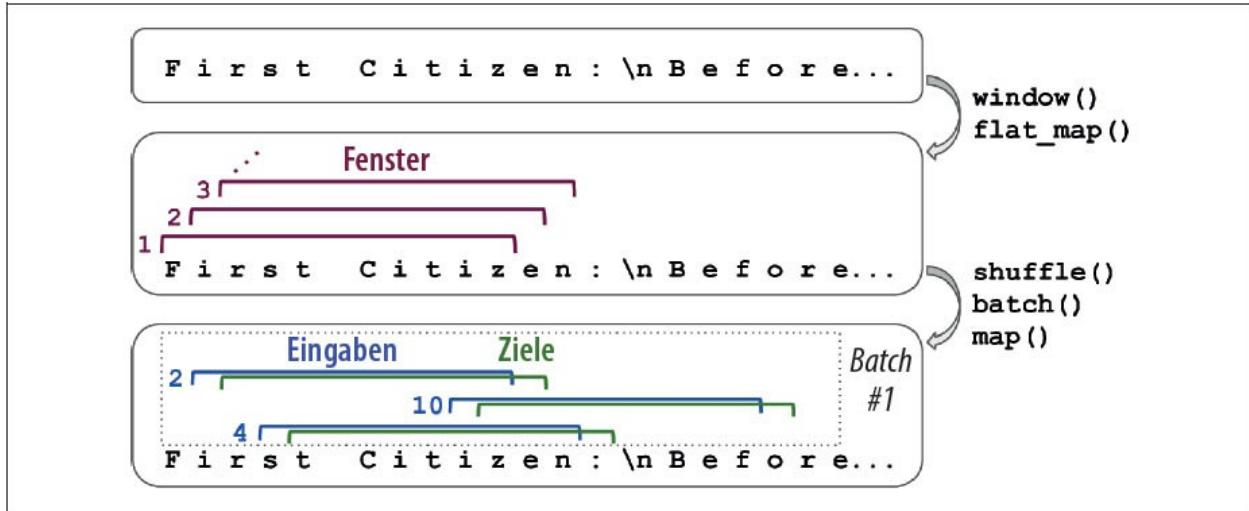


Abbildung 16-1: Einen Datensatz mit durchmischten Fenstern vorbereiten

Wie in Kapitel 13 besprochen, sollten kategorische Eingabemerkmale im Allgemeinen codiert werden – normalerweise als One-Hot-Vektoren oder Embeddings. Hier werden wir jedes Zeichen durch einen One-Hot-Vektor codieren, weil es verhältnismäßig wenige unterschiedliche Zeichen gibt (nur 39):

```
dataset = dataset.map(  
  
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

Schließlich müssen wir nur noch das Prefetching hinzufügen:

```
dataset = dataset.prefetch(1)
```

Das ist alles! Das Vorbereiten des Datensatzes war der schwerste Teil. Erstellen wir jetzt das Modell.

Das Char-RNN-Modell bauen und trainieren

Um ausgehend von den vorherigen 100 Zeichen das nächste Zeichen vorherzusagen, können wir ein RNN mit zwei GRU-Schichten mit jeweils 128 Einheiten und 20% Drop-out sowohl für die

Eingaben (`dropout`) wie auch für die verborgenen Status (`recurrent_dropout`) verwenden. Bei Bedarf können wir diese Hyperparameter später noch anpassen. Die Ausgabeschicht ist eine zeitverteilte Dense-Schicht wie die in [Kapitel 15](#). Dieses Mal muss diese Schicht 39 Einheiten haben (`max_id`), weil es 39 verschiedene Zeichen im Text gibt und wir eine Wahrscheinlichkeit für jedes mögliche Zeichen (bei jedem Zeitschritt) ausgeben wollen. Die Ausgabewahrscheinlichkeiten sollten sich bei jedem Zeitschritt zu 1 aufsummieren, daher wenden wir die Softmax-Aktivierungsfunktion auf die Ausgaben der Dense-Schicht an. Wir können dann dieses Modell kompilieren, wobei wir als Verlust `sparse_categorical_crossentropy` und einen Adam-Optimierer einsetzen. Schließlich sind wir bereit, das Modell viele Epochen lang zu trainieren (das kann abhängig von Ihrer Hardware viele Stunden dauern):

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=20)
```

Das Char-RNN-Modell verwenden

Jetzt haben wir ein Modell, das das nächste Zeichen in einem von Shakespeare geschriebenen Text vorhersagen kann. Um ihm Text zu übergeben, müssen wir ihn wie zuvor vorverarbeiten, also erstellen wir eine kleine Funktion dafür:

```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

Verwenden wir jetzt das Modell, um das nächste Zeichen in einem Text vorherzusagen:

```
>>> X_new = preprocess(["How are yo"])
>>> Y_pred = model.predict_classes(X_new)
```

```
>>> tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1. Satz, letztes Zeichen  
'u'
```

Erfolg! Das Modell hat richtig geschätzt. Nutzen wir es jetzt, um neuen Text zu erzeugen.

Einen gefälschten Shakespeare-Text erzeugen

Um mit dem Char-RNN-Modell neuen Text zu erzeugen, könnten wir ihm Text übergeben, das Modell den wahrscheinlichsten nächsten Buchstaben vorhersagen lassen, diesen am Ende des Texts anfügen, diesen erweiterten Text wieder an das Modell zum Vorhersagen des nächsten Buchstabens übergeben und so weiter. Aber in der Praxis führt dies oft dazu, dass wieder und wieder die gleichen Wörter wiederholt werden. Stattdessen können wir den nächsten Buchstaben zufällig mit einer Wahrscheinlichkeit auswählen, die der geschätzten Wahrscheinlichkeit entspricht. Dazu nutzen wir die TensorFlow-Funktion `tf.random.categorical()`. Dieses Vorgehen sorgt für abwechslungsreicheren und interessanteren Text. Die Funktion `categorical()` sampelt Zufallsindizes von Kategorien mithilfe der Log-Wahrscheinlichkeiten (Logits). Um mehr Kontrolle über die Diversität des erzeugten Texts zu haben, können wir die Logits durch eine Zahl dividieren, die als *Temperatur* bezeichnet wird und an der wir ganz nach Belieben drehen können: Eine Temperatur nahe null wird die Zeichen mit höherer Wahrscheinlichkeit bevorzugen, eine sehr hohe Temperatur wird hingegen allen Zeichen eine gleiche Wahrscheinlichkeit verschaffen. Die folgende Funktion `next_char()` nutzt dieses Vorgehen, um das nächste Zeichen auszuwählen, das dem Eingabetext angefügt werden soll:

```
def next_char(text, temperature=1):  
    X_new = preprocess([text])  
    y_proba = model.predict(X_new)[0, -1:, :]  
    rescaled_logits = tf.math.log(y_proba) / temperature  
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1  
    return tokenizer.sequences_to_texts(char_id.numpy())[0]
```

Als Nächstes können wir eine kleine Funktion schreiben, die `next_char()` wiederholt aufruft, um das nächste Zeichen zu erhalten und es an den gegebenen Text anzufügen:

```
def complete_text(text, n_chars=50, temperature=1):  
    for _ in range(n_chars):  
        text += next_char(text, temperature)  
    return text
```

Nun sind wir dazu in der Lage, Text zu generieren! Versuchen wir es mit unterschiedlichen Temperaturen:

```
>>> print(complete_text("t", temperature=0.2))  
the belly the great and who shall be the belly the  
>>> print(complete_text("w", temperature=1))  
thing? or why you gremio.  
who make which the first  
>>> print(complete_text("w", temperature=2))  
th no cce:  
yeolg-hormer firi. a play asks.  
fol rusb
```

Offensichtlich funktioniert unser Shakespeare-Modell am besten mit einer Temperatur nahe an 1. Um überzeugendere Texte zu generieren, könnten Sie versuchen, mehr GRU-Schichten und mehr Neuronen pro Schicht einzusetzen, länger zu trainieren und Regularisierung einzusetzen (beispielsweise könnten Sie `recurrent_drop out=0.3` in den GRU-Schichten setzen). Zudem ist das Modell so nicht in der Lage, Muster länger als `n_steps` zu lernen, was aktuell nur 100 Zeichen sind. Sie könnten versuchen, dieses Fenster größer zu machen, aber dadurch wird auch das Training schwerer, und selbst LSTM- oder GRU-Zellen können keine sehr langen Sequenzen verarbeiten. Alternativ könnten Sie ein zustandsbehaftetes RNN einsetzen.

Zustandsbehaftetes RNN

Bisher haben wir nur ein *zustandsloses RNN* verwendet: Bei jeder Trainingsiteration beginnt das Modell mit einem verborgenen Status voller Nullen, dann aktualisiert es diesen Status bei jedem Zeitschritt, und nach dem letzten Zeitschritt wirft es ihn weg, weil er nicht mehr benötigt wird. Was würde passieren, wenn wir das RNN anwiesen, diesen abschließenden Status nach dem Verarbeiten eines Trainingsbatchs aufzuheben und ihn als initialen Status für den nächsten Trainingsbatch zu verwenden? So könnte das Modell Langzeitmuster lernen, auch wenn nur kurze Sequenzen per Backpropagation verarbeitet werden. Dies wird als *zustandsbehaftetes RNN* bezeichnet. Schauen wir uns an, wie wir eines bauen können.

Zuerst einmal sollte Ihnen klar sein, dass ein zustandsbehaftetes RNN nur sinnvoll ist, wenn jede Eingabesequenz in einem Batch genau dort endet, wo die entsprechende Sequenz im vorherigen Batch geendet hat. Daher müssen wir vor allem dafür sorgen, dass wir sequenzielle und nicht überlappende Eingabesequenzen verwenden (statt die durchmischten und überlappenden Sequenzen, mit denen wir das zustandslose RNN trainiert haben). Erstellen wir das Dataset, müssen wir daher `shift=n_steps` verwenden (statt `shift=1`), wenn wir die Methode `window()` aufrufen. Zudem dürfen wir offensichtlich *nicht* die Methode `shuffle()` aufrufen.

Leider wird das Batching viel schwieriger, wenn wir einen Datensatz für ein zustandsbehaftetes RNN vorbereiten. Würden wir `batch(32)` aufrufen, würden die 32 aufeinanderfolgenden Fenster im gleichen Batch landen, und der folgende Batch würde nicht jedes dieser Fenster dort wieder aufnehmen, wo es endete. Der erste Batch würde die Fenster 1 bis 32, der zweite die Fenster 33 bis 64 enthalten. Nähmen Sie beispielsweise das erste Fenster aus jedem Batch (also die Fenster 1 und 33), würden Sie sehen, dass sie nicht aufeinanderfolgen. Am einfachsten ist es da, »Batches« zu verwenden, die aus nur einem einzigen Fenster bestehen:

```
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])

dataset = dataset.window(window_length, shift=n_steps, drop_remainder=True)

dataset = dataset.flat_map(lambda window: window.batch(window_length))

dataset = dataset.batch(1)

dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))

dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))

dataset = dataset.prefetch(1)
```

Abbildung 16-2 fasst diese ersten Schritte zusammen.

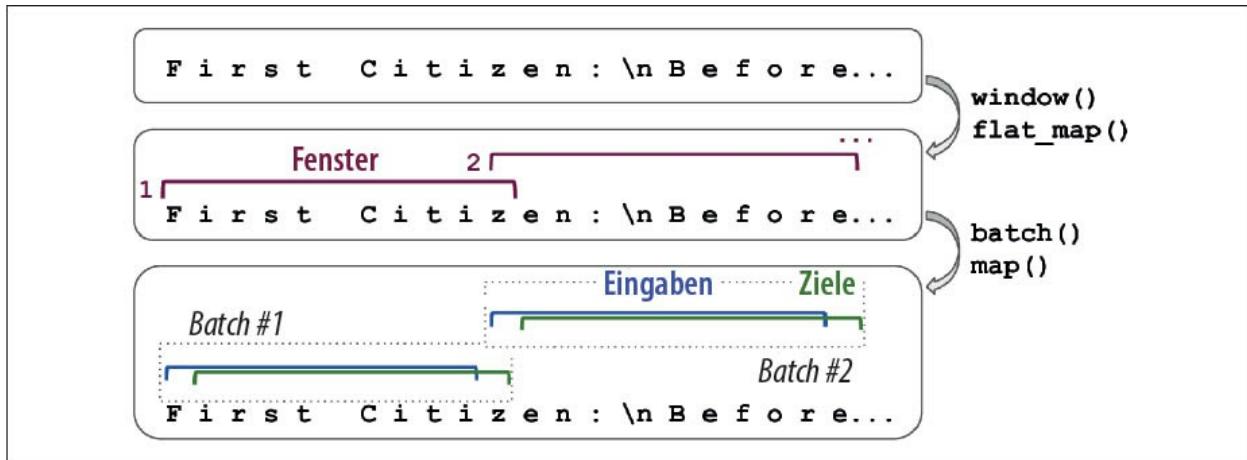


Abbildung 16-2: Einen Datensatz mit aufeinanderfolgenden Sequenzfragmenten für ein zustandsbehaftetes RNN vorbereiten

Das Batching ist schwieriger, aber nicht unmöglich. Wir könnten zum Beispiel Shakespeares Text in 32 Texte gleicher Länge unterteilen, ein Dataset mit aufeinanderfolgenden Eingabesequenzen für jeden davon erzeugen und schließlich `tf.train.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows))` verwenden, um passende aufeinanderfolgende Batches zu erstellen, bei denen die n . Eingabesequenz in einem Batch genau dort beginnt, wo die n . Eingabesequenz des vorherigen Batches endet (im Notebook finden Sie den vollständigen Code).

Erstellen wir jetzt das zustandsbehaftete RNN. Als Erstes müssen wir bei jeder rekurrenten Schicht, die wir anlegen, `stateful=True` setzen. Dann muss das zustandsbehaftete RNN die Batchgröße kennen (da es sich einen Status für jede Eingabesequenz im Batch merkt), daher müssen wir das Argument `batch_input_shape` in der ersten Schicht setzen. Beachten Sie, dass wir die zweite Dimension unspezifiziert lassen können, da die Eingaben beliebige Längen haben können:

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2,
                     batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
```

Am Ende jeder Epoche müssen wir die Status zurücksetzen, bevor wir wieder an den Anfang des Texts springen. Dazu verwenden wir einen kleinen Callback:

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

Jetzt können wir das Modell kompilieren und fitten (mit mehr Epochen, weil jede Epoche viel kleiner als zuvor ist und es nur eine Instanz pro Batch gibt):

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
model.fit(dataset, epochs=50, callbacks=[ResetStatesCallback()])
```

- ☞ Nachdem dieses Modell trainiert wurde, wird es nur genutzt werden können, um Vorhersagen für Batches der gleichen Größe wie beim Training zu erstellen. Um diese Einschränkung zu vermeiden, erstellen Sie ein identisches *zustandsloses* Modell und kopieren die Gewichte des zustandsbehafteten Modells dorthin.

Nachdem wir ein Modell auf Zeichenebene gebaut haben, ist es an der Zeit, uns das Modell auf Wortebene anzuschauen und eine häufige Aufgabe in der Verarbeitung natürlicher Sprache

anzugehen – die *Sentimentanalyse*. Dabei werden Sie lernen, wie Sie Sequenzen variabler Länge mithilfe des Maskierens verarbeiten können.

Sentimentanalyse

Wenn MNIST das »Hallo Welt« der Computer Vision ist, handelt es sich beim IMDb-Reviews-Datensatz um das »Hallo Welt« der natürlichen Sprachverarbeitung: Er besteht aus 50.000 Filmkritiken auf Englisch (25.000 zum Trainieren, 25.000 zum Testen), die aus der berühmten Internet Movie Database (<https://imdb.com/>) extrahiert wurden. Dazu gehört noch ein einfaches Binärziel für jede Kritik, die angibt, ob sie negativ (0) oder positiv (1) ist. Wie bei MNIST ist der IMDb-Review-Datensatz aus gutem Grund beliebt: Er ist ausreichend, um ihn auf einem Laptop in vernünftiger Zeit verarbeiten zu können, gleichzeitig aber herausfordernd genug, um damit Spaß zu haben. Keras bietet eine einfache Funktion, um ihn zu laden:

```
>>> (X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()  
  
>>> X_train[0][:10]  
  
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

Wo sind die Filmkritiken? Nun, wie Sie sehen, ist das Dataset für Sie schon vorverarbeitet: `X_train` besteht aus einer Liste von Reviews, die jeweils durch ein NumPy-Array mit Integer-Werten repräsentiert wird, wobei jede Zahl ein Wort darstellt. Alle Satzzeichen wurden entfernt, die Wörter wurden in Kleinbuchstaben umgewandelt, an den Leerzeichen getrennt und schließlich nach ihrer Häufigkeit indexiert (niedrige Zahlen stehen also für häufige Wörter). Die Integer-Zahlen 0, 1 und 2 sind besonders: Sie repräsentieren das Padding-Token, das *Start-of-Sequence*-(SoS-)Token und unbekannte Wörter. Wollen Sie eine Filmkritik ausgeben, können Sie sie wie folgt decodieren:

```
>>> word_index = keras.datasets.imdb.get_word_index()  
  
>>> id_to_word = {id_ + 3: word for word, id_ in word_index.items()}  
  
>>> for id_, token in enumerate(("<pad>", "<sos>", "<unk>")):  
...     id_to_word[id_] = token  
  
...  
  
>>> " ".join([id_to_word[id_] for id_ in X_train[0][:10]])  
  
'<sos> this film was just brilliant casting location scenery story'
```

In einem richtigen Projekt werden Sie den Text selbst vorverarbeiten müssen. Sie können dazu die weiter oben schon eingesetzte Klasse `Tokenizer` nutzen, aber dieses Mal mit `char_level=False` (dem Standard). Beim Codieren von Wörtern werden viele Zeichen ausgefiltert, einschließlich eines Großteils der Satzzeichen, Zeilenumbrüche und Tabs (aber Sie

können das über das Argument `filters` anpassen). Am wichtigsten ist, dass die Klasse Leerzeichen nutzt, um Wortgrenzen zu erkennen. Das ist für Englisch und viele andere Schriftsysteme in Ordnung, aber nicht überall kommen Leerzeichen zum Einsatz. Vietnamesisch verwendet Leerzeichen auch mitten in Wörtern, und in Sprachen wie Deutsch werden mehrere Wörter gern ohne Leerzeichen aneinandergehängt. Und selbst im Englischen sind Leerzeichen nicht immer die beste Option, Text zu tokenize – denken Sie nur an »San Francisco« oder »#ILoveDeepLearning«.

Zum Glück gibt es bessere Möglichkeiten! Ein Artikel (<https://homl.info/subword>)⁴ aus dem Jahr 2018 von Taku Kudo stellt eine Technik des unüberwachten Lernens zum sprachunabhängigen Tokenizen und Detokenizen von Text auf Subwortebene vor, die Leerzeichen wie ganz normale Zeichen behandelt. Mit diesem Ansatz kann Ihr Modell auch beim Finden eines noch unbekannten Worts eine sinnvolle Einschätzung seiner Bedeutung abgeben. So hat es vielleicht während des Trainings noch nie das Wort »smartest« gesehen, aber eventuell schon das Wort »smart« gelernt und auch erlernt, dass das Suffix »est« »am meisten« bedeutet. Damit kann es die Bedeutung von »smartest« ableiten. Googles *SentencePiece*-Projekt (<https://github.com/google/sentencepiece>) liefert dazu eine Open-Source-Implementierung, die in einem Artikel (<https://homl.info/sentencepiece>)⁵ von Taku Kudo und John Richardson beschrieben ist.

Eine andere Möglichkeit wurde in einem älteren Artikel (<https://homl.info/rarewords>)⁶ von Rico Sennrich et al. vorgeschlagen. Dabei werden Subwörter auf andere Weise codiert (zum Beispiel per *Byte-Pair Encoding*). Und schließlich hat das TensorFlow-Team im Juni 2019 die TF.Text-Bibliothek (<https://homl.info/tftext>) veröffentlicht, die verschiedene Tokenization-Strategien implementiert, unter anderem WordPiece (<https://homl.info/wordpiece>)⁷ (eine Variante des Byte-Pair Encoding).

Wollen Sie Ihr Modell auf einem mobilen Gerät oder in einem Webbrowser deployen und möchten Sie nicht jedes Mal eine andere Vorverarbeitungsfunktion schreiben, sollten Sie für die Vorverarbeitung nur TensorFlow-Operationen einsetzen, damit sie im Modell selbst eingebaut werden kann. Schauen wir uns an, wie das geht. Zuerst laden wir die Original-IMDb-Reviews als Text (Bytestrings) mithilfe von TensorFlow-Datasets (siehe [Kapitel 13](#)):

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)

train_size = info.splits["train"].num_examples
```

Dann schreiben wir die Vorverarbeitungsfunktion:

```
def preprocess(x_batch, y_batch):

    x_batch = tf.strings.substr(x_batch, 0, 300)

    x_batch = tf.strings.regex_replace(x_batch, b"<br\\s*/?>", b" ")
```

```

X_batch = tf.strings.regex_replace(X_batch, b"[^a-zA-Z]", b" ")
X_batch = tf.strings.split(X_batch)

return X_batch.to_tensor(default_value=b"<pad>"), y_batch

```

Zuerst werden die Reviews abgeschnitten und nur die ersten 300 Zeichen behalten. Das beschleunigt das Training und beeinflusst die Leistung nicht zu sehr, weil Sie im Allgemeinen schon nach den ersten ein oder zwei Sätzen sagen können, ob eine Kritik positiv oder negativ ist. Dann werden mithilfe regulärer Ausdrücke
-Tags durch Leerzeichen ersetzt, ebenso alle Zeichen, die keine Buchstaben oder Anführungszeichen sind. So wird zum Beispiel der Text "Well, I can't
" zu "Well I can't". Schließlich teilt die Funktion `preprocess()` die Reviews an den Leerzeichen auf und erhält damit einen Ragged-Tensor, den sie in einen Dense-Tensor umwandelt, wobei sie alle Reviews mit dem Padding-Token "<pad>" so auffüllt, dass sie die gleiche Länge bekommen.

Als Nächstes müssen wir das Vokabular erstellen. Dazu müssen wir einmal durch den gesamten Trainingsdatensatz gehen, unsere Funktion `preprocess()` anwenden und einen Counter nutzen, um das Auftreten jedes Worts zu zählen:

```

from collections import Counter

vocabulary = Counter()

for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))

```

Schauen wir uns die drei häufigsten Wörter an:

```

>>> vocabulary.most_common()[:3]
[(b'<pad>', 215797), (b'the', 61137), (b'a', 38564)]

```

Super! Unser Modell muss vermutlich nicht alle Wörter im Dictionary kennen, um eine gute Performance zu liefern, daher wollen wir das Vokabular nach den 10.000 häufigsten Wörtern abschneiden:

```

vocab_size = 10000

truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]

```

Jetzt müssen wir einen Vorverarbeitungsschritt ergänzen, um jedes Wort durch seine ID zu ersetzen (also seinen Index im Vokabular). Wie in [Kapitel 13](#) werden wir dafür eine Lookup-

Tabelle mit 1.000 Out-of-Vocabulary-(OoV-)Buckets verwenden:

```
words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

Die Tabelle können wir dann nutzen, um die IDs von ein paar Wörtern auszugeben:

```
>>> table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
<tf.Tensor: [...], dtype=int64, numpy=array([[ 22,     12,     11, 10054]])>
```

Beachten Sie, dass die Wörter »this«, »movie« und »was« in der Tabelle gefunden wurden, womit ihre IDs kleiner als 10.000 sind, während das Wort »faaaaaantastic« nicht enthalten ist und damit einem der OoV-Buckets mit einer ID von mindestens 10.000 zugeordnet wurde.

TF Transform (siehe [Kapitel 13](#)) stellt einige nützliche Funktionen für die Arbeit mit solchen Vokabularen bereit. Schauen Sie sich beispielsweise die Funktion `tft.compute_and_apply_vocabulary()` an: Sie durchläuft das Dataset, um alle unterschiedlichen Wörter zu finden und ein Vokabular aufzubauen, und dann erzeugt sie die TF-Operationen, die zum Codieren jedes Worts mit diesem Vokabular erforderlich sind.

Jetzt können wir endlich den Trainingsdatensatz erstellen. Wir batchen die Reviews, wandeln sie mit der Funktion `preprocess()` in kurze Wortsequenzen um, codieren diese Wörter mit einer einfachen Funktion `encode_words()`, die die gerade erstellte Tabelle einsetzt, und prefetchen schließlich den nächsten Batch:

```
def encode_words(X_batch, y_batch):
    return table.lookup(X_batch), y_batch

train_set = datasets["train"].batch(32).map(preprocess)
train_set = train_set.map(encode_words).prefetch(1)
```

Nun können wir das Modell erstellen und trainieren:

```
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
```

```

        input_shape=[None]),

    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")

])

model.compile(loss="binary_crossentropy", optimizer="adam",
               metrics=["accuracy"])

history = model.fit(train_set, epochs=5)

```

Die erste Schicht ist eine Embedding-Schicht, die Wort-IDs in Embeddings umwandelt (siehe [Kapitel 13](#)). Die Embedding-Matrix benötigt eine Zeile pro Wort-ID (`vocab_size + num_oov_buckets`) und eine Spalte pro Embedding-Dimension (dieses Beispiel nutzt 128 Dimensionen, aber das ist ein Hyperparameter, an dem Sie drehen können). Während die Eingaben des Modells 2-D-Tensoren der Form [*Batchgröße, Zeitschritte*] sind, handelt es sich bei der Ausgabe der Embedding-Schicht um einen 3-D-Tensor der Form [*Batchgröße, Zeitschritte, Embeddinggröße*].

Der Rest des Modells ist recht klar: Er besteht aus zwei GRU-Schichten, wobei die zweite nur die Ausgabe des letzten Zeitschritts zurückgibt. Die Ausgabeschicht ist ein einzelnes Neuron mit der Sigmoid-Aktivierungsfunktion, um die geschätzte Wahrscheinlichkeit zu liefern, dass das Review ein positives Gefühl bezüglich des Films ausdrückt. Wir kompilieren dann das Modell ziemlich einfach und passen es für ein paar Epochen auf den zuvor vorbereiteten Datensatz an.

Maskieren

In der aktuellen Umsetzung wird das Modell lernen, dass die Padding-Token ignoriert werden sollten. Aber das wissen wir doch schon! Warum erzählen wir es nicht dem Modell, sodass es sich auf die Daten konzentrieren kann, die wirklich wichtig sind? Das ist sogar ganz einfach: Ergänzen Sie beim Erstellen der Embedding-Schicht einfach `mask_zero=True`. Damit werden Padding-Token (deren ID 0 ist)⁸ von allen weiteren Schichten ignoriert.

Die Embedding-Schicht erzeugt nun einen *Maskierungstensor* mit dem Inhalt `K.not_equal(inputs, 0)` (wobei `K = keras.backend`): Das ist ein boolescher Tensor mit der gleichen Form wie die Eingaben, der überall dort `False` ist, wo die Wort-IDs den Wert 0 haben, während er sonst den Wert `True` enthält. Dieser Maskierungstensor wird dann automatisch vom Modell an alle folgenden Schichten weitergegeben, solange die Zeitdimension beibehalten wird. In diesem Beispiel erhalten also beide GRU-Schichten automatisch diese Maske, aber da die zweite GRU-Schicht keine Sequenzen zurückgibt (sie liefert nur die Ausgabe des letzten Zeitschritts), wird die Maske nicht an die Dense-Schicht weitergegeben. Jede Schicht geht eventuell anders mit der Maske um, aber im Allgemeinen werden maskierte Zeitschritte einfach ignoriert (also solche, für die die Maske `False` ist). Trifft beispielsweise eine

rekurrierende Schicht auf einen maskierten Zeitschritt, kopiert sie einfach die Ausgabe des vorherigen Zeitschritts. Wird die Maske komplett bis zur Ausgabe weitergegeben (in Modellen, die Sequenzen ausgeben, was hier nicht der Fall ist), wird sie auch auf die Verluste angewendet, sodass die maskierten Zeitschritte nicht zum Verlust beitragen (ihr Verlust wird 0 sein).

 Die Schichten LSTM und GRU besitzen eine optimierte Implementierung für GPUs, die auf Nvidias cuDNN-Bibliothek basieren. Diese Implementierung unterstützt allerdings kein Maskieren. Nutzt Ihr Modell eine Maske, werden diese Schichten auf die (viel langsamere) Standardimplementierung zurückfallen. Beachten Sie, dass Sie bei der optimierten Implementierung auch die Standardwerte einiger Hyperparameter nutzen müssen: activation, recurrent_activation, recurrent_dropout, unroll, use_bias und reset_after.

Alle Schichten, die die Maske übergeben bekommen, müssen das Maskieren unterstützen (ansonsten wird eine Exception geworfen). Dazu gehören alle rekurrenden Schichten, aber auch die TimeDistributed-Schicht und ein paar andere Schichten. Jede Schicht, die ein Maskieren unterstützt, muss ein Attribut supports_masking besitzen, das auf True gesetzt ist. Wollen Sie Ihre eigene Schicht mit Maskierung implementieren, sollten Sie die Methode call() um ein Argument mask erweitern (und offensichtlich die Methode die Maske auch irgendwie verwenden lassen). Zudem sollten Sie im Konstruktor self.supports_masking=True setzen. Beginnt Ihre Schicht nicht mit einer Embedding-Schicht, können Sie stattdessen keras.layers.Masking verwenden: Es setzt die Maske auf K.any(K.not_equal(inputs, 0), axis=-1), was heißt, dass Zeitschritte, bei denen die letzte Dimension voller Nullen ist, in folgenden Schichten maskiert werden (auch wieder solange die Zeitdimension existiert).

Der Einsatz von maskierenden Schichten und dem automatischen Weiterleiten von Masken funktioniert am besten für einfache Sequential-Modelle. Bei komplexeren Modellen geht das nicht immer, zum Beispiel wenn Sie Conv1D-Schichten mit rekurrenden Schichten mischen. In solchen Fällen werden Sie die Maske explizit berechnen und an die entsprechenden Schichten weitergeben müssen – entweder über die Functional API oder die Subclassing API. Das folgende Modell ist beispielsweise identisch mit dem vorherigen Modell, nur ist es über die Functional API aufgebaut und kümmert sich manuell um das Maskieren:

```
K = keras.backend

inputs = keras.layers.Input(shape=[None])

mask = keras.layers.Lambda(lambda inputs: K.not_equal(inputs, 0))(inputs)

z = keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size)(inputs)

z = keras.layers.GRU(128, return_sequences=True)(z, mask=mask)

z = keras.layers.GRU(128)(z, mask=mask)

outputs = keras.layers.Dense(1, activation="sigmoid")(z)

model = keras.Model(inputs=[inputs], outputs=[outputs])
```

Nach dem Trainieren über ein paar Epochen wird dieses Modell ziemlich gut bei der Voraussage werden, ob eine Kritik positiv oder negativ ist. Verwenden Sie den Callback `TensorBoard()`, können Sie die Embeddings in TensorBoard während des Lernens visualisieren: Es ist faszinierend zu beobachten, wie sich Wörter wie »awesome« und »amazing« auf der einen Seite des Embedding-Raums zusammensetzen, während sich Wörter wie »awful« oder »terrible« auf der anderen Seite finden. Manche Wörter sind nicht so positiv, wie Sie vielleicht erwarten würden (zumindest mit diesem Modell), wie zum Beispiel »good« – vermutlich, weil viele negative Kritiken die Phrase »not good« enthalten. Beeindruckend ist, dass das Modell nützliche Word-Embeddings mit nur 25.000 Filmkritiken lernen kann. Stellen Sie sich vor, wie gut die Embeddings wären, wenn wir Milliarden von Reviews zum Trainieren besäßen! Leider haben wir das nicht, aber vielleicht können wir Word Embeddings wiederverwenden, die mit anderen großen Textkorpora trainiert wurden (zum Beispiel Wikipedia-Artikeln), auch wenn diese nicht aus Filmkritiken bestehen? Denn schließlich hat das Wort »amazing« im Allgemeinen immer die gleiche Bedeutung – egal ob Sie über Filme oder etwas anderes sprechen. Zudem wären Embeddings vielleicht auch dann zur Sentimentanalyse nützlich, wenn sie für eine andere Aufgabe trainiert wurden: Da Wörter wie »awesome« und »amazing« eine ähnliche Bedeutung haben, werden sie sich im Embedding-Raum auch dann clustern, wenn es um andere Aufgaben geht (zum Beispiel das Vorhersagen des nächsten Worts in einem Satz). Wenn alle positiven Wörter und alle negativen Wörter Cluster bilden, werden diese für die Sentimentanalyse hilfreich sein. Statt also so viele Parameter zum Lernen von Word Embeddings zu verwenden, schauen wir doch mal, ob wir nicht einfach vortrainierte Embeddings nutzen können.

Vortrainierte Embeddings wiederverwenden

Das TensorFlow-Hub-Projekt macht es einfach, vortrainierte Modellkomponenten in Ihren eigenen Modellen einzusetzen. Diese Modellkomponenten werden als *Module* bezeichnet. Wenn Sie sich im TF Hub Repository (<https://tfhub.dev>) umschauen, finden Sie das Benötigte und können das Codebeispiel in Ihre Projekt kopieren. Das Modul wird dann automatisch zusammen mit seinen vortrainierten Gewichten heruntergeladen und in Ihr Modell eingebunden. Ganz einfach!

Verwenden wir beispielsweise das Sentence-Embedding-Modul `nnlm-en-dim50` in Version 1 in unserem Modell zur Sentimentanalyse:

```
import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                  dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")])
```

```
])
model.compile(loss="binary_crossentropy", optimizer="adam",
metrics=["accuracy"])
```

Die Schicht `hub.KerasLayer` lädt das Modul von der angegebenen URL herunter. Dieses spezielle Modul ist ein *Sentence Encoder*: Es übernimmt Strings als Eingaben und codiert jeden als einzelnen Vektor (in diesem Fall als 50-dimensionalen Vektor). Intern parst es den String (und trennt die Wörter an den Leerzeichen) und erstellt für jedes Wort ein Embedding mithilfe einer Embedding-Matrix, die mit einem riesigen Korpus trainiert wurde – dem Google News 7B Corpus (sieben Milliarden Wörter groß!). Dann berechnet es den Mittelwert aller Word Embeddings, und das Ergebnis ist das Sentence Embedding.⁹ Wir können dann zwei einfache Dense-Schichten hinzufügen, um ein gutes Modell zur Sentimentanalyse zu erstellen. Standardmäßig ist ein `hub.KerasLayer` nicht trainierbar, aber Sie können beim Erstellen `trainable=True` setzen, um das zu ändern, sodass Sie ihn für Ihre Aufgabe optimieren können.



Nicht alle TF-Hub-Module unterstützen TensorFlow 2, achten Sie also darauf, dass Sie ein Modul auswählen, das das tut.

Als Nächstes können wir einfach den Datensatz mit den IMDb-Reviews herunterladen – wir müssen ihn nicht mehr vorverarbeiten (außer zum Batchen und Prefetchen) – und das Modell direkt trainieren:

```
datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)

train_size = info.splits["train"].num_examples

batch_size = 32

train_set = datasets["train"].batch(batch_size).prefetch(1)

history = model.fit(train_set, epochs=5)
```

Beachten Sie, dass der letzte Teil der URL des TF-Hub-Moduls angibt, dass wir Version 1 des Modells haben wollten. Dieses Versionieren stellt sicher, dass unser Modell auch dann noch funktioniert, wenn eine neue Modulversion veröffentlicht wird. Geben Sie diese URL einfach in einen Webbrowser ein, werden Sie praktischerweise die Dokumentation für dieses Modul erhalten. Standardmäßig wird TF Hub die heruntergeladenen Dateien im Temp-Verzeichnis des lokalen Systems cachen. Vielleicht möchten Sie sie lieber in ein dauerhafteres Verzeichnis herunterladen, um zu vermeiden, dass sie erneut heruntergeladen werden, wenn das System aufgeräumt wird. Dazu setzen Sie die Umgebungsvariable `TFHUB_CACHE_DIR` auf das Verzeichnis Ihrer Wahl (zum Beispiel `os.environ["TFHUB_CACHE_DIR"] = "./my_tfhub_cache"`).

Bisher haben wir uns Zeitserien, Texterzeugung mithilfe von Char-RNNs und Sentimentanalyse

mit RNN-Modellen auf Wortebene angeschaut und dabei unsere eigenen Word Embeddings trainiert oder vortrainierte Embeddings eingesetzt. Wenden wir uns nun einer anderen wichtigen NLP-Aufgabe zu: *neuronale maschinelle Übersetzung* (Neural Machine Translation, NMT). Dazu nutzen wir zuerst ein reines Encoder-Decoder-Modell, dann verbessern wir es mit Attention-Mechanismen, und schließlich werfen wir einen Blick auf die außerordentliche Transformer-Architektur.

Ein Encoder-Decoder-Netzwerk für die neuronale maschinelle Übersetzung

Schauen wir uns ein einfaches Modell zur neuronalen maschinellen Übersetzung (<https://homl.info/103>) an¹⁰, das englische Sätze ins Französische übersetzt (siehe Abbildung 16-3).

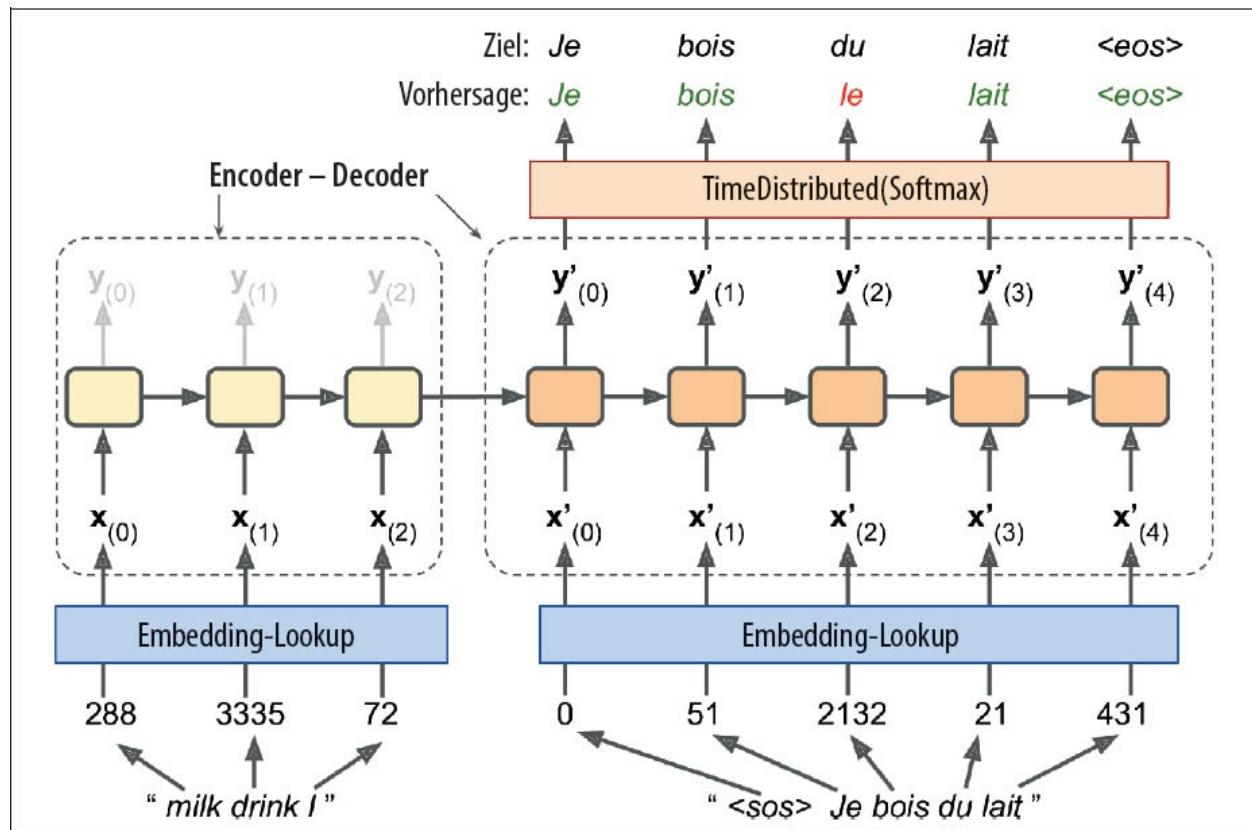


Abbildung 16-3: Ein einfaches Modell zur maschinellen Übersetzung

Kurz gesagt, werden die englischen Sätze an den Encoder übergeben, und der Decoder liefert die französische Übersetzung. Beachten Sie, dass die französischen Übersetzungen auch als Eingaben für den Decoder verwendet werden, allerdings um einen Schritt zurückverschoben. Mit anderen Worten: Dem Decoder wird als Eingabe das Wort übergeben, das er als Ausgabe im vorherigen Schritt hätte ausgeben sollen (unabhängig davon, was er tatsächlich ausgegeben hat). Beim allerersten Wort wird stattdessen das Start-of-Sequence-(SOS-)Token übermittelt. Der Decoder soll den Satz mit einem End-of-Sequence-(EOS-)Token beenden.

Beachten Sie, dass die englischen Sätze umgedreht werden, bevor sie in den Encoder geraten. So wird beispielsweise »I drink milk« umgekehrt zu »milk drink I«. Das stellt sicher, dass der Anfang des englischen Satzes als Letztes an den Encoder gegeben wird, was nützlich ist, weil dies im Allgemeinen das erste ist, was der Decoder übersetzen muss.

Jedes Wort wird initial durch seine ID repräsentiert (zum Beispiel 288 für »milk«). Als Nächstes gibt eine embedding-Schicht das Word Embedding zurück. Diese Word Embeddings sind das, was eigentlich an den Encoder und Decoder übergeben wird.

Bei jedem Schritt gibt der Decoder einen Score für jedes Wort im Ausgabevokabular (also Französisch) aus, dann wandelt die Softmax-Schicht diese Scores in Wahrscheinlichkeiten um. So hat beispielsweise im ersten Schritt das Wort »Je« vielleicht eine Wahrscheinlichkeit von 20%, »Tu« eine von 1% und so weiter. Das Wort mit der größten Wahrscheinlichkeit ist die Ausgabe. Das entspricht recht weitgehend einer normalen Klassifikationsaufgabe, daher können Sie das Modell mit dem Verlust "sparse_categorical_crossentropy" trainieren, so wie wir es im Char-RNN-Modell getan haben.

Beachten Sie, dass Sie den Zielsatz zur Inferenzzeit (nach dem Training) nicht an den Decoder übergeben müssen. Übergeben Sie stattdessen dem Decoder einfach das Wort, das er im vorherigen Schritt ausgegeben hat, wie in [Abbildung 16-4](#) zu sehen ist (dazu ist ein Embedding-Lookup notwendig, das im Diagramm nicht zu sehen ist).

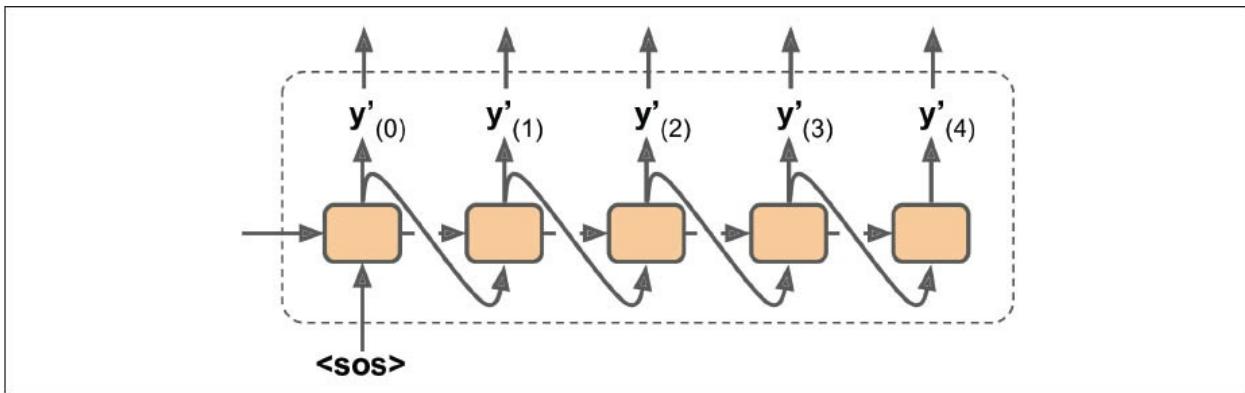


Abbildung 16-4: Das Wort der vorherigen Ausgabe zur Inferenzzeit als Eingabe übergeben

Okay, jetzt haben Sie einen Überblick erhalten. Es gibt aber ein paar Details, um die Sie sich kümmern müssen, wenn Sie dieses Modell implementieren:

- Bisher sind wir davon ausgegangen, dass alle Eingabesequenzen (an den Encoder und den Decoder) eine konstante Länge haben. Aber offensichtlich variieren die Satzlängen. Da normale Tensoren feste Formen besitzen, können sie nur Sätze der gleichen Länge aufnehmen. Sie können das per Maskieren angehen, wie wir es weiter oben besprochen haben. Aber wenn die Sätze sehr unterschiedliche Längen besitzen, können Sie sie nicht einfach abschneiden, wie wir es bei der Sentimentanalyse getan haben (weil wir vollständige statt abgeschnittene Übersetzungen haben wollen). Gruppieren Sie stattdessen Sätze in Buckets mit ähnlicher Länge (zum Beispiel ein Bucket für die Ein- bis Sechs-Wort-Sätze, ein anderer für die Sieben- bis Zwölf-Wort-Sätze und so weiter) und verwenden Sie Padding für die kürzeren Sequenzen, um sicherzustellen, dass alle

Sätze in einem Bucket die gleiche Länge haben (schauen Sie sich dafür die Funktion `tf.data.experimental.bucket_by_sequence_length()` an). So wird beispielsweise »I drink milk« zu »<pad> <pad> <pad> milk drink I«.

- Wir wollen jegliche Ausgabe hinter dem EOS-Token ignorieren, daher sollten diese Token nicht zum Verlust beitragen (sie müssen maskiert werden). Gibt das Modell beispielsweise »Je boid du lait <eos> oui« aus, sollte der Verlust für das letzte Wort ignoriert werden.
- Ist das Ausgabevokabular groß (wie in diesem Fall), kann die Ausgabe einer Wahrscheinlichkeit für jedes einzelne Wort unerträglich langsam sein. Enthält das Zielvokabular zum Beispiel 50.000 französische Wörter, würde der Decoder 50.000-dimensionale Vektoren ausgeben, und das Ermitteln der Softmax-Funktion über solch einen großen Vektor wäre sehr rechenintensiv. Um das zu vermeiden, kann man sich nur die Logits-Ausgaben des Modells für das korrekte Wort und für einen zufälligen Satz falscher Wörter anschauen und dann eine Näherung des Verlusts basierend auf diesen Logits berechnen. Diese *Sampled-Softmax*-Technik wurde 2015 von Sébastien Jean et al. (<https://homl.info/104>) vorgestellt¹¹. In TensorFlow können Sie dafür während des Trainings die Funktion `tf.nn.sampled_softmax_loss()` verwenden, während beim Einsatz des Modells die normale Softmax-Funktion zum Einsatz kommt (dort ist Sampled Softmax nicht möglich, weil das Ziel bekannt sein muss).

Das TensorFlow-Addons-Projekt enthält viele Sequence-to-Sequence-Tools, mit denen Sie recht einfach produktiv einsetzbare Encoder-Decoder bauen können. So erzeugt beispielsweise der folgende Code ein einfaches Encoder-Decoder-Modell, das dem in Abbildung 16-3 ähnelt:

```
import tensorflow_addons as tfa

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)

embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)

encoder = keras.layers.LSTM(512, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()
```

```

decoder_cell = keras.layers.LSTMCell(512)

output_layer = keras.layers.Dense(vocab_size)

decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell, sampler,
                                                 output_layer=output_layer)

final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)

Y_proba = tf.nn.softmax(final_outputs.rnn_output)

model = keras.Model(inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
                     outputs=[Y_proba])

```

Dieser Code ist weitgehend selbsterklärend, aber es gibt ein paar Dinge, auf die hingewiesen werden sollte. So setzen wir beim Erstellen der LSTM-Schicht `return_state=True`, sodass wir den abschließenden verborgenen Status erhalten und an den Decoder übergeben können. Da wir eine LSTM-Zelle verwenden, werden sogar zwei verborgene Status zurückgegeben (kurzfristig und langfristig). Der `Training Sampler` ist einer der diversen Samplers, die in TensorFlow Addons zu finden ist: Ihre Rolle ist, dem Decoder bei jedem Schritt zu sagen, als was die vorherige Ausgabe anzunehmen ist. Während der Inferenz sollte dies das Embedding des Tokens sein, das tatsächlich ausgegeben wurde. Während des Trainings sollte es das Embedding des vorherigen Zieltokens sein – darum der Einsatz des `TrainingSampler`. In der Praxis ist es häufig sinnvoll, das Training mit dem Embedding des Ziels des vorherigen Zeitschritts zu starten und nach und nach zur Verwendung des Embedding des Token zu wechseln, das tatsächlich im vorherigen Schritt ausgegeben wurde. Diese Idee wurde in einem Artikel (<https://hml.info/scheduledsampling>) aus dem Jahr 2015 von Samy Bengio et al. vorgeschlagen¹². Der `ScheduledEmbeddingTraining Sampler` wählt zufällig zwischen dem Ziel und der tatsächlichen Ausgabe, wobei Sie die Wahrscheinlichkeit während des Trainings nach und nach anpassen können.

Bidirektionale RNNs

Bei jedem Zeitschritt schaut eine normale rekurrente Schicht nur auf die vergangenen und aktuellen Eingaben, bevor sie ihre Ausgabe erzeugt. Sie ist »kausal«, sie kann also nicht in die Zukunft schauen. Diese Art von RNN ist sinnvoll, wenn es um die Vorhersage von Zeitserien geht, aber bei vielen NLP-Aufgaben, wie zum Beispiel bei der neuronalen maschinellen Übersetzung, möchte man auch auf die nächsten Wörter schauen können, bevor ein gegebenes Wort codiert wird. Nehmen wir beispielsweise die Phrasen »the Queen of the United Kingdom«, »the queen of hearts« und »the queen bee«: Um das Wort »queen« korrekt zu codieren, müssen Sie nach vorne schauen. Um dies zu implementieren, lassen Sie zwei rekurrente Schichten mit

den gleichen Eingaben laufen, von denen die eine die Wörter von links nach rechts und die andere sie von rechts nach links laufen lässt. Dann kombinieren Sie einfach bei jedem Zeitschritt deren Ausgaben, meist per Konkatenieren. Das nennt sich eine *bidirektionale rekurrente Schicht* (siehe Abbildung 16-5).

Um eine bidirektionale rekurrente Schicht in Keras zu implementieren, verpacken Sie eine rekurrente Schicht in einer Schicht vom Typ `keras.layers.Bidirectional`. Der folgende Code erzeugt beispielsweise eine bidirektionale GRU-Schicht:

```
keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
```



Die `Bidirectional`-Schicht erzeugt einen Klon der `GRU`-Schicht (aber in umgekehrter Richtung), führt beide aus und konkateniert ihre Ausgaben. Somit hat die `GRU`-Schicht zwar 10 Einheiten, die `Bidirectional`-Schicht aber 20 Werte pro Zeitschritt.

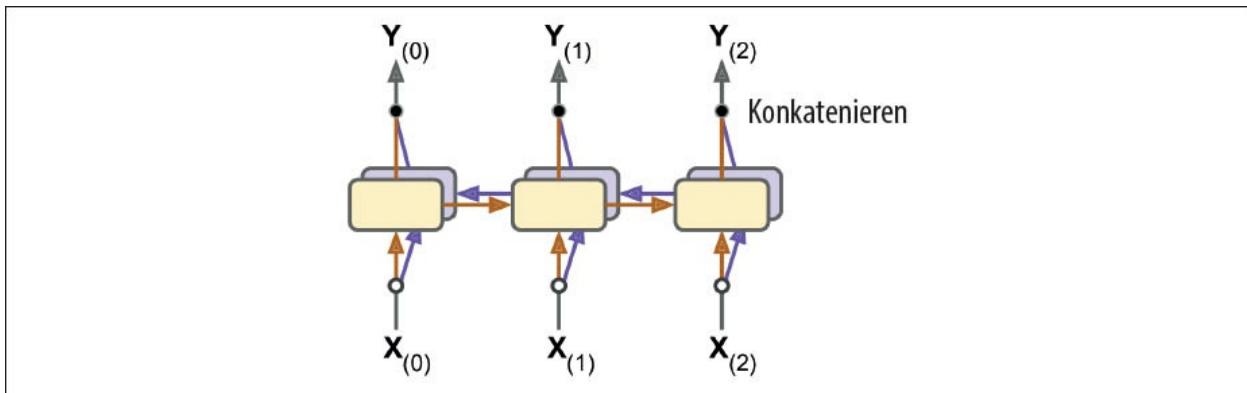


Abbildung 16-5: Eine bidirektionale rekurrente Schicht

Beam Search

Stellen Sie sich vor, Sie trainierten ein Encoder-Decoder-Modell und nutzen es, um den französischen Satz »Comment vas-tu?« ins Englische zu übersetzen. Sie hoffen, dass es die korrekte Übersetzung (»How are you?«) ausgeben wird, aber leider liefert es »How will you?«. Schauen Sie sich den Trainingsdatensatz an, finden Sie viele Sätze wie »Comment vas-tu jouer?«, der als »How will you play?« übersetzt wird. Das Modell hat also mit »How will« gar keine so absurde Ausgabe geliefert, nachdem es »Comment vas« gelesen hat. Leider war das in diesem Fall ein Fehler, und das Modell konnte nicht zurückgehen und ihn beheben, daher hat es versucht, den Satz so gut wie möglich zu beenden. Indem bei jedem Schritt gierig das wahrscheinlichste Wort ausgegeben wird, war die Übersetzung am Ende nicht optimal. Wie können wir dem Modell die Chance geben, zurückzugehen und Fehler zu beheben, die es früher gemacht hat? Eine häufig eingesetzte Lösung ist die *Beam Search*: Sie merkt sich eine kurze Liste der k vielversprechendsten Sätze (beispielsweise drei) und versucht bei jedem Encoder-Schritt, sie um ein Wort zu erweitern und dann wieder nur die k wahrscheinlichsten Sätze zu behalten. Der Parameter k wird als *Beam Width* bezeichnet.

Nehmen wir an, Sie nutzen das Modell zum Übersetzen des Satzes »Comment vastu?« mit Beam

Search und einer Beam Width von 3. Im ersten Decoder-Schritt wird das Modell eine geschätzte Wahrscheinlichkeit für jedes mögliche Wort ausgeben. Stellen Sie sich vor, die besten drei Wörter seien »How« (75% geschätzte Wahrscheinlichkeit), »What« (3%) und »You« (1%). Das ist so weit unsere Short List. Als Nächstes erstellen wir drei Kopien unseres Modells und verwenden sie, um das nächste Wort für jeden Satz zu finden. Jedes Modell wird eine geschätzte Wahrscheinlichkeit pro Wort im Vokabular ausgeben. Das erste Modell wird versuchen, das nächste Wort im Satz »How« zu finden, und vielleicht wird es eine Wahrscheinlichkeit von 36% für das Wort »will«, 32% für »are« und 16% für »do« liefern und so weiter. Beachten Sie, dass dies eigentlich *bedingte* Wahrscheinlichkeiten sind, wenn der Satz mit »How« beginnt. Das zweite Modell wird versuchen, den Satz »What« zu vollenden – vielleicht liefert es eine bedingte Wahrscheinlichkeit von 50% für »are« und so weiter. Sofern das Vokabular aus 10.000 Wörtern besteht, wird jedes Modell 10.000 Wahrscheinlichkeiten ausgeben.

Als Nächstes berechnen wir die Wahrscheinlichkeiten für jeden der 30.000 Zwei-Wort-Sätze, die diese Modelle berücksichtigt haben (3×10.000). Dazu multiplizieren wir die geschätzten bedingten Wahrscheinlichkeiten jedes Worts mit den geschätzten Wahrscheinlichkeiten des Satzes, den es vervollständigt. So lag beispielsweise die geschätzte Wahrscheinlichkeit des Satzes »How« bei 75%, während die geschätzte bedingte Wahrscheinlichkeit für »will« (sofern das erste Wort »How« ist) bei 36% liegt, sodass die geschätzte Wahrscheinlichkeit für »How will« $75\% \times 36\% = 27\%$ beträgt. Nach dem Berechnen der Wahrscheinlichkeiten aller 30.000 Zwei-Wort-Sätze merken wir uns nur die besten drei. Vielleicht beginnen sie alle mit dem Wort »How«: »How will« (27%), »How are« (24%) und »How do« (12%). Aktuell würde »How will« gewinnen, aber »How are« ist noch nicht aus dem Rennen.

Dann wiederholen wir den gesamten Prozess: Wir nutzen drei Modelle zum Vorhersagen des nächsten Worts in jedem der drei Sätze und berechnen die Wahrscheinlichkeiten aller 30.000 Drei-Wort-Sätze, die wir berücksichtigen wollen. Vielleicht sind die Top Drei jetzt »How are you« (10%), »How do you« (8%) und »How will you« (2%). Im nächsten Schritt erhalten wir eventuell »How do you do« (7%), »How are you <eos>« (6%) und »How are you doing« (3%). Beachten Sie, dass »How will« jetzt entfernt wurde und wir drei sehr vernünftige Übersetzungen haben. Wir haben die Performance unseres Encoder-Decoder-Modells ohne zusätzliches Training verbessert, einfach durch einen sinnvoller Einsatz.

Sie können die Beam Search ziemlich einfach mithilfe der TensorFlow Addons implementieren:

```
beam_width = 10

decoder = tfa.seq2seq.beam_search_decoder.BeamSearchDecoder(
    cell=decoder_cell, beam_width=beam_width, output_layer=output_layer)

decoder_initial_state = tfa.seq2seq.beam_search_decoder.tile_batch(
    encoder_state, multiplier=beam_width)

outputs, _, _ = decoder(
    embedding_decoder, start_tokens=start_tokens, end_token=end_token,
```

```
initial_state=decoder_initial_state)
```

Wir erstellen zuerst einen BeamSearchDecoder, der alle Decoder-Klone verpackt (in diesem Fall 10). Dann erstellen wir eine Kopie des abschließenden Status des Encoders für jeden Decoder-Klon und übergeben diese Status an den Decoder zusammen mit den Start- und End-Token.

All das zusammen liefert gute Übersetzungen für recht kurze Sätze (insbesondere wenn Sie vortrainierte Word Embeddings verwenden). Leider ist dieses Modell bei langen Sätzen sehr schlecht. Wieder findet sich das Problem in dem beschränkten Kurzzeitgedächtnis von RNNs. *Attention-Mechanismen* sind hier die entscheidende Innovation, die dieses Problem angeht.

Attention-Mechanismen

Schauen Sie sich in [Abbildung 16-3](#) den Weg vom Wort »milk« zum Wort »lait« an – er ist ziemlich lang! Das heißt, dass eine Repräsentation dieses Worts (zusammen mit allen anderen Wörtern) über viele Schritte transportiert werden muss, bevor es tatsächlich zum Einsatz kommt. Können wir diesen Weg nicht abkürzen?

Das war die zentrale Idee in einem bahnbrechenden Artikel (<https://homl.info/attention>)¹³ aus dem Jahr 2014 von Dzmitry Bahdanau et al. Die Autoren stellen dort eine Technik vor, die es dem Decoder erlaubt, sich bei jedem Zeitschritt auf die passenden Wörter zu konzentrieren (vom Encoder codiert). So wird der Encoder beispielsweise bei dem Zeitschritt, bei dem er das Wort »lait« ausgeben muss, seine Aufmerksamkeit auf das Wort »milk« fokussieren. Der Weg von einem Eingabewort zu seiner Übersetzung ist jetzt viel kürzer, sodass die Beschränkungen des Kurzzeitgedächtnisses von RNNs weniger Auswirkungen haben. Attention-Mechanismen haben die neuronale maschinelle Übersetzung (und NLP im Allgemeinen) revolutioniert und deutliche Verbesserungen hervorgebracht – insbesondere für lange Sätze (über 30 Wörtern).¹⁴

In [Abbildung 16-6](#) sehen Sie die Architektur dieses Modells (ein bisschen vereinfacht, wie wir noch sehen werden). Links haben Sie den Encoder und den Decoder. Statt nur den finalen verborgenen Status an den Decoder zu schicken (was trotzdem geschieht, auch wenn es nicht in der Abbildung gezeigt wird), senden wir nun alle Ausgaben an den Decoder. Bei jedem Zeitschritt berechnet die Gedächtniszelle des Decoders eine gewichtete Summe dieser Encoder-Ausgaben. Damit wird festgelegt, auf welche Wörter er sich in diesem Schritt konzentriert. Das Gewicht $\alpha_{(t,i)}$ ist das Gewicht der i . Encoder-Ausgabe beim t . Decoder-Zeitschritt. Ist beispielsweise das Gewicht $\alpha_{(3,2)}$ viel größer als die Gewichte $\alpha_{(3,0)}$ und $\alpha_{(3,1)}$, wird der Decoder viel mehr Aufmerksamkeit auf das Wort mit der Nummer 2 (»milk«) als auf die anderen zwei Wörter legen – zumindest in diesem Zeitschritt. Der Rest des Decoders funktioniert wie zuvor: Bei jedem Zeitschritt empfängt die Gedächtniszelle die eben besprochenen Eingaben, dazu den verborgenen Status des vorherigen Zeitschritts und schließlich (wenn auch nicht in der Abbildung dargestellt) das Zielwort aus dem vorherigen Zeitschritt (oder beim Inferieren die Ausgabe des vorherigen Zeitschritts).

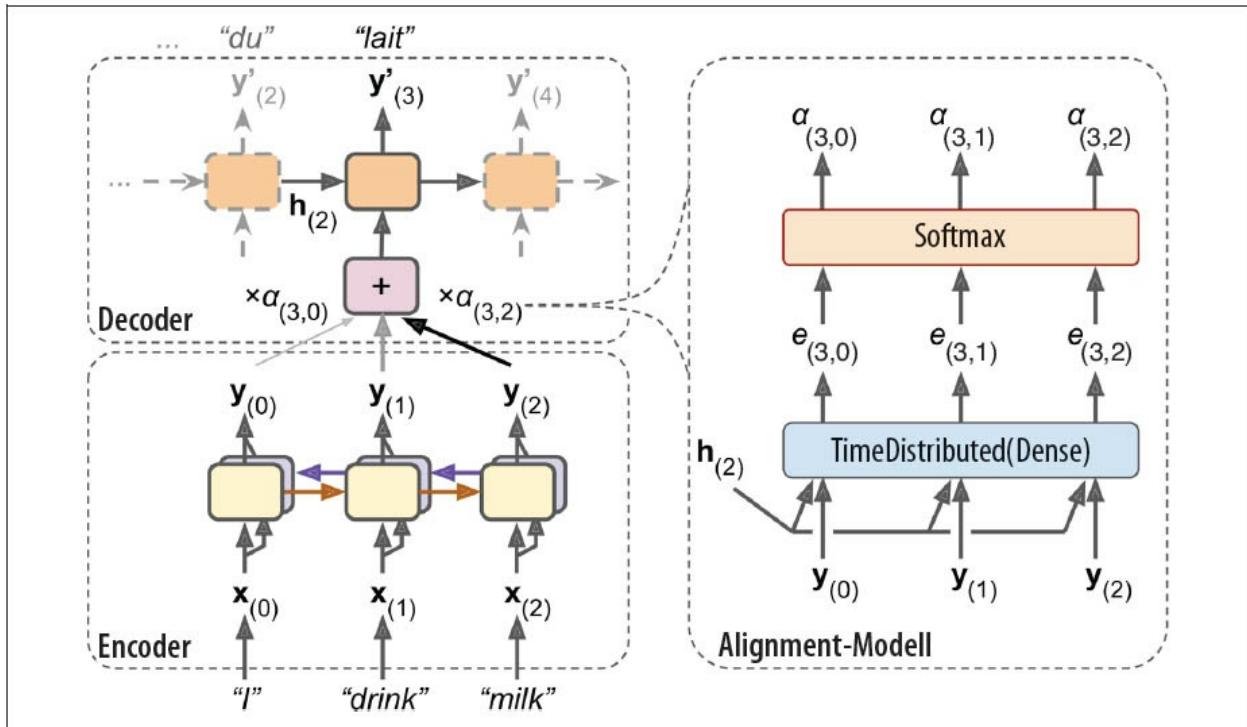


Abbildung 16-6: Neuronale maschinelle Übersetzung mit einem Encoder-Decoder-Netzwerk mit einem Attention-Modell

Aber woher kommen diese Gewichte $\alpha_{(t,i)}$? Das ist tatsächlich ziemlich einfach: Sie werden von einem kleinen neuronalen Netz erzeugt, das als *Alignment-Modell* (oder *Attention-Schicht*) bezeichnet und zusammen mit dem Rest des Encoder- Decoder-Modells trainiert wird. Dieses Alignment-Modell sehen Sie rechts in Abbildung 16-6. Es beginnt mit einer zeitverteilten Dense-Schicht¹⁵ mit einem einzelnen Neuron, das als Eingabe alle Encoder-Ausgaben erhält – verbunden mit dem vorherigen verborgenen Status des Decoders (zum Beispiel $h_{(2)}$). Diese Schicht gibt einen Score (oder eine Energie) für jede Decoder-Ausgabe aus (zum Beispiel $e_{(3,2)}$): Dieser Score misst, wie gut jede Ausgabe mit dem vorherigen verborgenen Status des Decoders übereinstimmt. Schließlich durchlaufen alle Scores eine Softmax- Schicht, um ein finales Gewicht für jede Encoder-Ausgabe zu erhalten (zum Beispiel $\alpha_{(3,2)}$). Alle Gewichte für einen gegebenen Decoder-Zeitschritt addieren sich zu 1 (da die Softmax-Schicht nicht zeitverteilt ist). Dieser spezielle Attention-Mechanismus wird als *Bahdanau-Attention* bezeichnet (benannt nach dem erstgenannten Autor des Artikels). Da er die Ausgabe des Encoders mit dem vorherigen verborgenen Status des Decoders verbindet, wird er manchmal auch als *Concatenative Attention* (oder *Additive Attention*) bezeichnet.

- ▀ Ist der Eingabesatz n Wörter lang und geht man davon aus, dass der Ausgabesatz ebenfalls etwa so lang ist, muss dieses Modell etwa n^2 Gewichte berechnen. Zum Glück ist diese quadratische Rechenkomplexität noch handhabbar, weil selbst lange Sätze nicht aus Tausenden von Wörtern bestehen.

Kurze Zeit später wurde in einem Artikel (<https://homl.info/luongattention>)¹⁶ aus dem Jahr 2015

von Minh-Thang Luong et al. ein anderer Attention-Mechanismus vorgeschlagen, der Verbreitung fand. Weil das Ziel des Attention-Mechanismus das Messen der Ähnlichkeit zwischen einer der Ausgaben des Encoders und dem vorherigen verborgenen Status des Decoders ist, haben die Autoren vorgeschlagen, einfach das *Skalarprodukt* dieser beiden Vektoren zu berechnen (siehe [Kapitel 4](#)), da dies häufig ein ziemlich guter Messwert für Ähnlichkeit ist und es von moderner Hardware viel schneller berechnet werden kann. Damit das möglich ist, müssen beide Vektoren die gleiche Dimensionalität haben. Das nennt sich *Luong-Attention* (hier erneut nach dem erstgenannten Autor des Artikels) oder manchmal auch *multiplikative Attention*. Das Skalarprodukt liefert einen Score, und alle Scores (eines gegebenen Decoder-Zeitschritts) durchlaufen eine Softmax-Schicht für die finalen Gewichte – so wie bei der Bahdanau-Attention. Eine andere vorgeschlagene Vereinfachung war der Einsatz des verborgenen Status des Decoders beim aktuellen Zeitschritt statt beim vorherigen Zeitschritt (also $\mathbf{h}_{(t)}$ statt $\mathbf{h}_{(t-1)}$), um dann die Ausgabe des Attention-Mechanismus (geschrieben als $\tilde{\mathbf{h}}_{(i)}$) direkt zum Berechnen der Vorhersagen des Decoders zu nutzen (statt sie zum Berechnen des aktuellen verborgenen Status des Decoders zu verwenden). Zudem haben sie eine Variante des Skalarprodukt-Mechanismus vorgeschlagen, bei der die Encoder-Ausgaben zuerst eine lineare Transformation durchlaufen (also eine zeitverteilte Dense-Schicht ohne einen Bias-Term), bevor die Skalarprodukte berechnet werden. Das nennt man den »allgemeinen« Skalarprodukt-Ansatz. Die Autoren haben beide Skalarprodukt-Varianten mit dem konkatenierenden Attention-Mechanismus verglichen (wobei ein reskalierender Parametervektor \mathbf{v} addiert wurde) und beobachtet, dass die Skalarprodukt-Varianten bessere Ergebnisse bringen als die konkatenierende Attention. Aus diesem Grund wird Letztere mittlerweile deutlich seltener eingesetzt. Die Gleichungen für diese drei Attention-Mechanismen sind in [Formel 16-1](#) zusammengefasst.

Formel 16-1: Attention-Mechanismen

$$\tilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)}$$

mit $\alpha_{(t,i)} = \frac{\exp(e_{(t,i)})}{\sum_i' \exp(e_{(t,i')})}$

und $e_{(t,i)} = \begin{cases} \mathbf{h}_{(t)}^\top \mathbf{y}_{(i)} & \text{Skalarprodukt} \\ \mathbf{h}_{(t)}^\top \mathbf{W} \mathbf{y}_{(i)} & \text{allgemein} \\ \mathbf{v}^\top \tanh(\mathbf{W}[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}]) & \text{concat} \end{cases}$

So können Sie die Luong-Attention zu einem Encoder-Decoder-Modell mithilfe der TensorFlow Addons hinzufügen:

```
attention_mechanism = tfa.seq2seq.attention_wrapper.LuongAttention(
    units, encoder_state, memory_sequence_length=encoder_sequence_length)
```

```
attention_decoder_cell = tfa.seq2seq.attention_wrapper.AttentionWrapper(  
    decoder_cell, attention_mechanism, attention_layer_size=n_units)
```

Wir verpacken die Decoder-Zelle einfach in einem `AttentionWrapper` und stellen den gewünschten Attention-Mechanismus bereit (in diesem Beispiel die Luong- Attention).

Visuelle Attention

Attention-Mechanismen werden mittlerweile aus einer Reihe von Gründen eingesetzt. Eine der ersten Anwendungen außerhalb des NMT war das Erzeugen von Bildunterschriften mithilfe der visuellen Attention (<https://homl.info/visualattention>):¹⁷ Zuerst verarbeitet ein Convolutional Neural Network das Bild und gibt ein paar Feature Maps aus, dann erzeugt ein mit einem Attention-Mechanismus ausgestattetes Decoder-RNN die Bildunterschrift – jeweils ein Wort nach dem anderen. Bei jedem Decoder-Zeitschritt (also bei jedem Wort) nutzt der Decoder das Attention-Modell, um sich auf den richtigen Teil des Bilds zu fokussieren. So hat das Modell beispielsweise in [Abbildung 16-7](#) die Bildunterschrift »A woman is throwing a frisbee in a park« erzeugt, und Sie können sehen, auf welchen Teil des Eingabebilds der Decoder seine Aufmerksamkeit gerichtet hat, als er das Wort »frisbee« ausgegeben hat.



*Abbildung 16-7: Visuelle Attention: ein Eingabebild (links) und der Fokus des Modells vor dem Erzeugen des Worts »frisbee« (rechts)*¹⁸

Explainability

Ein zusätzlicher Vorteil der Attention-Mechanismen ist, dass sie es leichter machen, zu verstehen, was das Modul zu seiner Ausgabe geführt hat. Das nennt sich *Explainability*. Sie kann insbesondere dann nützlich sein, wenn das Modell einen Fehler macht. Wird beispielsweise ein Bild mit einem Hund, der im Schnee läuft, als »Wolf, der im Schnee läuft« beschriftet, können Sie zurückgehen und prüfen, worauf sich das Modell fokussiert hat, als es das Wort »Wolf« ausgab. Vielleicht finden Sie heraus, dass es nicht nur auf den

Hund, sondern auch auf den Schnee geachtet hat, was auf eine mögliche Erklärung hindeutet: Eventuell hat das Modell gelernt, zwischen Hunden und Wölfen zu unterscheiden, indem es prüft, ob es viel Schnee zu sehen gibt. Sie können das dann beheben, indem Sie das Modell mit mehr Bildern von Wölfen ohne Schnee und Hunden mit Schnee trainieren. Dieses Beispiel entstammt einem großartigen Artikel (<https://hml.info/explainclass>)¹⁹ aus dem Jahr 2016 von Marco Túlio Ribeiro et al., der bei der Explainability einen anderen Ansatz verfolgt, indem ein interpretierbares Modell lokal rund um die Vorhersage eines Klassifizierers lernt.

In manchen Anwendungen ist Explainability nicht nur ein Werkzeug, um ein Modell zu debuggen – es kann eine rechtliche Anforderung sein (denken Sie an ein System, das entscheidet, ob Sie einen Kredit erhalten dürfen oder nicht).

Attention-Mechanismen sind so leistungsfähig, dass Sie tatsächlich nur damit State-of-the-Art-Modelle bereits bauen können.

Attention Is All You Need: Die Transformer-Architektur

In einem bahnbrechenden Artikel (<https://hml.info/transformer>)²⁰ aus dem Jahr 2017 hat ein Forscherteam von Google vorgeschlagen, dass »Attention Is All You Need«. Sie haben es geschafft, eine Architektur namens *Transformer* zu erzeugen, die den Stand der Technik im Bereich NMT deutlich verbessert hat, ohne rekurrente oder Convolutional-Schichten einzusetzen,²¹ sondern nur Attention-Mechanismen (plus Embedding-Schichten, Dense-Schichten, Normalisierungsschichten und ein paar anderen Dingen). Zusätzlich ließ sich diese Architektur viel schneller trainieren und leichter parallelisieren, daher konnten die Forscher sie in einem Bruchteil der Zeit und Kosten vorheriger State-of-the-Art-Modelle trainieren.

Die Transformer-Architektur ist in [Abbildung 16-8](#) dargestellt.

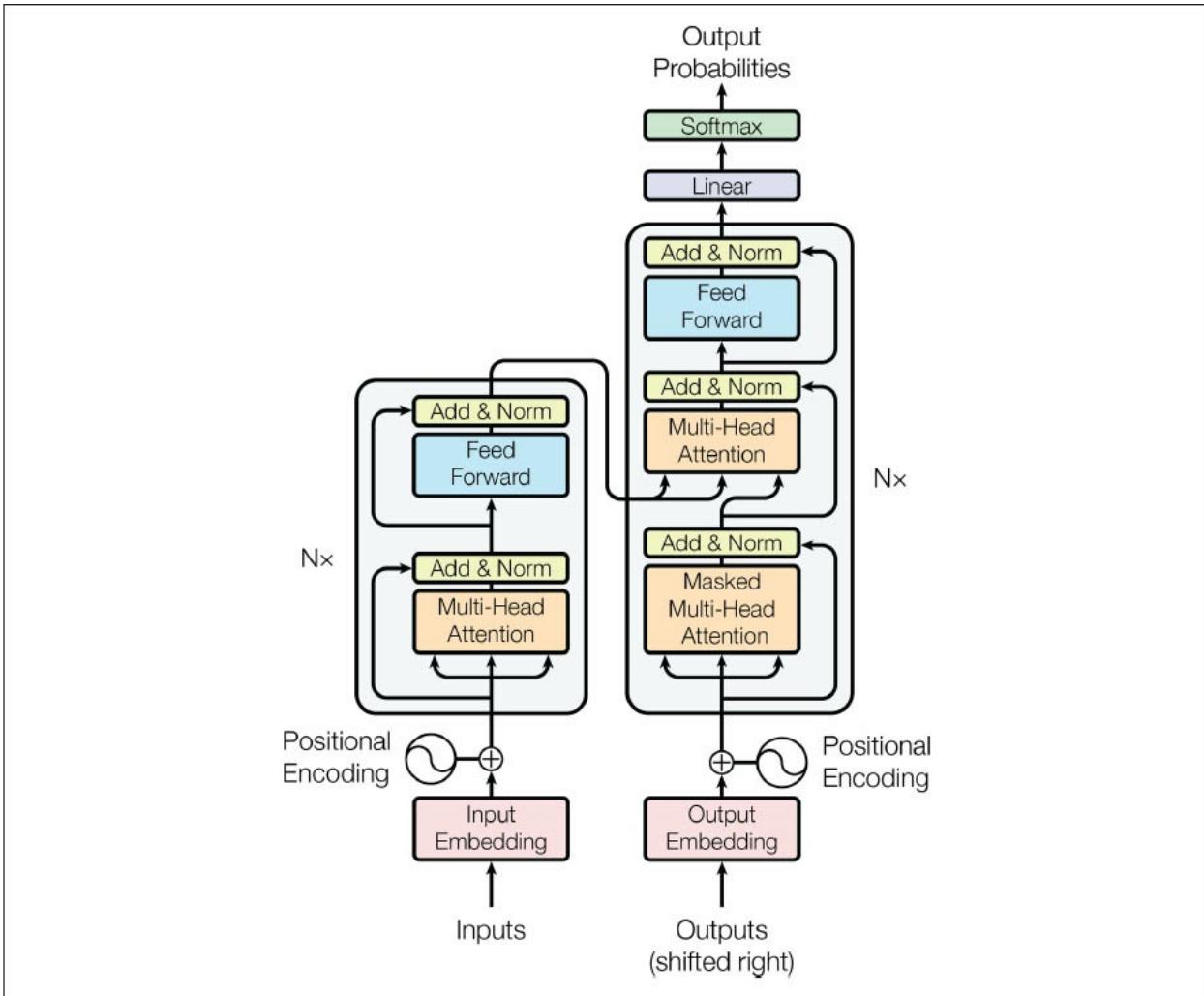


Abbildung 16-8: Die Transformer-Architektur²²

Schauen wir uns diese Abbildung genauer an:

- Auf der linken Seite findet sich der Encoder. Wie zuvor übernimmt er als Eingabe einen Batch mit Sätzen, die als Sequenzen von Wort-IDs umgesetzt sind (die Eingabeform ist [Batchgröße, maximale Eingabesatzlänge]), und er codiert jedes Wort in eine 512-dimensionale Repräsentation (sodass die Ausgabeform des Encoders [Batchgröße, maximale Eingabesatzlänge, 512] ist). Beachten Sie, dass der obere Teil des Encoders N Mal gestackt ist (im Artikel ist $N = 6$).
- Auf der rechten Seite findet sich der Decoder. Während des Trainings übernimmt er den Zielsatz als Eingabe (ebenfalls umgesetzt als Sequenz von Wort-IDs), um einen Zeitschritt nach rechts verschoben (es wird also ein Start-of-Sequence-Token am Anfang eingefügt). Zudem übernimmt er die Ausgaben des Encoders (die Pfeile von der linken Seite). Beachten Sie, dass der obere Teil des Decoders ebenfalls N Mal gestackt ist und die finalen Ausgaben des Encoder-Stacks auf jeder dieser N Ebenen an den Decoder übergeben werden. Wie zuvor gibt der Decoder bei jedem Zeitschritt eine Wahrscheinlichkeit für jedes nächste Wort aus (seine Ausgabeform ist [Batchgröße,

maximale Ausgabesatzlänge, Vokabulargröße]).

- Während der Inferenz können dem Decoder keine Ziele übergeben werden, daher füttern wir ihn dann mit dem zuvor ausgegebenen Wörtern (beginnend mit einem Start-of-Sequence-Token). Daher muss das Modell wiederholt aufgerufen werden, wobei jeweils ein weiteres Wort vorhergesagt wird (das in der nächsten Runde an den Decoder übergeben wird, bis das End-of-Sequence-Token ausgegeben wird).
- Wenn Sie genauer hinschauen, können Sie sehen, dass Sie mit den meisten Komponenten schon vertraut sind: Es gibt zwei Embedding-Schichten, $5 \times N$ Skip-Verbindungen, auf die jeweils eine Normalisierungsschicht folgt, $2 \times N$ »Feed Forward«-Module, die jeweils aus zwei Dense-Schichten bestehen (die erste nutzt die ReLU-Aktivierungsfunktion, die zweite hat keine solche Funktion), und schließlich ist die Ausgabeschicht eine Dense-Schicht mit der Softmax-Aktivierungsfunktion. Alle diese Schichten sind zeitverteilt, daher wird jedes Wort unabhängig von allen anderen Wörtern behandelt. Aber wie können wir einen Satz übersetzen, wenn wir uns immer nur ein Wort gleichzeitig anschauen? Nun, hier kommen die neuen Komponenten ins Spiel:
 - Die *Multi-Head-Attention*-Schicht des Encoders codiert die Beziehung jedes Worts mit jedem anderen Wort im gleichen Satz, wobei sie mehr Aufmerksamkeit auf die relevantesten legt. So hängt beispielsweise die Ausgabe dieser Schicht für das Wort »Queen« im Satz »They welcomed the Queen of the United Kingdom« von allen Wörtern im Satz ab, aber es wird vermutlich mehr Aufmerksamkeit auf die Wörter »United« und »Kingdom« gelegt als auf »They« oder »welcomed«. Dieser Attention-Mechanismus wird als *Self-Attention* bezeichnet (der Satz achtet auf sich selbst). Wir werden gleich genauer darüber reden, wie er funktioniert. Die *Masked Multi-Head-Attention*-Schichten des Decoders machen das Gleiche, aber jedes Wort darf nur auf Wörter achten, die vor ihm liegen. Und schließlich achtet die obere Multi-Head-Attention-Schicht des Decoders auf die Wörter im Eingabesatz. So wird der Decoder beispielsweise vermutlich im Eingabesatz genau auf das Wort »Queen« achten, wenn es darum geht, die Übersetzung dieses Worts auszugeben.
 - Die *Positional Encodings* sind einfache Dense-Vektoren (ähnlich den Word Embeddings), die die Position eines Worts im Satz repräsentieren. Das n . Positional Encoding wird zum Word Embedding des n . Worts in jedem Satz hinzugefügt. Damit erhält das Modell Zugriff auf die Position jedes Worts, was erforderlich ist, weil die Multi-Head-Attention-Schichten die Reihenfolge der Position der Wörter nicht berücksichtigt – sie schauen nur auf deren Beziehungen. Da alle anderen Schichten zeitverteilt sind, können sie die Position jedes Worts nicht kennen (weder relativ noch absolut). Offensichtlich sind die relativen und absoluten Wortpositionen wichtig, daher müssen wir diese Informationen irgendwie an den Transformer übergeben, und Positional Encodings sind dazu eine gute Möglichkeit.

Schauen wir uns diese beiden neuen Komponenten der Transformer-Architektur etwas genauer an und beginnen wir mit den Positional Encodings.

Positional Encodings

Ein Positional Encoding ist ein Dense-Vektor, der die Position eines Worts in einem Satz codiert: Das i . Positional Encoding wird einfach zum Word Embedding des i . Worts im Satz addiert. Diese Positional Encodings können vom Modell gelernt werden, aber im Artikel haben es die Autoren bevorzugt, feste Positional Encodings zu verwenden, die über die Sinus- und Kosinus-Funktionen verschiedener Frequenzen definiert wurden. Die Positional-Embedding-Matrix \mathbf{P} ist in [Formel 16-2](#) definiert und (transponiert) unten in [Abbildung 16-9](#) dargestellt, wobei $P_{p,i}$ die i . Komponente des Embedding für das Wort an der p . Position im Satz ist.

Formel 16-2: Sinus und Kosinus für Positional Encodings

$$P_{p,2i} = \sin(p / 10000^{2i/d})$$

$$P_{p,2i+1} = \cos(p / 10000^{2i/d})$$

Diese Lösung liefert die gleiche Performance wie gelernte Positional Encodings, kann aber auf beliebig lange Sätze erweitert werden, weshalb sie bevorzugt wird. Nachdem die Positional Encodings zu den Word Embeddings addiert wurden, hat der Rest des Modells Zugriff auf die absolute Position jedes Worts im Satz, weil es für jede Position ein eindeutiges Positional Encoding gibt (beispielsweise wird das Positional Encoding für das Wort an der 22. Position in einem Satz durch die vertikale gestrichelte Linie unten links in [Abbildung 16-9](#) dargestellt, und Sie können sehen, dass es an dieser Stelle eindeutig ist). Zudem ermöglicht die Wahl der oszillierenden Funktionen (Sinus und Kosinus) dem Modell, auch relative Positionen zu lernen. So haben beispielsweise Wörter, die 38 Wörter auseinanderliegen (zum Beispiel an den Positionen $p = 22$ und $p = 60$), immer die gleichen Werte für die Positional Encodings in den Embedding-Dimensionen $i = 100$ und $i = 101$, wie Sie in [Abbildung 16-9](#) sehen. Das erklärt, warum wir für jede Frequenz Sinus und Kosinus benötigen: Würden wir nur den Sinus verwenden (die blaue Welle bei $i = 100$), würde das Modell nicht zwischen den Positionen $p = 25$ und $p = 35$ (mit einem Kreuz markiert) unterscheiden können.

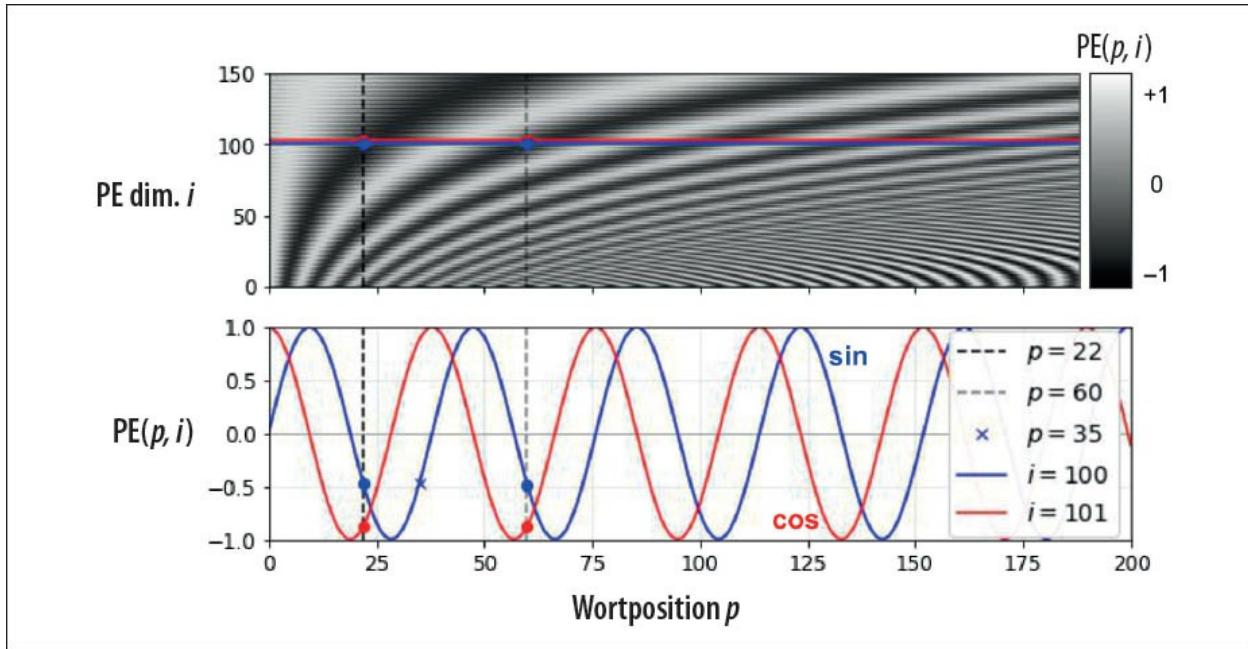


Abbildung 16-9: Sinus/Kosinus-Positional-Encoding-Matrix (transponiert, oben) mit einem Fokus auf zwei Werte von i (unten)

Es gibt in TensorFlow keine PositionalEncoding-Schicht, aber es lässt sich leicht eine erstellen. Aus Gründen der Effizienz berechnen wir die Positional-Embedding- Matrix schon im Konstruktor (daher müssen wir die maximale Satzlänge `max_steps` und die Anzahl der Dimensionen für jede Wortrepräsentation `max_dims` kennen). Dann beschneidet die Methode `call()` diese Embedding-Matrix auf die Größe der Eingaben und addiert sie zu den Eingaben. Da wir beim Erstellen der Positional- Embedding-Matrix eine zusätzliche Dimension der Größe 1 hinzugefügt haben, stellen die Verteilungsregeln sicher, dass die Matrix zu jedem Satz in den Eingaben addiert wird:

```
class PositionalEncoding(keras.layers.Layer):

    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)

        if max_dims % 2 == 1: max_dims += 1 # max_dims must be even

        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))

        pos_emb = np.empty((1, max_steps, max_dims))

        pos_emb[0, :, ::2] = np.sin(p / 10000**((2 * i) / max_dims)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**((2 * i) / max_dims)).T

        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))

    def call(self, inputs):
```

```

shape = tf.shape(inputs)

return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]

```

Dann können wir die erste Schicht des Transformers erzeugen:

```

embed_size = 512; max_steps = 500; vocab_size = 10000

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)

embeddings = keras.layers.Embedding(vocab_size, embed_size)

encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)

positional_encoding = PositionalEncoding(max_steps, max_dims=embed_size)

encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)

```

Schauen wir uns nun den Kern des Transformer-Modells genauer an – die Multi-Head-Attention-Schicht.

Multi-Head Attention

Um zu verstehen, wie eine Multi-Head-Attention-Schicht funktioniert, müssen wir erst die *Skaled Dot-Product- Attention-Schicht* verstehen, auf der sie basiert. Nehmen wir an, dass der Encoder den Eingabesatz »They played chess« analysiert und es geschafft hat, zu verstehen, dass das Wort »They« das Subjekt und das Wort »played« das Verb ist. Also hat er diese Information in den Repräsentationen dieser Wörter codiert. Nehmen wir nun weiter an, dass der Decoder das Subjekt schon übersetzt hat und davon ausgeht, als Nächstes das Verb zu übersetzen. Dazu muss er sich das Verb aus dem Eingabesatz holen. Dies ist analog zu einem Dictionary-Lookup: als hätte der Encoder ein Dictionary `{"subject": "They", "verb": "played", ...}` erstellt und der Decoder suche nach dem Wert, der zum Schlüssel »verb« gehört. Aber das Modell hat keine eindeutigen Token, um die Schlüssel zu repräsentieren (wie »subject« oder »verb«) – es besitzt vektorisierte Repräsentationen dieser Konzepte (die es während des Trainings gelernt hat), daher wird der Schlüssel für den Lookup (die *Query*) nicht perfekt auf einen Schlüssel im Dictionary passen. Die Lösung ist, einen Ähnlichkeitswert zwischen der Query und jedem Schlüssel im Dictionary zu berechnen und dann diese Ähnlichkeitswerte mit der Softmax-Funktion in Gewichte umzuwandeln, die sich zu 1 aufaddieren. Ähnelt der Schlüssel, der das Verb repräsentiert, der Query stark, wird das Gewicht des Schlüssels nahe 1 sein. Dann kann das Modell eine gewichtete Summe der korrespondierenden Werte berechnen, und wenn das Gewicht des Schlüssels »verb« nahe 1 ist, wird die gewichtete Summe sehr nahe an der Repräsentation des Worts »played« liegen. Kurz gesagt, können Sie sich den gesamten Prozess

als differenzierbaren Dictionary-Lookup vorstellen. Der vom Transformer genutzte Ähnlichkeitswert ist wie bei der Luong-Attention einfach das Skalarprodukt. Tatsächlich ist die Gleichung abgesehen von einem Skalierungsfaktor sogar dieselbe. Sie finden sie in vektorisierter Form in [Formel 16-3](#).

Formel 16-3: Scaled Dot-Product Attention

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{Schlüssel}}}}\right)\mathbf{V}$$

In dieser Gleichung gilt:

- **Q** ist eine Matrix mit einer Zeile pro Query. Ihre Form ist $[n_{\text{Queries}}, d_{\text{Schlüssel}}]$, wobei n_{Queries} die Anzahl an Queries und $d_{\text{Schlüssel}}$ die Anzahl an Dimensionen jeder Query und jedes Schlüssels ist.
- **K** ist eine Matrix mit einer Zeile pro Schlüssel. Ihre Form ist $[n_{\text{Schlüssel}}, d_{\text{Schlüssel}}]$, wobei $n_{\text{Schlüssel}}$ die Anzahl an Schlüsseln und Werten ist.
- **V** ist eine Matrix mit einer Zeile pro Wert. Ihre Form ist $[n_{\text{Schlüssel}}, d_{\text{Werte}}]$, wobei d_{Werte} die Anzahl der Dimensionen jedes Werts ist.
- Die Form von $\mathbf{Q}\mathbf{K}^T$ ist $[n_{\text{Queries}}, n_{\text{Schlüssel}}]$: Sie enthält einen Ähnlichkeitswert für jedes Schlüssel-Wert-Paar. Die Ausgabe der Softmax-Funktion hat die gleiche Form, aber alle Zeilen addieren sich zu 1. Die finale Ausgabe besitzt die Form $[n_{\text{Queries}}, d_{\text{Werte}}]$: Es gibt eine Zeile pro Query, und jede Zeile repräsentiert das Query-Ergebnis (eine gewichtete Summe der Werte).
- Der Skalierungsfaktor skaliert die Ähnlichkeitswerte herunter, um eine Sättigung der Softmax-Funktion zu vermeiden, weil dies zu sehr kleinen Gradienten führen würde.
- Es ist möglich, einige der Schlüssel-Wert-Paare zu maskieren, indem direkt vor dem Berechnen des Softmax zu den entsprechenden Ähnlichkeitswerten ein sehr großer negativer Wert addiert wird. Das ist in der Masked Multi-Head- Attention-Schicht nützlich.

Im Encoder wird diese Gleichung auf jeden Eingabesatz im Batch angewendet, wobei **Q**, **K** und **V** alle gleich der Liste der Wörter im Eingabesatz sind (sodass jedes Wort im Satz mit jedem Wort im gleichen Satz verglichen wird, einschließlich mit sich selbst). Genauso wird in der Masked-Attention-Schicht im Decoder die Gleichung auf jeden Zielsatz im Batch angewendet, wobei **Q**, **K** und **V** alle gleich der Liste der Wörter im Zielsatz sind, dieses Mal aber unter Verwendung einer Maske, um zu verhindern, dass ein Wort mit Wörtern nach ihm verglichen wird (zur Inferenzzeit hat der Decoder nur Zugriff auf die schon ausgegebenen Wörter, nicht aber auf zukünftige Wörter, da müssen wir während des Trainings zukünftige Ausgabeketten maskieren). In der oberen Attention-Schicht des Decoders sind die Schlüssel **K** und die Werte **V** einfach die Liste der Word Encodings, die vom Encoder produziert wurden, während die Queries **Q** die Liste der durch den Decoder erzeugten Word Encodings ist.

Die Schicht `keras.layers.Attention` implementiert die Scaled Dot-Product Attention und

wendet dabei effizient [Formel 16-3](#) an, um die Sätze in einem Batch zu multiplizieren. Ihre Eingaben sind einfach wie **Q**, **K** und **V**, nur dass sie eine zusätzliche Batchdimension enthalten (die erste Dimension).

Sind in TensorFlow **A** und **B** Tensoren mit mehr als zwei Dimensionen – zum Beispiel mit den Formen [2, 3, 4, 5] und [2, 3, 5, 6] –, behandelt `tf.matmul(A, B)` diese Tensoren als 2×3 -Arrays, bei denen jede Zelle eine Matrix enthält, und wird die entsprechenden Matrizen multiplizieren: Die Matrix in der i . Zeile und j . Spalte in **A** wird mit der Matrix in der i . Zeile und j . Spalte in **B** multipliziert. Da das Produkt einer 4×5 -Matrix mit einer 5×6 -Matrix eine 4×6 -Matrix ist, wird `tf.matmul(A, B)` ein Array der Form [2, 3, 4, 6] zurückgeben.

Ignorieren wir die Skip-Verbindungen, die Normalisierungsschichten, die Feed-Forward-Blöcke und die Tatsache, dass dies eine Scaled Dot-Product Attention und nicht genau eine Multi-Head Attention ist, kann der Rest des Transformer-Modells wie folgt implementiert werden:

```
Z = encoder_in

for N in range(6):
    Z = keras.layers.Attention(use_scale=True)([Z, Z])

encoder_outputs = Z

Z = decoder_in

for N in range(6):
    Z = keras.layers.Attention(use_scale=True, causal=True)([Z, Z])
    Z = keras.layers.Attention(use_scale=True)([Z, encoder_outputs])

outputs = keras.layers.TimeDistributed(
    keras.layers.Dense(vocab_size, activation="softmax"))(Z)
```

Das Argument `use_scale=True` erzeugt einen zusätzlichen Parameter, mit dem die Schicht lernen kann, wie sie die Ähnlichkeitswerte sauber herunterrechnen kann. Das ist ein bisschen anders als beim Transformer-Modell, das die Ähnlichkeitswerte immer um den gleichen Faktor ($\sqrt{d_{\text{Schlüssel}}}$) reduziert. Das Argument `causal=True` beim Erstellen der zweiten Attention-Schicht sorgt dafür, dass jedes Ausgabe-Token nur auf vorherige Ausgabe-Tokens achtet, aber nicht auf zukünftige.

Schauen wir uns jetzt den letzten Baustein an: Was ist eine Multi-Head-Attention-Schicht? Ihre Architektur sehen Sie in [Abbildung 16-10](#).

Wie Sie sehen, handelt es sich nur um einen Haufen Scaled- Dot-Product-Attention-Schichten, vor denen jeweils eine lineare Transformation der Werte, Schlüssel und Queries steht (also eine zeitverteilte Dense-Schicht ohne Aktivierungsfunktion). Alle Ausgaben werden einfach

konkateniert und durchlaufen dann eine abschließende lineare Transformation (wieder zeitverteilt). Aber warum? Was ist der Sinn hinter dieser Architektur? Nun, denken wir noch mal an das Wort »played«, über das wir zuvor gesprochen haben (im Satz »They played chess«). Der Encoder war schlau genug, die Tatsache zu codieren, dass es sich um ein Verb handelt. Aber die Wortdarstellung enthält dank der Positional Encodings auch ihre Position im Text und vermutlich noch viele andere Merkmale, die für seine Übersetzung nützlich sind, wie zum Beispiel die Tatsache, dass es in der Vergangenheit steht. Kurz gesagt, codiert die Wortdarstellung viele verschiedene Eigenschaften des Worts. Würden wir nur eine einzelne Scaled-Dot-Product-Attention-Schicht nutzen, könnten wir all diese Eigenschaften nur einmal abfragen. Darum wendet die Multi-Head-Attention-Schicht mehrere verschiedene lineare Transformationen der Werte, Schlüssel und Queries an: So kann das Modell viele verschiedene Projektionen der Wortdarstellung in unterschiedlichen Subräumen anwenden, die sich jeweils auf eine Untergruppe der Eigenschaften der Wörter fokussieren. Vielleicht projiziert eine der linearen Schichten die Wortdarstellung in einen Subraum, in dem nur die Information übrig bleibt, dass es sich bei dem Wort um ein Verb handelt, während eine andere lineare Schicht die Tatsache extrahiert, dass es in der Vergangenheit steht, und so weiter. Dann implementieren die Scaled-Dot-Product-Attention-Schichten die Lookup-Phase, und schließlich konkatenieren wir alle Ergebnisse und projizieren sie wieder zurück in den ursprünglichen Raum.

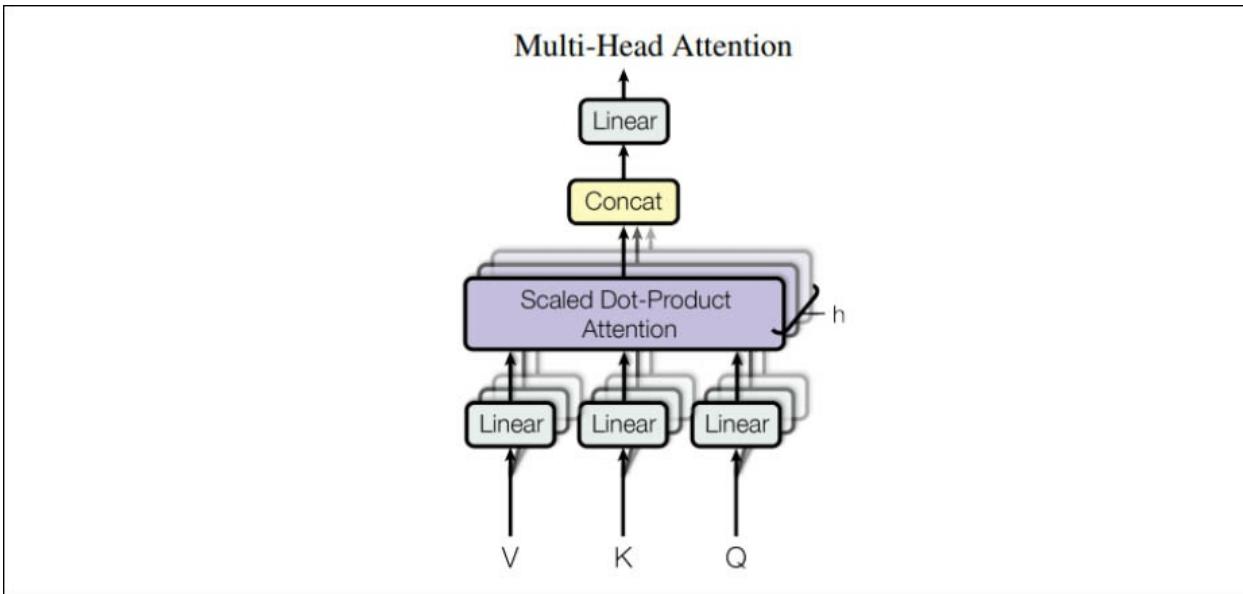


Abbildung 16-10: Architektur einer Multi-Head-Attention-Schicht²³

Aktuell gibt es keine Klasse Transformer oder MultiHeadAttention in TensorFlow 2. Aber Sie können sich das großartige TensorFlow-Tutorium (<https://homl.info/transformertuto>) für das Bauen eines Transformer-Modells zum Sprachverständnis anschauen. Zudem portiert das TF-Hub-Team zurzeit eine Reihe von auf Transformer basierenden Modulen nach TensorFlow 2, die bald zur Verfügung stehen sollten. Bis dahin habe ich Ihnen hoffentlich gezeigt, dass es nicht so schwer ist, selbst einen Transformer zu implementieren. Zudem ist es eine tolle Übungsaufgabe!

Aktuelle Entwicklungen bei Sprachmodellen

Das Jahr 2018 wurde als »ImageNet-Moment für NLP« bezeichnet: Die Fortschritte waren erstaunlich, und es gab immer größere LSTM- und Transformerbasierte Architekturen, die mit riesigen Datensätzen trainiert wurden. Ich empfehle Ihnen die folgenden Artikel, die alle 2018 veröffentlicht wurden:

- Der ELMo-Artikel (<https://homl.info/elmo>)²⁴ von Matthew Peters hat Embeddings from Language Models (ELMo) vorgestellt: Dabei handelt es sich um kontextualisierte Word Embeddings, die aus den internen Status eines tiefen, bidirektionalen Sprachmodells erlernt wurden. So wird beispielsweise das Wort »queen« in »Queen of the United Kingdom« und »queen bee« nicht die gleichen Embeddings besitzen.
- Der ULMFiT-Artikel (<https://homl.info/ulmfit>)²⁵ von Jeremy Howard und Sebastian Ruder hat die Effektivität unüberwachten Vortrainings für NLP-Aufgaben gezeigt: Die Autoren haben ein LSTM-Sprachmodell mit selbstüberwachtem Lernen an einem riesigen Textkorpus trainiert (also die Labels automatisch aus den Daten generiert) und es dann mit verschiedenen Aufgaben optimiert. Ihr Modell schlug den State-of-the-Art bei sechs Textklassifikationsaufgaben deutlich (die Fehlerrate wurde in den meisten Fällen um 18 bis 24% verringert). Zudem haben sie gezeigt, dass sie durch ein Anpassen des vortrainierten Modells an nur 100 gelabelten Beispielen die gleiche Performance erreichen konnten wie ein Modell, das von Grund auf mit 10.000 Beispielen trainiert wurde.
- Der GPT-Artikel (<https://homl.info/gpt>)²⁶ von Alec Radford und anderen OpenAI-Forschern hat ebenfalls die Effektivität von unüberwachtem Vortraining demonstriert, hier aber mit einer Transformer-ähnlichen Architektur. Die Autoren haben eine große, aber recht einfache Architektur aus einem Stack mit zwölf Transformer-Modulen (nur unter Verwendung von Masked-Multi-Head-Attention-Schichten) an einem großen Datensatz trainiert – ebenfalls mit selbstüberwachtem Lernen. Dann haben sie das Modell mit einer Reihe von Sprachaufgaben optimiert und dabei bei jeder Aufgabe nur kleinere Anpassungen vorgenommen. Die Aufgaben waren ziemlich verschieden: unter anderem Textklassifikation, Entailment (ob aus Satz A dann Satz B folgt)²⁷, Ähnlichkeit (zum Beispiel ähnelt »Nice weather today« dem Satz »It is sunny«) und das Beantworten von Fragen (mit ein paar Absätzen als Kontext muss das Modell Multiple-Choice-Fragen beantworten). Nur ein paar Monate später veröffentlichten Alec Radford, Jeffrey Wu und andere OpenAI-Forscher im Februar 2019 den GPT-2-Artikel (<https://homl.info/gpt2>)²⁸, der eine sehr ähnliche Architektur vorschlug, die größer ist (mit über 1,5 Milliarden Parametern!), und sie zeigten, dass sie eine gute Leistung bei vielen Aufgaben erreichen kann, ohne angepasst werden zu müssen. Das wird als *Zero-Shot Learning* (ZSL) bezeichnet. Eine kleinere Version des GPT-2-Modells (mit »nur« 117 Millionen Parametern) steht zusammen mit ihren vortrainierten Gewichten unter <https://github.com/openai/gpt-2> zur Verfügung.
- Der BERT-Artikel (<https://homl.info/bert>)²⁹ von Jacob Devlin und anderen Google-Forschern hat ebenfalls die Effektivität von selbstüberwachtem Vortraining mit einem großen Korpus gezeigt. Dabei kam eine ähnliche Architektur wie bei GPT zum Einsatz,

hier aber mit nicht maskierten Multi-Head-Attention-Schichten (wie im Encoder des Transformers). Dadurch ist das Modell ganz natürlich bidirektional (und daher das B in BERT – *Bidirectional Encoder Representations from Transformers*). Am wichtigsten ist, dass die Autoren zwei Vortrainingsaufgaben vorgestellt haben, die einen Großteil der Stärken des Modells erklären:

Masked Language Model (MLM)

Jedes Wort in einem Satz hat eine Wahrscheinlichkeit von 15%, maskiert zu werden, und das Modell wird darauf trainiert, die maskierten Wörter vorherzusagen. Ist der ursprüngliche Satz beispielsweise »She had fun at the birthday party«, erhält das Modell vielleicht den Satz »She <mask> fun at the <mask> party« und muss die Wörter »had« und »birthday« vorhersagen (die anderen Ausgaben werden ignoriert). Genauer gesagt, hat jedes ausgewählte Wort eine Chance von 80%, maskiert zu werden, mit 10% Wahrscheinlichkeit wird es durch ein zufälliges Wort ersetzt (um die Diskrepanz zwischen Vortraining und Anpassen zu verringern, da das Modell beim Anpassen keine <mask>-Token sehen wird), und mit 10% wird es einfach stehen gelassen (um das Modell zur richtigen Antwort zu leiten).

Next Sentence Prediction (NSP)

Das Modell wird darauf trainiert, vorherzusagen, ob zwei Sätze aufeinanderfolgen oder nicht. So sollte es beispielsweise vorhersagen, dass »The dog sleeps« und »It snores loudly« aufeinanderfolgende Sätze sind, während »The dog sleeps« und »The Earth orbits the Sun« nicht aufeinanderfolgen. Das ist eine herausfordernde Aufgabe, und sie verbessert die Leistung des Modells deutlich, wenn es für Aufgaben wie das Beantworten von Fragen oder Entailment angepasst wird.

Wie Sie sehen, drehen sich die wichtigsten Fortschritte in den Jahren 2018 und 2019 um eine bessere Subwort-Tokenisierung, den Wechsel von LSTMs hin zu Transformern und das Vortrainieren universeller Sprachmodelle durch selbstüberwachtes Lernen und das folgende Anpassen der Modelle durch nur wenige architektonische Änderungen (oder auch gar keine). Die Dinge sind in Bewegung, und niemand kann sagen, welche Architekturen das nächste Jahr bringt. Heutzutage sind es ganz klar Transformers, aber morgen vielleicht CNNs (schauen Sie sich zum Beispiel den Artikel (<https://homl.info/pervasiveattention>)³⁰ aus dem Jahr 2018 von Maha Elbayad et al. an, bei dem Forscher maskierte zweidimensionale Convolutional Layers für Sequence-to-Sequence-Aufgaben nutzen). Oder eventuell gelingt RNNs ein überraschendes Comeback (siehe beispielsweise den Artikel (<https://homl.info/indrnn>)³¹ aus dem Jahr 2018 von Shuai Li et al., der zeigt, dass es durch voneinander unabhängige Neuronen in einer RNN-Schicht möglich ist, viel tiefere RNNs zu trainieren, die viel längere Sequenzen lernen können).

Im nächsten Kapitel werden wir besprechen, wie man tiefe Repräsentationen unüberwacht mit Autoencodern lernen kann, und wir werden Generative Adversarial Networks (GANs) nutzen, um Bilder und anderes zu erzeugen!

Übungen

1. Was sind die Vor- und Nachteile eines zustandsbehafteten RNN im Vergleich zu einem

zustandslosen RNN?

2. Warum nutzen die Menschen Encoder-Decoder-RNNs statt die schlichten Sequence-to-Sequence-RNNs zur automatischen Übersetzung?
3. Wie können Sie mit Eingabesequenzen unterschiedlicher Länge umgehen? Wie ist es mit verschiedenen langen Ausgabesequenzen?
4. Was ist Beam Search, und warum sollten Sie es verwenden? Welche Tools können Sie zum Implementieren einsetzen?
5. Was ist ein Attention-Mechanismus? Wie hilft er?
6. Was ist die wichtigste Schicht in der Transformer-Architektur? Was ist ihr Zweck?
7. Wann werden Sie Sampled Softmax nutzen müssen?
8. Hochreiter und Schmidhuber haben in ihrem Artikel (<https://homl.info/93>) über LSTMs *Embedded Reber Grammars* verwendet. Dabei handelt es sich um künstliche Grammatiken, die Strings wie »BPBTSXXVPSEPE« erzeugen. Schauen Sie sich Jenny Orrs nette Einführung (<https://homl.info/108>) in dieses Thema an. Wählen Sie eine bestimmte Embedded Reber Grammar aus (wie zum Beispiel die von Jenny Orrs Seite) und trainieren Sie dann ein RNN darauf, zu erkennen, ob ein String diese Grammatik befolgt oder nicht. Sie werden zuerst eine Funktion schreiben müssen, die einen Trainingsbatch erzeugt, dessen Strings die Grammatik zu 50% befolgen.
9. Trainieren Sie ein Encoder-Decoder-Modell, das einen Datumsstring von einem Format in ein anderes umwandeln kann (zum Beispiel von »April 22, 2019« nach »2019-04-22«).
10. Gehen Sie das TensorFlow-Tutorium zu Neural Machine Translation with Attention (<https://homl.info/nmttuto>) durch.
11. Nutzen Sie eines der neueren Sprachmodelle (zum Beispiel BERT), um überzeugendere Shakespeare-Texte zu schaffen.

Lösungen zu diesen Übungen finden Sie in [Anhang A](#).

Representation Learning und Generative Learning mit Autoencodern und GANs

Autoencoder sind künstliche neuronale Netze, die eine effiziente Repräsentation der Eingabedaten, die *latenten Repräsentationen* oder *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](#)), insbesondere für Visualisierungszwecke. Autoencoder helfen auch bei der Merkmalserkennung 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. Allerdings sind die generierten Bilder im Allgemeinen unscharf und nicht sehr realistisch.

Im Gegensatz dazu sind von Generative Adversarial Networks (GANs) erzeugte Gesichter mittlerweile so realistisch, dass man nur schwer glauben kann, dass die dort abgebildeten Personen nicht real sind. Sie können sich selbst davon überzeugen, indem Sie die Website <https://thispersondoesnotexist.com> aufrufen, die von einer aktuellen GAN-Architektur namens *StyleGAN* erzeugte Gesichter vorstellt (sie können sich auch <https://thisrentaldoesnotexist.com> anschauen, um generierte AirBnB-Zimmer zu bestaunen). GANs werden jetzt häufig zur Super-Resolution eingesetzt (also zum Erhöhen der Auflösung eines Bilds), zur Kolorierung (<https://github.com/jantic/DeOldify>), zur leistungsfähigen Bildbearbeitung (zum Beispiel zum Ersetzen von Foto-Bombern durch realistische Hintergründe), zum Umwandeln einer einfachen Skizze in ein fotorealistisches Bild, zum Vorhersagen des nächsten Frames in einem Video, zum Erweitern eines Datensatzes (um andere Modelle zu trainieren), zum Generieren anderer Datentypen (wie Text, Audio und Zeitserien), zum Identifizieren der Schwächen in anderen Modellen und zum Stärken derselben und so weiter.

Autoencoder und GANs sind beide unüberwacht, sie lernen beide dichte Repräsentationen, sie können beide als generative Modelle genutzt werden, und sie haben viele gemeinsame Anwendungszwecke. Aber sie arbeiten sehr unterschiedlich:

- Autoencoder lernen einfach, ihre Eingaben in ihre Ausgaben zu kopieren. Das mag wie eine triviale Aufgabe klingen, aber wir werden sehen, dass dies durch das Beschränken des Netzes auf unterschiedlichste Art und Weise ziemlich schwierig werden kann. So können Sie beispielsweise die Größe der latenten Repräsentation begrenzen oder

Rauschen auf die Eingaben legen und das Netzwerk darauf trainieren, die ursprünglichen Eingaben wiederherzustellen. Diese Beschränkungen verhindern, dass der Autoencoder die Eingaben trivial in die Ausgaben kopiert, was ihn dazu zwingt, effiziente Wege zum Repräsentieren von Daten zu lernen. Kurz gesagt, sind die Codings Nebenprodukte des Autoencoder beim Lernen der Identitätsfunktion unter bestimmten Einschränkungen.

- GANs bestehen aus zwei neuronalen Netzen: einem *Generator*, der versucht, Daten zu erzeugen, die wie die Trainingsdaten aussehen, und einem *Diskriminator*, der versucht, reale Daten von generierten Daten zu unterscheiden. Diese Architektur ist im Deep Learning etwas Besonderes, da Generator und Diskriminatör während des Trainings gegeneinander antreten: Der Generator wird oft mit einem Kriminellen verglichen, der versucht, realistisch wirkendes Falschgeld zu erzeugen, während der Diskriminatör wie ein Kommissar bei der Polizei versucht, echtes Geld und Falschgeld auseinanderzuhalten. *Adversarial Training* wird oft als eine der wichtigsten Ideen der letzten Jahre angesehen. 2016 hat Yann LeCun sogar gesagt, dass es die »interessanteste Idee der letzten zehn Jahre im Bereich des Machine Learning war«.

In diesem Kapitel werden wir damit beginnen, genauer zu erklären, wie Autoencoder funktionieren und wie man sie zur Dimensionalitätsreduktion, zur Merkmalsextrahierung, zum unüberwachten Vortraining oder als generatives Modell einsetzt. Das wird uns ganz natürlich zu GANs führen. Wir werden ein einfaches GAN zum Generieren von Fake-Bidern bauen, aber wir werden sehen, dass das Training häufig ziemlich schwierig ist. Wir werden die größten Schwierigkeiten beim Adversarial Training besprechen und die wichtigsten Techniken kennenlernen, um sie zu umgehen. Beginnen wir mit den Autoencodern.

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, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

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 es sich einfach um die Liste der geraden Zahlen von 50 abwärts bis 14 handelt. Haben Sie dieses Muster erst einmal erkannt, ist die zweite Folge viel einfacher zu merken, da Sie sich nur noch das Muster (also die fallenden geraden Zahlen) sowie die Start- und Endnummer 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://homl.info/111>).¹ Sie hatten beobachtet, dass professionelle Schachspieler sich die Position sämtlicher Figuren innerhalb von fünf Sekunden merken konnten, was die meisten

Menschen 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 latente 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 latente Repräsentation in die Ausgabe umwandelt (siehe Abbildung 17-1).

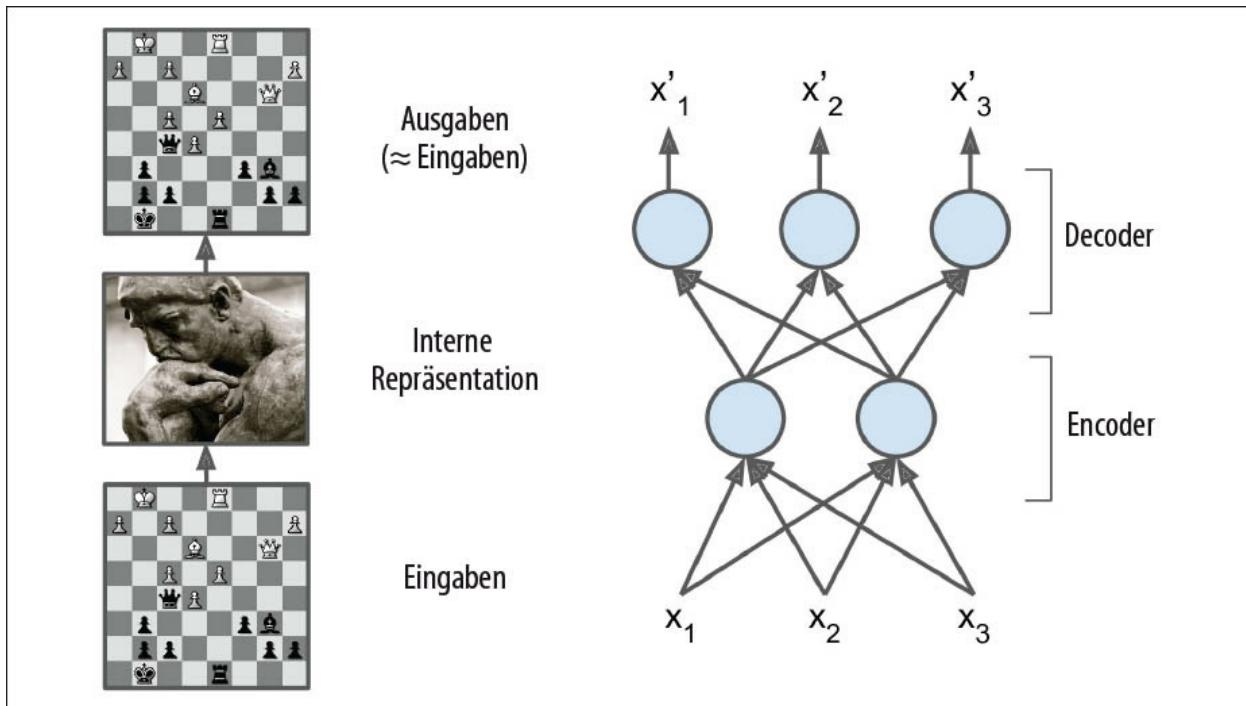


Abbildung 17-1: Das Gedächtnisexperiment mit Schachpartien (links) und ein einfacher Autoencoder (rechts)

Ein Autoencoder hat also normalerweise die gleiche Architektur wie ein mehrschichtiges Perzepron (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.

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 untvollständiger Autoencoder zur Dimensionsreduktion implementieren lässt.

Hauptkomponentenzerlegung mit einem untvollständigen linearen Autoencoder

Wenn der Autoencoder nur lineare Aktivierungen verwendet und die Kostenfunktion die mittlere quadratische Abweichung (MSE) ist, führt er letztendlich nur eine Hauptkomponentenzerlegung durch (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:

```
from tensorflow import keras

encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])))

decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])))

autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=0.1))
```

Dieser Code unterscheidet sich wirklich nicht besonders von den MLPs aus den vergangenen Kapiteln. Ein paar Dinge sind dabei zu aber betonen:

- Wir haben den Autoencoder in zwei Unterkomponenten aufgeteilt – den Encoder und den Decoder. Bei beiden handelt es sich um normale Sequential-Modelle mit einer einzelnen Dense-Schicht.
- Die Anzahl der Ausgaben und Eingaben des Autoencoder 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.

Trainieren wir das Modell nun mit einem einfachen generierten 3-D-Datensatz und nutzen es, um den gleichen Datensatz zu codieren (d.h. auf zwei Dimensionen zu projizieren):

```
history = autoencoder.fit(X_train, X_train, epochs=20)

codings = encoder.predict(X_train)
```

Beachten Sie, dass der gleiche Datensatz `X_train` sowohl als Eingabe wie auch als Ziel

verwendet wird. [Abbildung 17-2](#) zeigt den ursprünglichen 3-D-Datensatz (links) und die Ausgabe der verborgenen Schicht des Autoencoder (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).

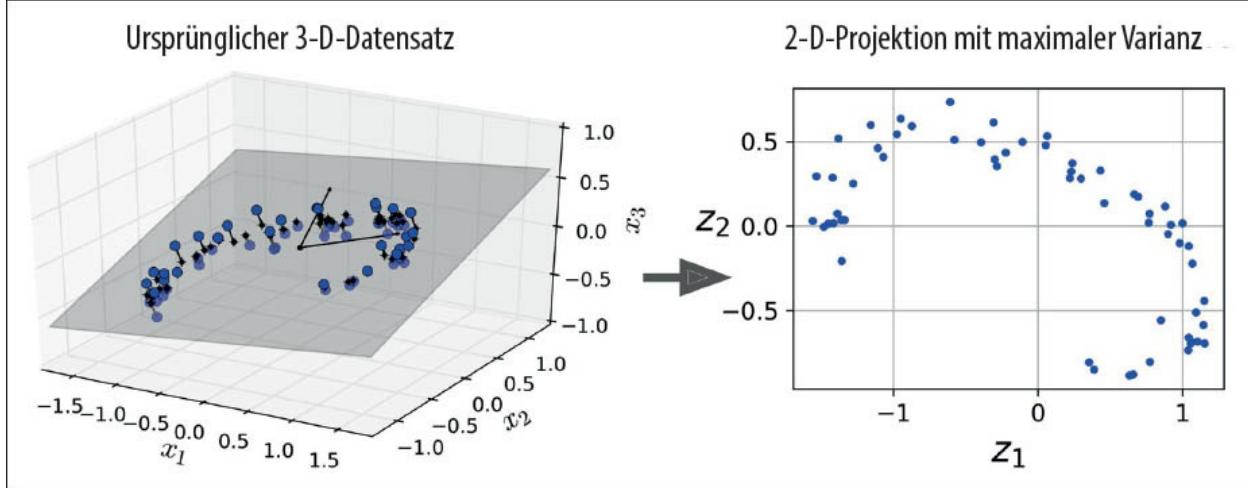


Abbildung 17-2: Von einem unvollständigen linearen Autoencoder durchgeführte PCA



Sie können sich Autoencoder als Form des selbstüberwachenden Lernens vorstellen (also eine selbstüberwachte Lerntechnik mit automatisch erzeugten Labels – in diesem Fall einfach gleich den Eingaben).

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 Autoencoder 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 100 Neuronen, einer zentralen verborgenen Schicht mit 30 Neuronen, einer weiteren Schicht mit 100 Neuronen und schließlich einer Ausgabeschicht mit 784 Neuronen. Ein solcher Stacked Autoencoder ist in [Abbildung 17-3](#) dargestellt.

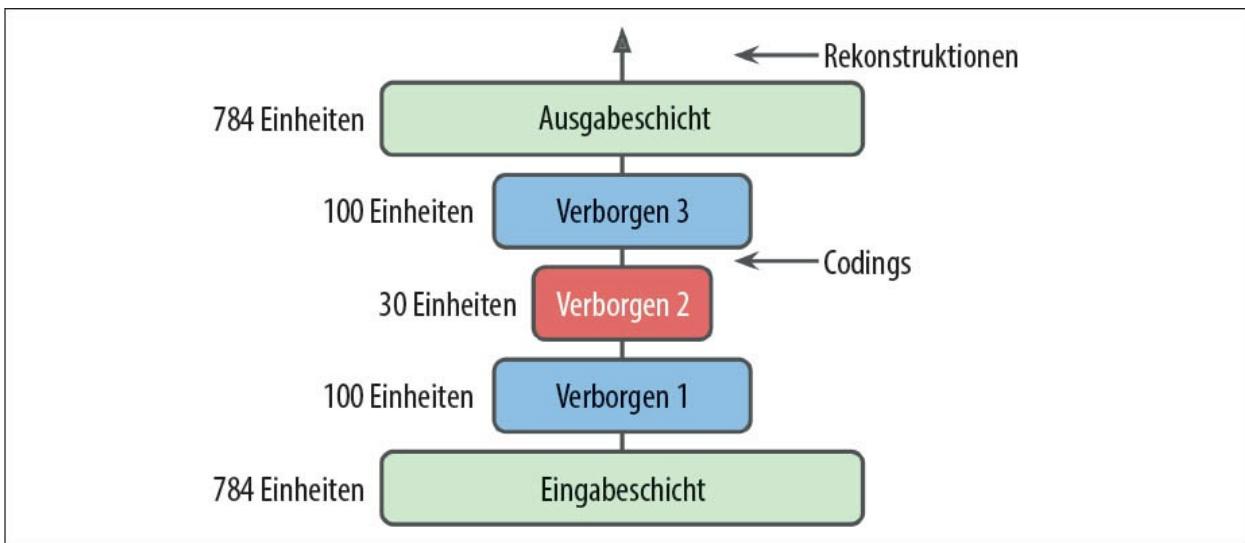


Abbildung 17-3: Stacked Autoencoder

Einen Stacked Autoencoder mit Keras implementieren

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 Fashion-MNIST-Daten (geladen und normalisiert wie in [Kapitel 10](#)) mit der SELU-Aktivierungsfunktion:

```

stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])

stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                    optimizer=keras.optimizers.SGD(lr=1.5))

```

```
history = stacked_ae.fit(X_train, X_train, epochs=10,
                         validation_data=[X_valid, X_valid])
```

Gehen wir diesen Code durch:

- Wie zuvor teilen wir das Autoencoder-Modell in zwei Untermodelle auf – den Encoder und den Decoder.
- Der Encoder übernimmt Graustufenbilder mit 28×28 Pixeln, klopft sie flach, sodass jedes Bild als Vektor der Größe 784 repräsentiert wird, und verarbeitet diese Vektoren dann in zwei Dense-Schichten mit schrumpfender Größe (100 Einheiten, dann 30 Einheiten), die jeweils die SELU-Aktivierungsfunktion nutzen (Sie können auch noch die LeCun-Initialisierung ergänzen, aber das Netz ist nicht sehr tief, daher wird es keinen großen Unterschied machen). Für jedes Eingabebild gibt der Encoder einen Vektor der Größe 30 aus.
- Der Decoder übernimmt Codings der Größe 30 (vom Encoder ausgegeben), verarbeitet sie in zwei Dense-Schichten von zunehmender Größe (100 Einheiten, dann 784 Einheiten) und wandelt die abschließenden Vektoren in 28×28 -Arrays um, sodass die Ausgaben des Decoders die gleiche Form wie die Eingaben des Encoders besitzen.
- Beim Kompilieren des Stacked Autoencoder nutzen wir als Verlust die binäre Kreuzentropie statt des mittleren quadratischen Fehlers. Wir behandeln die Rekonstruktionsaufgabe als binäres Multilabel-Klassifikationsproblem: Jede Pixelintensität repräsentiert die Wahrscheinlichkeit, dass das Pixel schwarz sein sollte. Wenn es so betrachtet (und nicht als Regressionsproblem), konvergiert das Modell auch schneller.²
- Schließlich trainieren wir das Modell mit `X_train` sowohl als Eingaben wie auch als Ziele (genauso nutzen wir `X_valid` für die Validierung sowohl als Eingaben wie auch für die Ziele).

Visualisieren der Rekonstruktionen

Um sicherzustellen, dass ein Autoencoder gut trainiert ist, können Sie die Ein- und Ausgaben miteinander vergleichen. Die Unterschiede sollten nicht signifikant sein. Zeichnen wir deshalb ein paar Bilder aus dem Validierungsdatensatz und deren Rekonstruktionen:

```
def plot_image(image):
    plt.imshow(image, cmap="binary")
    plt.axis("off")

def show_reconstructions(model, n_images=5):
    reconstructions = model.predict(X_valid[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
```

```

for image_index in range(n_images):
    plt.subplot(2, n_images, 1 + image_index)
    plot_image(X_valid[image_index])
    plt.subplot(2, n_images, 1 + n_images + image_index)
    plot_image(reconstructions[image_index])

show_reconstructions(stacked_ae)

```

[Abbildung 17-4](#) zeigt die sich ergebenden Bilder.



Abbildung 17-4: Ursprüngliche Bilder (oben) und ihre Rekonstruktionen (unten)

Die Rekonstruktionen sind erkennbar, aber es ist schon einiges verloren gegangen. Wir müssen das Modell länger trainieren, Encoder und Decoder tiefer machen oder die Codings verlängern. Aber wenn wir das Netz zu leistungsfähig machen, wird es perfekte Rekonstruktionen schaffen, ohne nützliche Muster in den Daten zu lernen. Daher wollen wir erst einmal bei diesem Modell bleiben.

Den Fashion-MNIST-Datensatz visualisieren

Nachdem wir nun einen Stacked Autoencoder trainiert haben, können wir ihn verwenden, um die Dimensionalität des Datensatzes zu reduzieren. Zur Visualisierung liefert das im Vergleich zu anderen Algorithmen zur Dimensionsreduktion (wie zum Beispiel die aus [Kapitel 8](#)) nicht allzu gute Ergebnisse, aber ein großer Vorteil von Autoencodern ist, dass sie mit großen Datensätzen mit vielen Instanzen und Merkmalen umgehen können. Eine Strategie ist daher, mit einem Autoencoder die Dimensionalität auf ein vernünftiges Maß zu reduzieren und dann einen anderen Algorithmus zur Dimensionsreduktion für die Visualisierung zu verwenden. Nutzen wir diese Strategie, um Fashion MNIST zu visualisieren. Als Erstes verwenden wir den Encoder aus unserem Stacked Autoencoder, um die Dimensionalität auf 30 zu verringern, dann setzen wir die Scikit-Learn-Implementierung des t-SNE-Algorithmus ein, um die Dimensionalität zur Visualisierung auf 2 zu reduzieren:

```
from sklearn.manifold import TSNE
```

```

X_valid_compressed = stacked_encoder.predict(X_valid)

tsne = TSNE()

X_valid_2D = tsne.fit_transform(X_valid_compressed)

```

Jetzt können wir den Datensatz plotten:

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
```

In Abbildung 17-5 sehen Sie den Scatterplot (ein bisschen aufgehübscht, indem wir einige der Bilder mit ausgeben). Der t-SNE-Algorithmus hat eine Reihe von Clustern erkannt, die recht gut zu den Kategorien passen (jede ist durch eine andere Farbe dargestellt).

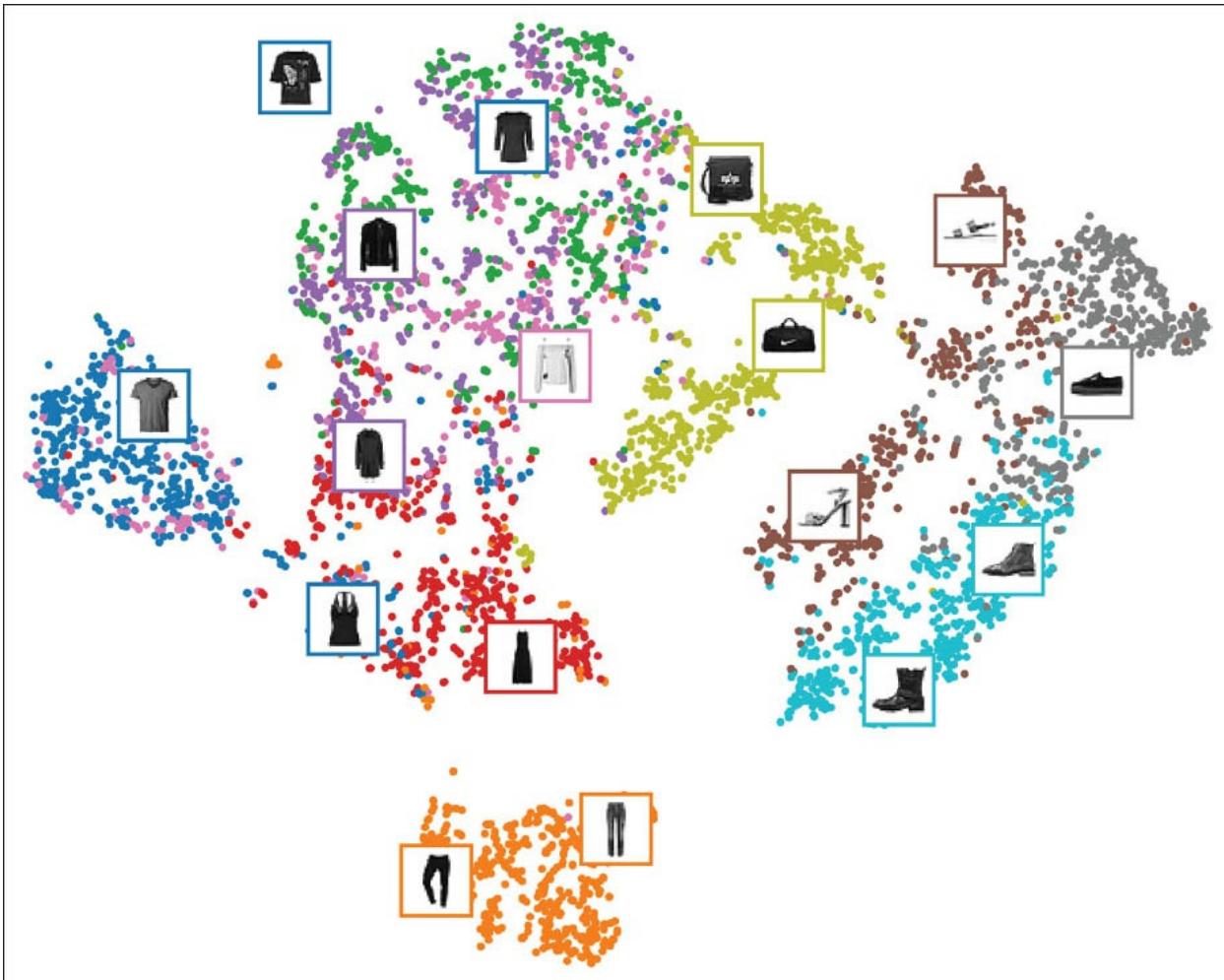


Abbildung 17-5: Visualisierung von Fashion MNIST mit einem Autoencoder und t-SNE

Autoencoder können also zur Dimensionsreduktion eingesetzt werden. Eine andere Anwendung ist das unüberwachte Vortrainieren.

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 17-6](#) zeigt, wie Sie einen Stacked Autoencoder zum unüberwachten Vortrainieren eines neuronalen Netzes zur Klassifikation einsetzen können. Beim Trainieren des Klassifikators sollten Sie die vorgenannten Schichten einfrieren (zumindest die ersten).

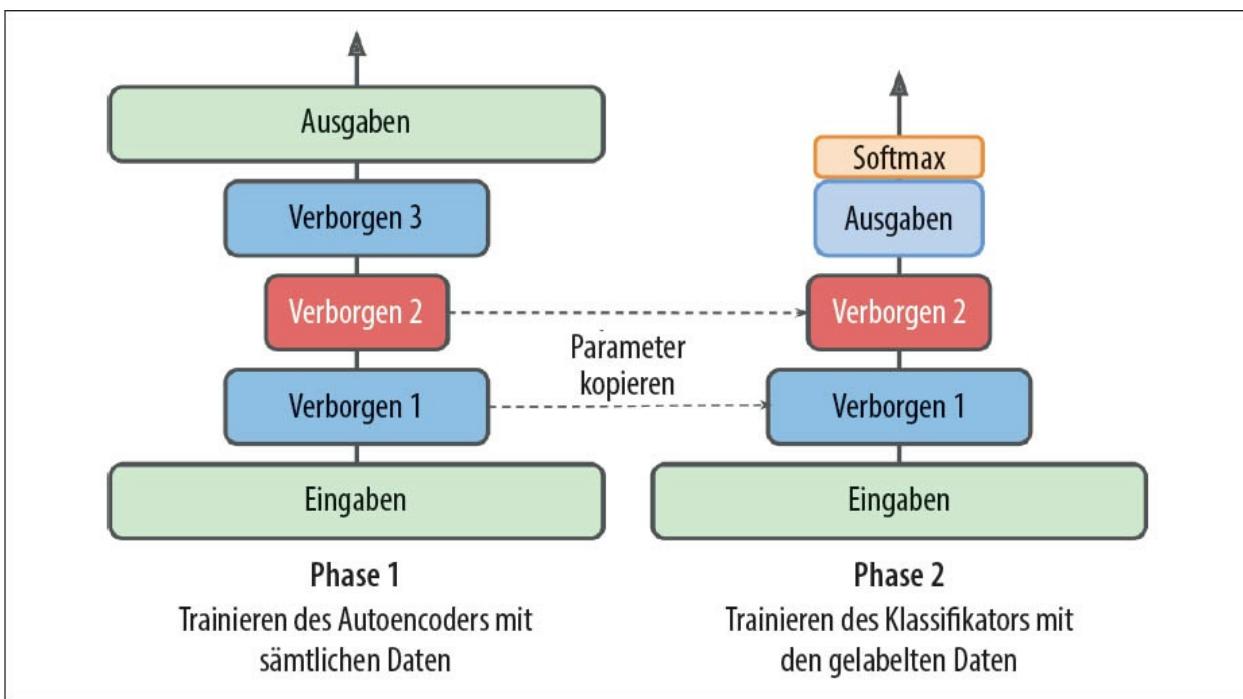


Abbildung 17-6: Unüberwachtes Vortrainieren mit Autoencodern

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.

An der Implementierung ist nichts besonders: Trainieren Sie einfach einen Autoencoder mit allen Trainingsdaten (gelabelte und ungelabelte), dann verwenden Sie seine Encoder-Schichten, um ein neues neuronales Netz zu erstellen (ein Beispiel finden Sie in den Übungen am Ende dieses Kapitels).

Schauen wir uns jetzt ein paar Techniken zum Trainieren von Stacked Autoencodern an.

Kopplung von Gewichten

Ist ein Autoencoder wie der soeben erstellte symmetrisch, lassen sich die Gewichte der Decoder- und Encoder-Schichten 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 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.

Um mit Keras Gewichte zwischen den Schichten zu koppeln, definieren wir eine eigene Schicht:

```
class DenseTranspose(keras.layers.Layer):

    def __init__(self, dense, activation=None, **kwargs):
        self.dense = dense
        self.activation = keras.activations.get(activation)
        super().__init__(**kwargs)

    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias", initializer="zeros",
                                      shape=[self.dense.input_shape[-1]])
        super().build(batch_input_shape)

    def call(self, inputs):
        z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(z + self.biases)
```

Diese eigene Schicht verhält sich wie eine normale Dense-Schicht, aber sie nutzt die Gewichte einer anderen Dense-Schicht, die transponiert sind (das Setzen von `transpose_b=True` entspricht dem Transponieren des zweiten Arguments, ist aber effizienter, da es innerhalb der `matmul()`-Operation geschieht). Aber sie nutzt ihren eigenen Bias-Vektor. Als Nächstes können wir einen neuen Stacked Autoencoder bauen, der so wie der vorherige ist, aber die Dense-Schichten des Decoders nutzt, die mit den Dense-Schichten des Encoders gekoppelt sind:

```
dense_1 = keras.layers.Dense(100, activation="selu")
dense_2 = keras.layers.Dense(30, activation="selu")

tied_encoder = keras.models.Sequential([
```

```

    keras.layers.Flatten(input_shape=[28, 28]),
    dense_1,
    dense_2
])

tied_decoder = keras.models.Sequential([
    DenseTranspose(dense_2, activation="selu"),
    DenseTranspose(dense_1, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

tied_ae = keras.models.Sequential([tied_encoder, tied_decoder])

```

Dieses Modell erreicht einen etwas kleineren Rekonstruktionsfehler als das vorherige, besitzt aber nur in etwa die Hälfte der Parameter.

Trainieren mehrerer Autoencoder nacheinander

Anstatt den gesamten Stacked Autoencoder wie eben gezeigt in einem Durchgang zu trainieren, ist es möglich, einen einschichtigen Autoencoder nach dem anderen zu trainieren. Anschließend lassen sich diese zu einem Stacked Autoencoder aufstapeln (daher der Name), wie [Abbildung 17-7](#) zeigt. Diese Technik kommt heutzutage nur noch selten zum Einsatz, aber eventuell stolpern Sie immer noch in Artikeln über »Greedy Layer-Wise Training«, daher ist es gut zu wissen, was es bedeutet.

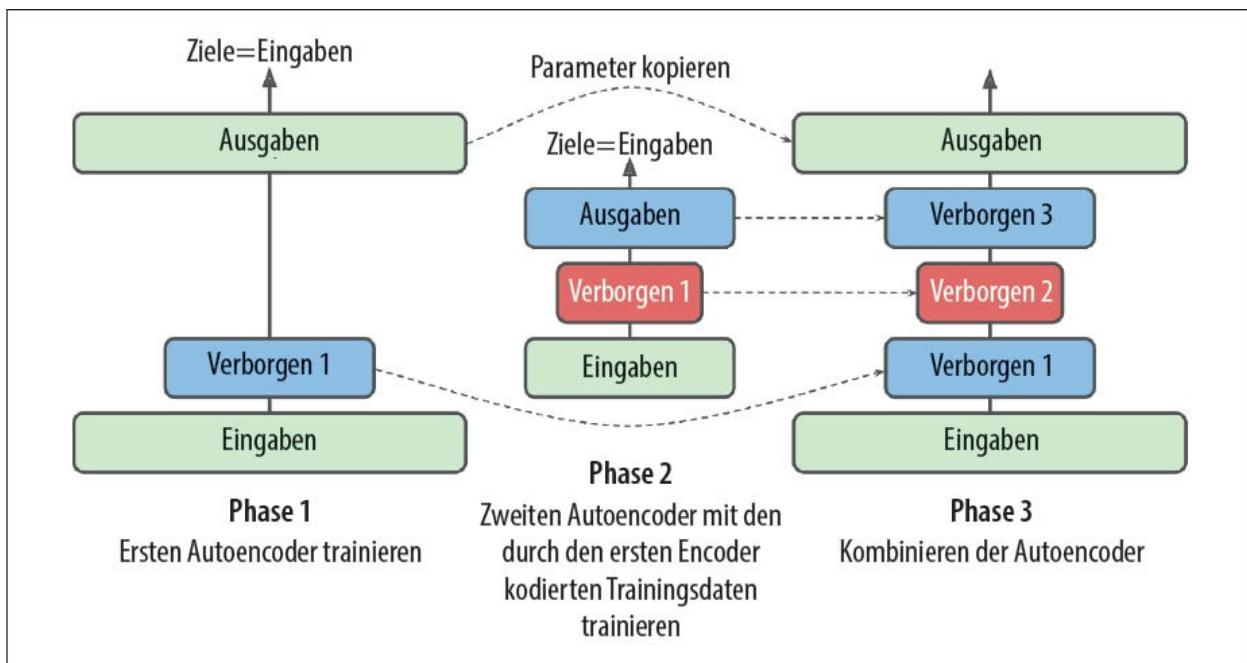


Abbildung 17-7: Trainieren mehrerer Autoencoder nacheinander

In der ersten Trainingsphase lernt der erste Autoencoder, die Eingaben zu rekonstruieren. Dann codieren wir den gesamten Trainingsdatensatz mit diesem ersten Autoencoder und erhalten damit einen neuen (komprimierten) Trainingsdatensatz. Dann trainieren wir einen zweiten Autoencoder mit diesem neuen Datensatz – das ist die zweite Phase des Trainings. Am Ende erstellen Sie einfach das in Abbildung 17-7 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.

Wie schon erwähnt, war einer der Auslöser für das aktuelle Interesse an Deep Learning die Entdeckung von Geoffrey Hinton et al. (<https://homl.info/136>) aus dem Jahr 2006, dass Deep Neural Networks unüberwacht vorgenommen werden können, wenn man diesen Greedy-Layer-Wise-Ansatz verfolgt. Die Autoren haben dazu Restricted Boltzmann Machines genutzt (RBMs, siehe Anhang E), aber im Jahr 2007 zeigten Yoshua Bengio et al. (<https://homl.info/112>)³, dass Autoencoder genauso gut funktionieren. Für viele Jahre war dies der einzige effiziente Weg, Deep Networks zu trainieren, bis es viele der in Kapitel 11 vorgestellten Techniken möglich machten, ein Deep Net auf einmal zu trainieren.

Autoencoder sind nicht auf dichte Netze beschränkt: Sie können auch Convolutional Autoencoder oder sogar rekurrente Autoencoder bauen. Schauen wir uns diese an.

Convolutional Autoencoder

Arbeiten Sie mit Bildern, werden die bisher genutzten Autoencoder nicht sehr gut funktionieren (sofern die Bilder nicht sehr klein sind): Wie wir in Kapitel 14 gesehen haben, sind

Convolutional Neural Networks dafür viel besser geeignet als dichte Netze. Wollen Sie also einen Autoencoder für Bilder bauen (zum Beispiel zum unüberwachten Vortraining oder zur Dimensionsreduktion), müssen Sie einen *Convolutional Autoencoder* (<https://homl.info/convae>) bauen.⁴ Der Encoder ist ein normales CNN aus Convolutional Layers und Pooling Layers. Es reduziert typischerweise die räumliche Dimension der Eingaben (also Höhe und Breite), während es die Tiefe (also die Anzahl an Feature Maps) erweitert. Der Decoder muss dann das Gegenteil tun (die Bilder wieder größer skalieren und deren Tiefe auf die ursprünglichen Dimensionen reduzieren), wofür Sie transponierte Convolutional Layers verwenden können (alternativ könnten Sie auch Upsampling Layers mit Convolutional Layers kombinieren). Hier ein einfacher Convolutional Autoencoder für Fashion MNIST:

```
conv_encoder = keras.models.Sequential([
    keras.layers.Reshape([28, 28, 1], input_shape=[28, 28]),
    keras.layers.Conv2D(16, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2)
])

conv_decoder = keras.models.Sequential([
    keras.layers.Conv2DTranspose(32, kernel_size=3, strides=2, padding="valid",
                               activation="selu",
                               input_shape=[3, 3, 64]),
    keras.layers.Conv2DTranspose(16, kernel_size=3, strides=2, padding="same",
                               activation="selu"),
    keras.layers.Conv2DTranspose(1, kernel_size=3, strides=2, padding="same",
                               activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

conv_ae = keras.models.Sequential([conv_encoder, conv_decoder])
```

Rekurrente Autoencoder

Wollen Sie einen Autoencoder für Sequenzen bauen, etwa für Zeitserien oder Text (zum Beispiel zum unüberwachten Lernen oder zur Dimensionsreduktion), können rekurrente neuronale Netze (siehe [Kapitel 15](#)) besser funktionieren als dichte Netze. Das Bauen eines *rekurrenten Autoencoder* ist recht simpel: Beim Encoder handelt es sich üblicherweise um ein Sequence-to-Vector-RNN, das die Eingabesequenz auf einen einzelnen Vektor komprimiert. Der Decoder ist ein Vector-to-Sequence-RNN, das den gegenteiligen Weg geht:

```
recurrent_encoder = keras.models.Sequential([
    keras.layers.LSTM(100, return_sequences=True, input_shape=[None, 28]),
    keras.layers.LSTM(30)
])

recurrent_decoder = keras.models.Sequential([
    keras.layers.RepeatVector(28, input_shape=[30]),
    keras.layers.LSTM(100, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(28, activation="sigmoid"))
])

recurrent_ae = keras.models.Sequential([recurrent_encoder, recurrent_decoder])
```

Dieser rekurrente Autoencoder kann Sequenzen beliebiger Länge verarbeiten, wobei pro Zeitschritt 28 Dimensionen möglich sind. Praktischerweise heißt das, dass Sie MNIST-Bilder verarbeiten können, indem Sie jedes Bild als Sequenz von Zeilen betrachten: Bei jedem Zeitschritt wird das RNN eine einzelne Zeile mit 28 Pixeln verarbeiten. Offensichtlich könnten Sie einen rekurrenten Autoencoder für jede Art von Sequenz einsetzen. Beachten Sie, dass wir als erste Schicht des Decoders eine `RepeatVector`-Schicht nutzen, um sicherzustellen, dass seine Eingangsvektoren dem Decoder bei jedem Zeitschritt mitgegeben werden.

Okay, gehen wir einen Schritt zurück. Bisher haben wir verschiedene Arten von Autoencodern kennengelernt (einfach, Stacked, Convolutional und rekurrent) und uns angeschaut, wie man sie trainiert (entweder in einem Zug oder Schicht für Schicht). Wir haben auch ein paar Anwendungen betrachtet: Datenvisualisierung und unüberwachtes Vortraining.

Bisher haben wir die Größe der Coding-Schichten begrenzt, um den Autoencoder zu zwingen, die interessanten Merkmale zu lernen, womit er unvollständig ist. Es gibt aber viele andere Arten von Beschränkungen, die zum Einsatz kommen können, unter anderem solche, die es der Coding-Schicht erlauben, so groß wie die Eingaben zu sein – oder sogar größer –, was zu einem *übergelösten Autoencoder* führt. Schauen wir uns einige dieser Ansätze an.

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. Diese Idee geistert seit den 1980er-Jahren herum (zum Beispiel ist sie in Yann LeCuns Master-Thesis von 1987 enthalten). In einem Artikel (<https://homl.info/113>)⁵ aus dem Jahr 2008 haben Pascal Vincent et al. gezeigt, dass Autoencoder auch zur Merkmalsextraktion genutzt werden können. Und in einem Artikel (<https://homl.info/114>)⁶ aus dem Jahr 2010 haben Vincent et al. *Stacked Denoising Autoencoder* vorgestellt.

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 17-8](#) zeigt beide Möglichkeiten.

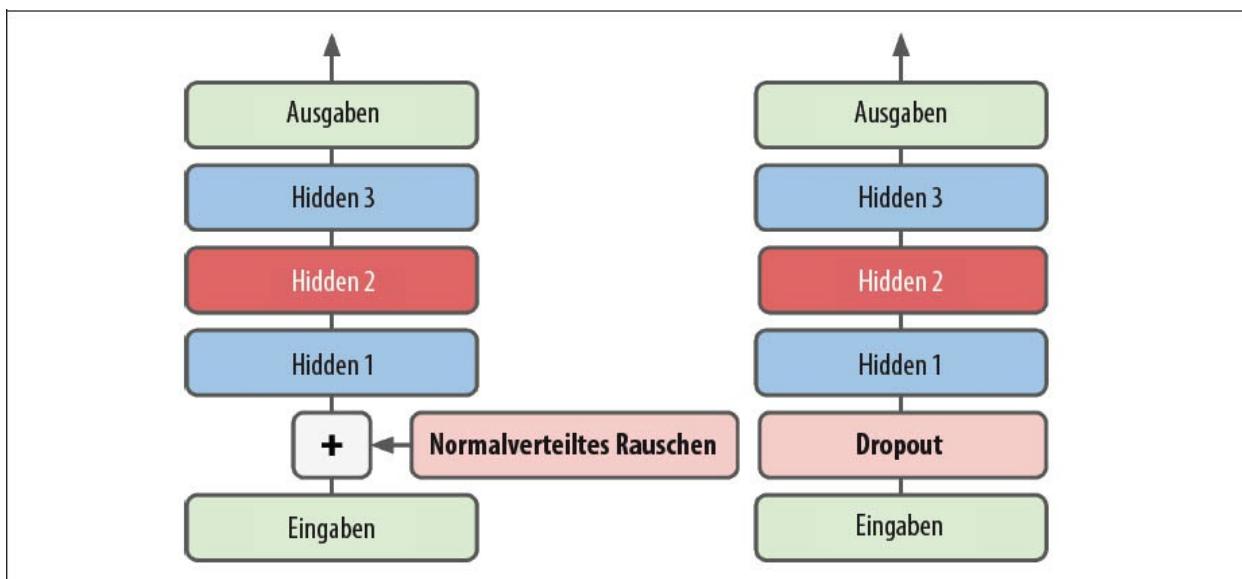


Abbildung 17-8: Denoising Autoencoder mit normalverteiltem Rauschen (links) oder Drop-out (rechts)

Die Implementierung ist recht klar: Es handelt sich um einen normalen Stacked Autoencoder mit einer zusätzlichen Dropout-Schicht, die auf die Eingaben des Encoders angewendet wird (oder Sie könnten stattdessen eine GaussianNoise-Schicht nutzen). Denken Sie daran, dass die Dropout-Schicht nur während des Trainings aktiv ist (wie auch die GaussianNoise-Schicht):

```
dropout_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu")
])

dropout_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid")
])
```

```

        keras.layers.Dense(100, activation="selu", input_shape=[30]),
        keras.layers.Dense(28 * 28, activation="sigmoid"),
        keras.layers.Reshape([28, 28])
    )
dropout_ae = keras.models.Sequential([dropout_encoder, dropout_decoder])

```

[Abbildung 17-9](#) zeigt ein paar verrauschte Bilder (bei denen die Hälfte der Pixel abgeschaltet wurde) sowie die vom Drop-out-basierten Denoising Autoencoder rekonstruierten Bilder. Beachten Sie, wie der Autoencoder Details vermutet, die gar nicht in den Eingaben vorhanden sind, wie zum Beispiel oben am weißen Shirt (untere Zeile, vierter Bild). Wie Sie sehen, können Denoising Autoencoder nicht nur zur Datenvisualisierung oder zum unüberwachten Vortraining eingesetzt werden, wie die anderen bisher besprochenen Autoencoder, sondern auch ziemlich einfach und effizient zum Entrauschen von Bildern.

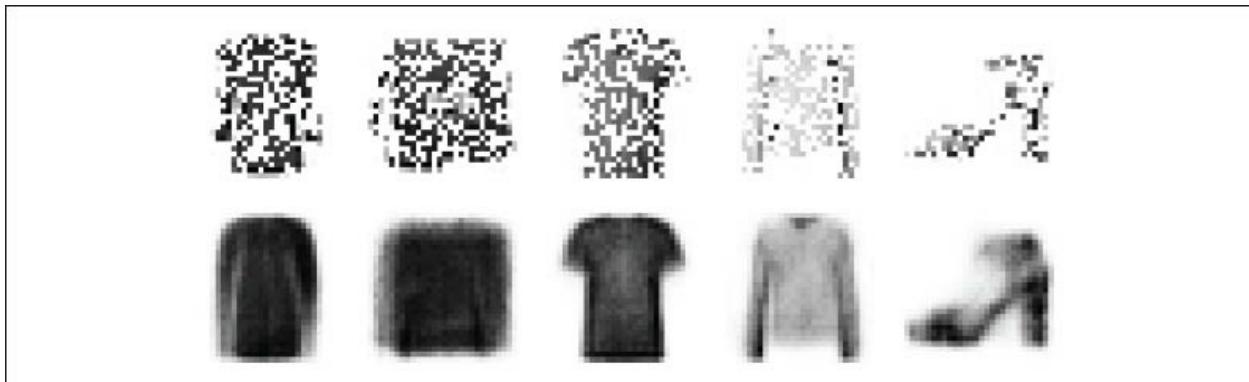


Abbildung 17-9: Verrauschte Bilder (oben) und ihre Rekonstruktionen (unten)

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. So 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).

Ein einfacher Ansatz ist die Verwendung der Sigmoid-Aktivierungsfunktion in der Coding-Schicht (um die Codings auf Werte zwischen 0 und 1 zu begrenzen), der Einsatz einer großen Coding-Schicht (zum Beispiel mit 300 Einheiten) und das Hinzufügen von ein wenig ℓ_1 -Regularisierung zu den Aktivierungen der Coding-Schicht (der Decoder ist einfach ein normaler Decoder):

```

sparse_l1_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid"),
    keras.layers.ActivityRegularization(l1=1e-3)
])

sparse_l1_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

sparse_l1_ae = keras.models.Sequential([sparse_l1_encoder, sparse_l1_decoder])

```

Diese ActivityRegularization-Schicht gibt einfach ihre Eingaben zurück, aber als Nebeneffekt fügt sie einen Trainingsverlust hinzu, der der Summe der absoluten Werte ihrer Eingaben entspricht (diese Schicht hat nur während des Trainings einen Effekt). Sie könnten stattdessen auch die ActivityRegularization-Schicht entfernen und in der vorherigen Schicht `activity_regularizer=keras.regularizers.l1(1e-3)` setzen. Diese Strafe unterstützt das neuronale Netz darin, Codings zu erzeugen, die nahe an 0 sind, aber da es auch bestraft wird, wenn es die Eingaben nicht korrekt rekonstruiert, wird es zumindest ein paar Werte liefern, die nicht 0 sind. Durch das Verwenden der ℓ_1 -Norm statt der ℓ_2 -Norm wird das Netz dahin gedrängt, die wichtigsten Codings zu behalten und diejenigen auszumerzen, die für die Eingabebilder nicht erforderlich sind (statt einfach alle Codings zu reduzieren).

Ein anderes Vorgehen, das häufig zu besseren Ergebnissen führt, ist das Messen der tatsächlichen Spärlichkeit der Coding-Schicht bei jeder Trainingsiteration. Dann wird das Modell bestraft, wenn die gemessene Spärlichkeit von einer Zielspärlichkeit abweicht. Dazu berechnen wir die durchschnittliche Aktivierung jedes Neurons in der Coding-Schicht über den gesamten Trainingsbatch. Die Batchgröße darf nicht zu klein sein, denn ansonsten wird der Mittelwert nicht genau sein.

Haben wir die mittlere Aktivierung pro Neuron, wollen wir die Neuronen bestrafen, die zu aktiv oder nicht aktiv genug sind, indem wir zur Kostenfunktion einen *Spärlichkeitsverlust* addieren. Messen wir beispielsweise, dass ein Neuron eine durchschnittliche Aktivierung von 0,3 hat, ist die Zielspärlichkeit aber 0,1, muss es bestraft werden, um weniger aktiv zu sein. Ein Ansatz könnte dabei sein, einfach den quadratischen Fehler zur Kostenfunktion zu addieren $(0,3 - 0,1)^2$, aber in der Praxis funktioniert es besser, die Kullback-Leibler-(KL-)Divergenz zu verwenden (die kurz in [Kapitel 4](#) besprochen wurde). Sie hat steilere Gradienten als der Mean Squared

Error, wie Sie in Abbildung 17-10 sehen.

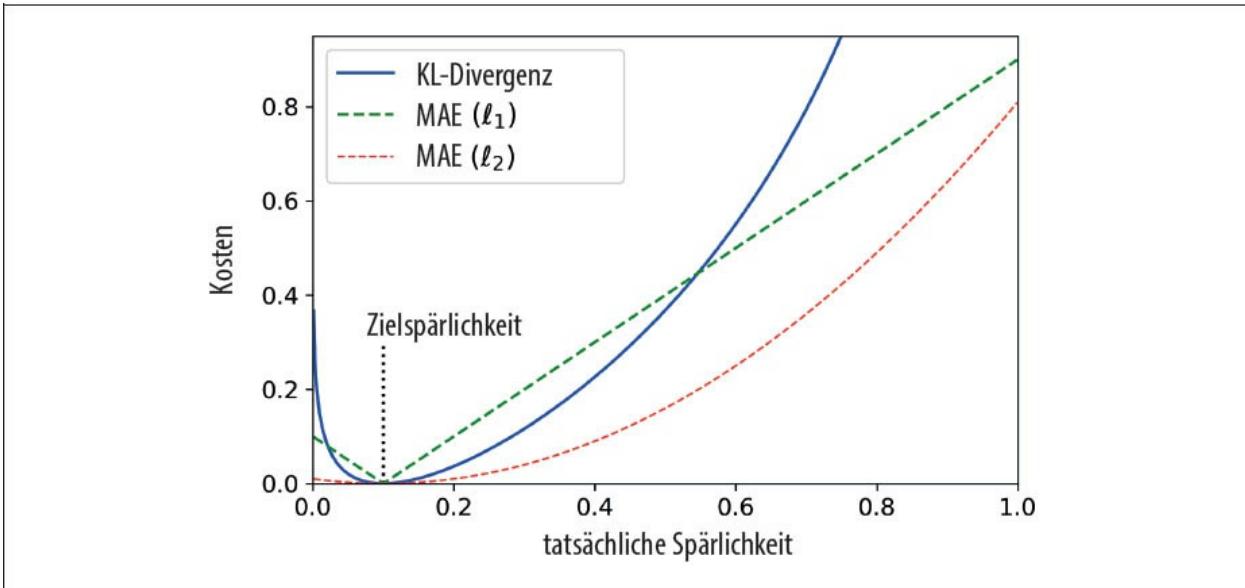


Abbildung 17-10: Spärlichkeitsverlust

Mit zwei gegebenen Wahrscheinlichkeitsverteilungen P und Q kann die KL-Divergenz zwischen diesen beiden Verteilungen, bezeichnet als $D_{\text{KL}}(P \parallel Q)$, wie in Formel 17-1 berechnet werden.

Formel 17-1: Kullback-Leibler-Divergenz

$$D_{\text{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

In unserem Fall wollen wir die Divergenz zwischen der Zielwahrscheinlichkeit p messen, dass ein Neuron in der Coding-Schicht aktiv sein wird, und der tatsächlichen Wahrscheinlichkeit q (also der mittleren Aktivierung über den Trainingsbatch). Daher vereinfacht sich die KL-Divergenz zu Formel 17-2.

Formel 17-2: KL-Divergenz zwischen der Zielspärlichkeit p und der tatsächlichen Spärlichkeit q

$$D_{\text{KL}}(p \parallel q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

Haben wir den Spärlichkeitsverlust für jedes Neuron in der Coding-Schicht berechnet, addieren wir diese Verluste und fügen noch das Ergebnis der Kostenfunktion hinzu. Um die relative Wichtigkeit des Spärlichkeitsverlusts und des Rekonstruktionsverlusts steuern zu können, multiplizieren wir den Spärlichkeitsverlust mit einem Spärlichkeitsgewicht als Hyperparameter. Ist das Gewicht zu hoch, wird das Modell nahe an der Zielspärlichkeit bleiben, aber die Eingaben eventuell nicht sauber rekonstruieren, womit es nutzlos wird. Ist das Gewicht hingegen zu niedrig, wird das Modell das Spärlichkeitsziel weitgehend ignorieren und keine interessanten Merkmale lernen.

Wir haben jetzt alles zusammen, was wir brauchen, um einen Sparse Autoencoder zu bauen, der

auf der KL-Divergenz basiert. Erstellen wir zuerst einen eigenen Regularisierer, um die KL-Divergenz-Regularisierung anzuwenden:

```
K = keras.backend

kl_divergence = keras.losses.kullback_leibler_divergence

class KLDivergenceRegularizer(keras.regularizers.Regularizer):

    def __init__(self, weight, target=0.1):
        self.weight = weight
        self.target = target

    def __call__(self, inputs):
        mean_activities = K.mean(inputs, axis=0)
        return self.weight * (
            kl_divergence(self.target, mean_activities) +
            kl_divergence(1. - self.target, 1. - mean_activities))
```

Jetzt können wir den Sparse Autoencoder bauen und dabei den KLDivergenceRegularizer für die Aktivierungen der Coding-Schicht einsetzen:

```
kld_reg = KLDivergenceRegularizer(weight=0.05, target=0.1)

sparse_kl_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid", activity_regularizer=kld_reg)
])

sparse_kl_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

sparse_kl_ae = keras.models.Sequential([sparse_kl_encoder, sparse_kl_decoder])
```

Nach dem Trainieren dieses Sparse Autoencoder mit dem Fashion MNIST sind die Aktivierungen der Neuronen in der Coding-Schicht größtenteils nahe an 0 (etwa 70% aller Aktivierungen sind kleiner als 0,1), und alle Neuronen besitzen eine durchschnittliche Aktivierung nahe an 0,1 (etwa 90% aller Neuronen haben eine mittlere Aktivierung zwischen 0,1 und 0,2), wie in [Abbildung 17-11](#) zu sehen ist.

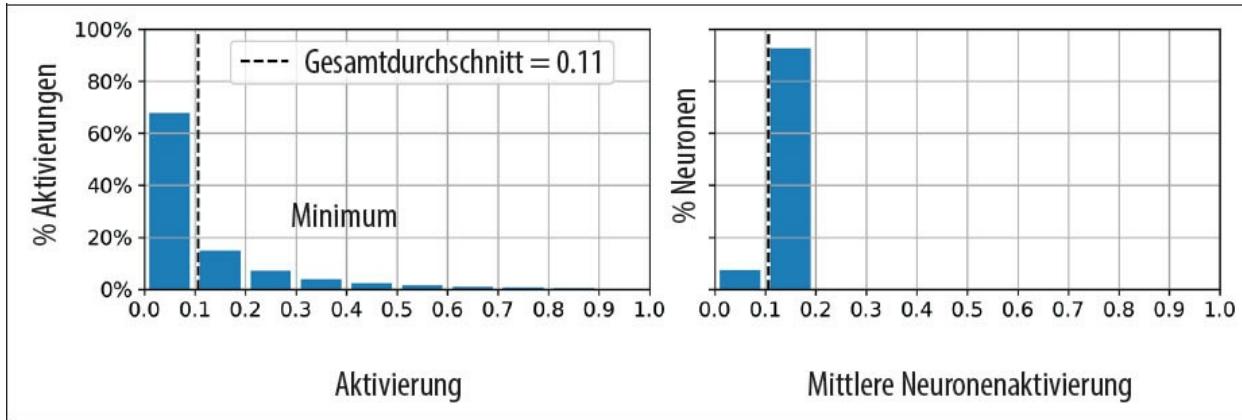


Abbildung 17-11: Verteilung aller Aktivierungen in der Coding-Schicht (links) und Verteilung der mittleren Aktivierung pro Neuron (rechts)

Variational Autoencoder

Eine weitere wichtige Art Autoencoder wurde im Jahr 2013 von Diederik Kingma und Max Welling erstmalig erwähnt (<https://hml.info/115>)⁷ 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, 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). Tatsächlich führen Variational Autoencoder, wie ihr Name schon andeutet, Variational Bayesian Inference durch (vorgestellt in [Kapitel 9](#)), bei der es sich um eine effiziente Methode zum Umsetzen einer angenäherten bayesschen Inferenz handelt.

Sehen wir uns ihre Funktionsweise an. [Abbildung 17-12](#) (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.

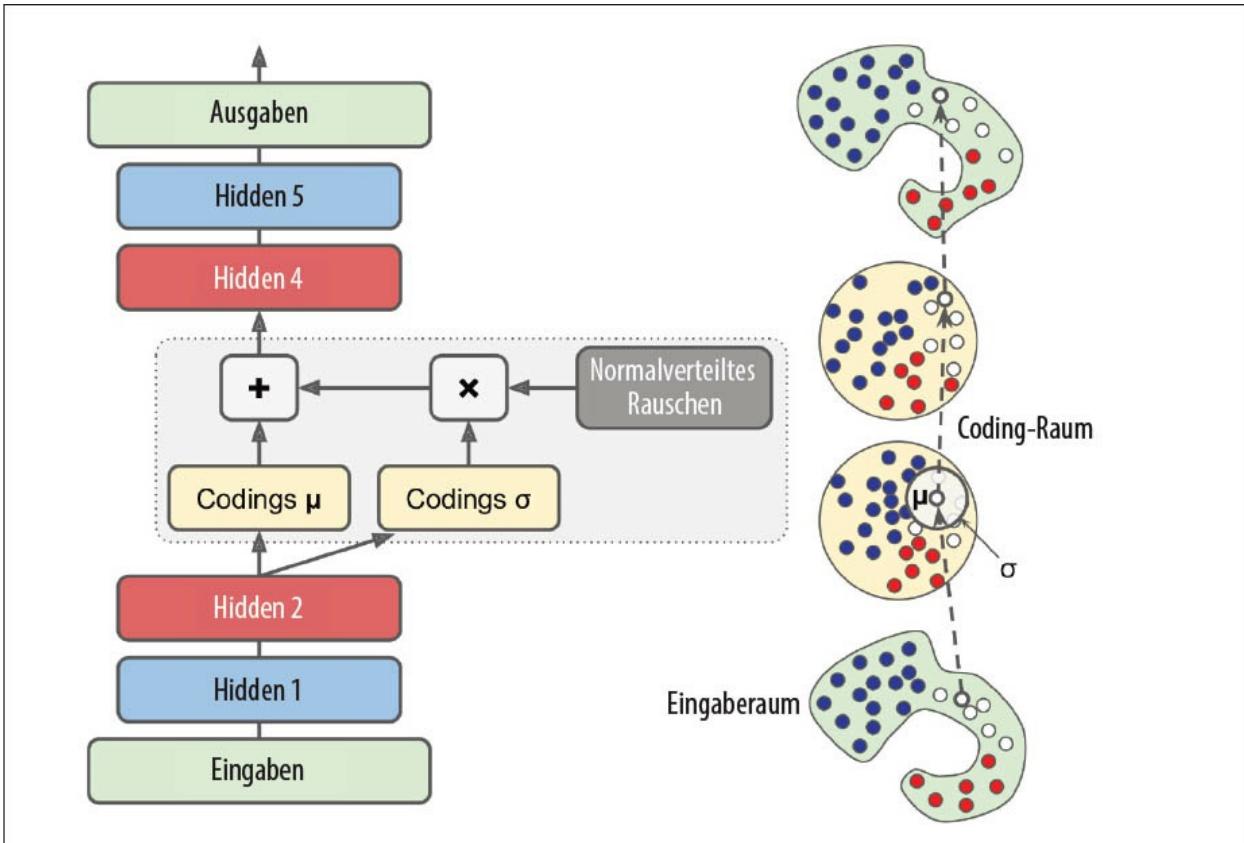


Abbildung 17-12: Variational Autoencoder (links) und ein von diesem verarbeiteter Datenpunkt (rechts)

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:⁸ Während des Trainierens drückt die Kostenfunktion (folgt in Kürze) die Codings allmählich in Richtung des Coderraums (auch als *latenter Raum* bezeichnet), der wie eine Wolke normalverteilter Punkte aussieht. Ein großer Vorteil hiervon ist, dass Sie nach dem Trainieren eines Variational Autoencoder 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 beim Sparse Autoencoder, insbesondere wegen des normalverteilten Rauschens, das die zur codierenden Schicht transportierte Informationsmenge begrenzt (und den Autoencoder zum Erlernen nützlicher Merkmale bewegt). Glücklicherweise vereinfachen sich die Gleichungen, sodass der latente Verlust recht einfach über [Formel 17-3](#) berechnet werden kann:⁹

Formel 17-3: Latenter Verlust des Variational Autoencoder

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^K 1 + \log(\sigma_i^2) - \sigma_i^2 - \mu_i^2$$

In dieser Gleichung ist \mathcal{L} der der latente Verlust, n die Dimension des Codings, und μ_i und σ_i sind Mittelwert und Standardabweichung der i . Komponente der Codings. Die Vektoren $\mathbf{\mu}$ und $\mathbf{\sigma}$ (die alle μ_i und σ_i enthalten) werden vom Encoder ausgegeben, wie in [Abbildung 17-12](#) (links) zu sehen ist.

Häufig optimiert man die Architektur eines Variational Autoencoder, indem man den Encoder $\gamma = \log(\sigma^2)$ statt σ ausgeben lässt. Der latente Verlust kann dann wie in [Formel 17-4](#) berechnet werden. Dieser Ansatz ist numerisch stabiler und beschleunigt das Training.

Formel 17-4: Latenter Verlust des Variational Autoencoder, umgeschrieben mit $\gamma = \log(\sigma^2)$

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^K 1 + \gamma_i - \exp(\gamma_i) - \mu_i^2$$

Bauen wir nun einen Variational Autoencoder für Fashion MNIST (wie in [Abbildung 17-12](#) dargestellt, aber mit der γ -Abwandlung). Zuerst brauchen wir eine eigene Schicht, um die Codings zu sampeln, wobei $\mathbf{\mu}$ und γ gegeben sind:

```
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return K.random_normal(tf.shape(log_var)) * K.exp(log_var / 2) + mean
```

Die Sampling-Schicht erwartet zwei Eingaben: `mean` ($\mathbf{\mu}$) und `log_var` (γ). Sie nutzt die Funktion `K.random_normal()`, um einen Zufallsvektor aus der Normalverteilung zu sampeln (mit der gleichen Form wie γ), wobei ein Mittelwert von 0 und eine Standardabweichung von 1 genutzt wird. Dann multipliziert sie ihn mit $\exp(\gamma / 2)$ (was gleich σ ist, wie Sie leicht überprüfen können), addiert schließlich $\mathbf{\mu}$ und gibt das Ergebnis zurück. Das führt zu einem normalverteilten Coding-Vektor mit Mittelwert $\mathbf{\mu}$ und Standardabweichung σ .

Als Nächstes können wir den Encoder erstellen, wobei wir die Functional API verwenden, weil das Modell nicht vollständig sequenziell ist:

```
codings_size = 10
```

```

inputs = keras.layers.Input(shape=[28, 28])

z = keras.layers.Flatten()(inputs)

z = keras.layers.Dense(150, activation="selu")(z)

z = keras.layers.Dense(100, activation="selu")(z)

codings_mean = keras.layers.Dense(codings_size)(z) #  $\mu$ 

codings_log_var = keras.layers.Dense(codings_size)(z) #  $\sigma$ 

codings = Sampling()([codings_mean, codings_log_var])

variational_encoder = keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])

```

Beachten Sie, dass die Dense-Schichten, die `codings_mean` (μ) und `codings_log_var` (σ) ausgeben, die gleichen Eingaben besitzen (also die Ausgaben der zweiten Dense-Schicht). Dann übergeben wir sowohl `codings_mean` wie auch `codings_log_var` an die Sampling-Schicht. Schließlich besitzt das `variational_encoder`-Modell drei Ausgaben, falls Sie die Werte von `codings_mean` und `codings_log_var` untersuchen wollen. Die einzige Ausgabe, die wir verwenden werden, ist die letzte (`codings`). Bauen wir jetzt den Decoder:

```

decoder_inputs = keras.layers.Input(shape=[codings_size])

x = keras.layers.Dense(100, activation="selu")(decoder_inputs)

x = keras.layers.Dense(150, activation="selu")(x)

x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)

outputs = keras.layers.Reshape([28, 28])(x)

variational_decoder = keras.Model(inputs=[decoder_inputs], outputs=[outputs])

```

Für den Decoder hätten wir statt der Functional API die Sequential API verwenden können, da es sich einfach nur um einen Stapel Schichten handelt, die mit vielen der schon von uns gebauten Decoder identisch sind. Bauen wir nun noch das Variational-Autoencoder-Modell:

```

_, _, codings = variational_encoder(inputs)

reconstructions = variational_decoder(codings)

variational_ae = keras.Model(inputs=[inputs], outputs=[reconstructions])

```

Beachten Sie, dass wir die ersten beiden Ausgaben des Encoders ignorieren (wir wollen nur die Codings an den Decoder übergeben). Schließlich müssen wir noch den latenten Verlust und den

Rekonstruktionsverlust hinzufügen:

```
latent_loss = -0.5 * K.sum(  
    1 + codings_log_var - K.exp(codings_log_var) - K.square(codings_mean),  
    axis=-1)  
  
variational_ae.add_loss(K.mean(latent_loss) / 784.)  
  
variational_ae.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

Wir wenden erst [Formel 17-4](#) an, um den latenten Verlust für jede Instanz im Batch zu berechnen (wir summieren über die letzte Achse). Dann berechnen wir den mittleren Verlust über alle Instanzen im Batch und dividieren das Ergebnis durch 784, um sicherzustellen, dass es im Vergleich zum Rekonstruktionsverlust den richtigen Maßstab besitzt. Tatsächlich wird der Rekonstruktionsverlust des Variational Autoencoder als Summe der Rekonstruktionsfehler der Pixel angenommen, aber wenn Keras den "binary_crossentropy"-Verlust ermittelt, berechnet es statt der Summe den Mittelwert über alle 784 Pixel. Also ist der Rekonstruktionsverlust 784 Mal kleiner, als wir ihn benötigen. Wir könnten einen eigenen Verlust definieren, um statt des Mittelwerts die Summe zu berechnen, aber es ist einfacher, den latenten Verlust durch 784 zu teilen (der finale Verlust wird 784 Mal kleiner sein, als er sein sollte, aber das heißt nur, dass wir eine größere Lernrate verwenden sollten).

Beachten Sie, dass wir den RMSprop-Optimierer verwenden, der in diesem Fall gut funktioniert. Und schließlich können wir den Autoencoder trainieren!

```
history = variational_ae.fit(X_train, X_train, epochs=50, batch_size=128,  
                             validation_data=[X_valid, X_valid])
```

Fashion-MNIST-Bilder erzeugen

Nutzen wir nun diesen Variational Autoencoder, um Bilder zu erzeugen, die wie Modeartikel aussehen. Dazu brauchen wir nur zufällige Sample-Codings aus einer gaußschen Verteilung, die wir dann decodieren:

```
codings = tf.random.normal(shape=[12, codings_size])  
  
images = variational_decoder(codings).numpy()
```

[Abbildung 17-13](#) zeigt die zwölf erzeugten Bilder.

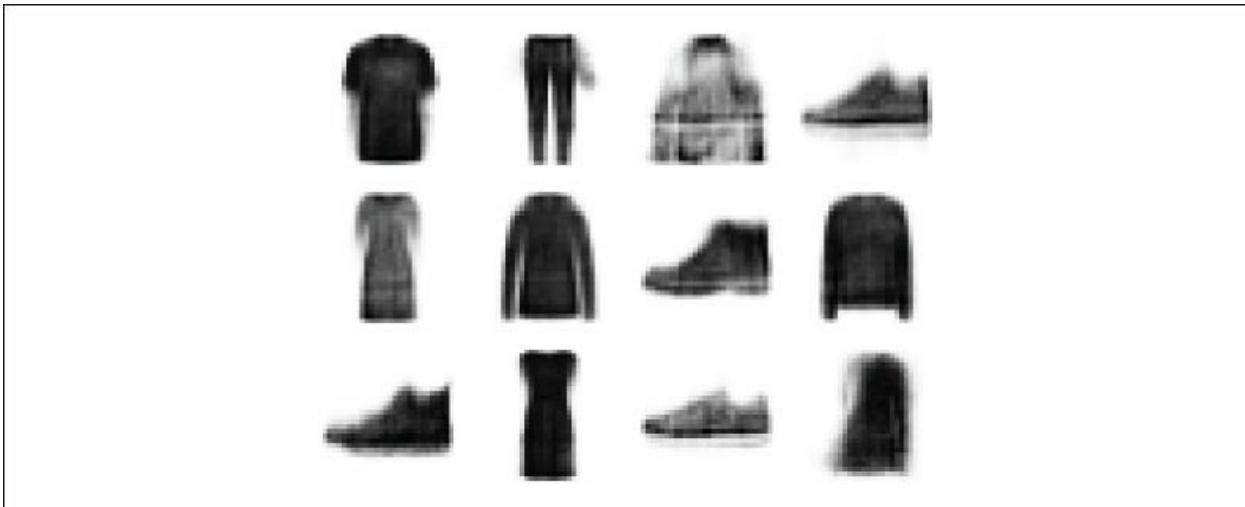


Abbildung 17-13: Fashion-MNIST-Bilder, die durch den Variational Autoencoder generiert wurden

Der Großteil der Bilder sieht recht überzeugend aus, wenn auch ein bisschen zu unscharf. Der Rest ist nicht toll, aber seien Sie mit dem Autoencoder nicht zu streng – er hatte nur ein paar Minuten zum Lernen! Verpassen Sie ihm ein bisschen mehr Fine-Tuning und Trainingszeit, sollten diese Bilder auch besser aussehen.

Variational Autoencoder ermöglichen es, eine *semantische Interpolation* durchzuführen: Statt zwei Bilder auf Pixelebene zu interpolieren (was so aussehen würde, als ob die Bilder überlagert würden), können wir auf Coding-Ebene interpolieren. Wir lassen erst beide Bilder durch den Encoder laufen, interpolieren dann die zwei erhaltenen Codings und decodieren schließlich die interpolierten Codings, um das endgültige Bild zu erhalten. Das wird dann wie ein normales Fashion-MNIST-Bild aussehen, aber es wird ein Mittelding zwischen den ursprünglichen Bildern sein. Im folgenden Codebeispiel nehmen wir die zwölf Codings, die wir eben erzeugt haben, ordnen sie in einem 3×4 -Raster an und nutzen die TensorFlow-Funktion `tf.image.resize()`, um dieses Raster auf 5×7 anzupassen. Standardmäßig führt die `resize()`-Funktion eine bilineare Interpolation durch, daher wird jede zweite Zeile und Spalte interpolierte Codings enthalten. Dann nutzen wir den Decoder, um all die Bilder zu erzeugen:

```
codings_grid = tf.reshape(codings, [1, 3, 4, codings_size])  
  
larger_grid = tf.image.resize(codings_grid, size=[5, 7])  
  
interpolated_codings = tf.reshape(larger_grid, [-1, codings_size])  
  
images = variational_decoder(interpolated_codings).numpy()
```

In [Abbildung 17-14](#) sehen Sie die Ergebnisse. Die ursprünglichen Bilder sind mit einem Rahmen versehen, der Rest sind die Ergebnisse der semantischen Interpolation zwischen den benachbarten Bildern. Beachten Sie beispielsweise, wie der Schuh in der vierten Zeile und fünften Spalte eine nette Interpolation zwischen den beiden Schuhen darüber und darunter ist.

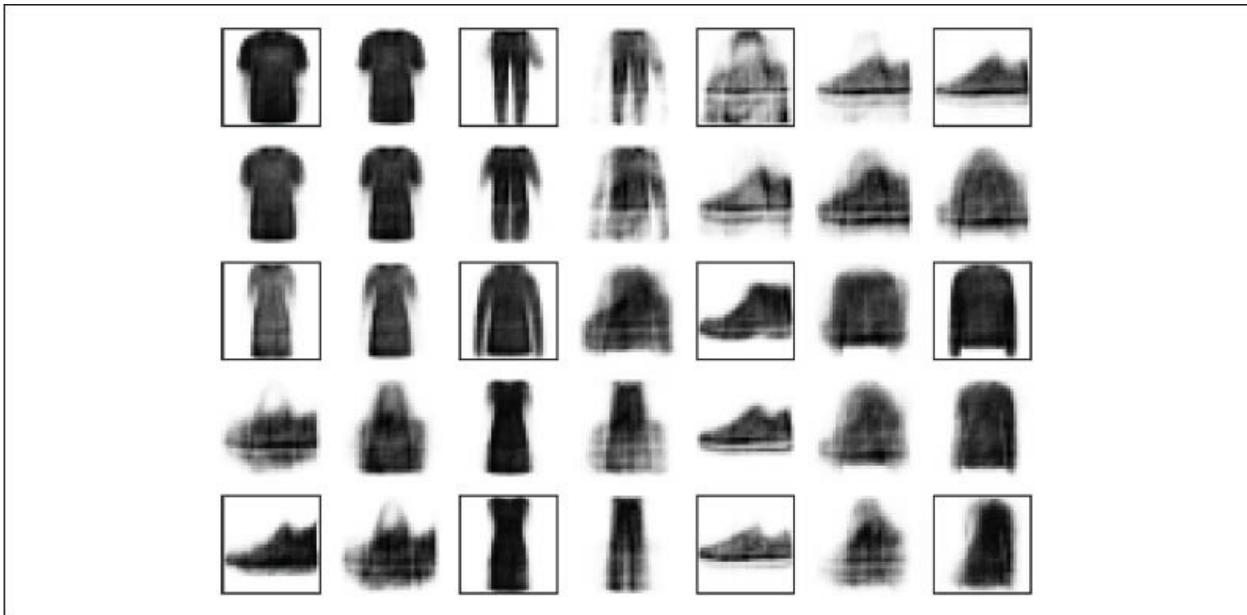


Abbildung 17-14: Semantische Interpolation

Viele Jahre lang waren Variational Autoencoder ziemlich beliebt, aber sie wurden schließlich von GANs überholt, insbesondere weil diese viel realistischere und schärfere Bilder erzeugen können. Wenden wir unsere Aufmerksamkeit daher nun den GANs zu.

Generative Adversarial Networks

Generative Adversarial Networks wurden in einem Artikel (<https://homl.info/gan>)¹⁰ von Ian Goodfellow et al. aus dem Jahr 2014 vorgeschlagen. Die Idee begeisterte Forscher fast sofort, aber es dauerte trotzdem ein paar Jahre, bis die Schwierigkeiten beim Trainieren von GANs überwunden werden konnten. Wie viele tolle Ideen sah es im Rückblick ganz einfach aus: Lasse neuronale Netze gegeneinander antreten und hoffe, dass dieser Wettbewerb sie zur Exzellenz treibt. Wie in Abbildung 17-15 gezeigt, besteht ein GAN aus zwei neuronalen Netzen:

Generator

Erwartet eine Zufallsverteilung als Eingabe (meist gaußförmig) und gibt etwas Datentypisches aus, zum Beispiel ein Bild. Sie können sich die zufälligen Eingaben als die latenten Repräsentationen (also die Codings) des zu erzeugenden Bilds vorstellen. Wie Sie sehen, bietet der Generator die gleiche Funktionalität wie ein Decoder in einem Variational Autoencoder, und er kann auch genauso verwendet werden, um neue Bilder zu erzeugen (übergeben Sie ihm einfach gaußsches Rauschen, erhalten Sie ein nagelneues Bild). Aber wie wir gleich sehen werden, wird er ganz anders trainiert.

Diskriminator

Übernimmt entweder ein künstliches Bild aus dem Generator oder ein echtes Bild aus dem Trainingsdatensatz als Eingabe und muss abschätzen, ob das Eingabebild echt oder künstlich ist.

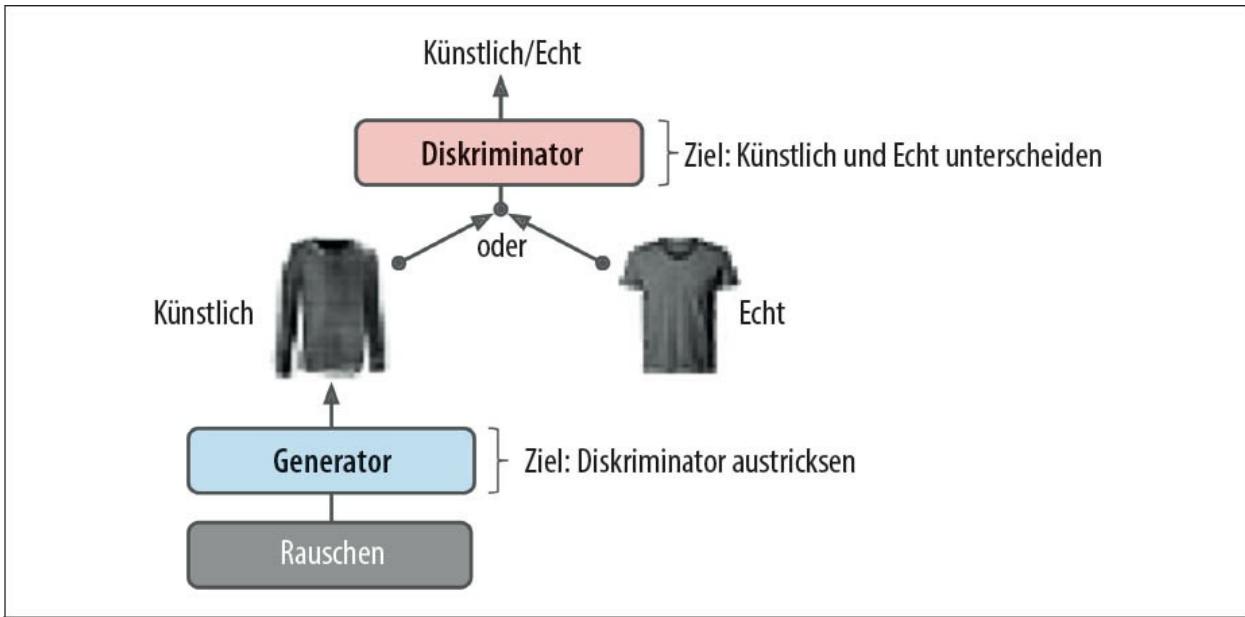


Abbildung 17-15: Ein Generative Adversarial Network

Während des Trainings haben Generator und Diskriminator entgegengesetzte Ziele: Der Diskriminator versucht, künstliche und echte Bilder voneinander zu unterscheiden, während der Generator versucht, Bilder zu erzeugen, die so real aussehen, dass der Diskriminator getäuscht wird. Da das GAN aus zwei Netzen mit unterschiedlichen Zielen besteht, kann es nicht wie ein normales neuronales Netz trainiert werden. Jede Trainingsiteration wird in zwei Phasen unterteilt:

- In der ersten Phase trainieren wir den Diskriminator. Ein Batch mit echten Bildern wird aus dem Trainingsdatensatz gesampelt und durch eine gleiche Zahl von künstlichen Bildern ergänzt, die der Generator erzeugt hat. Die Labels werden für die künstlichen Bilder auf 0 und für die echten Bilder auf 1 gesetzt, und der Diskriminator wird Batch für Batch auf diese Labels trainiert, wobei der binäre Kreuzentropie-Verlust zum Einsatz kommt. Wichtig ist, dass die Backpropagation in dieser Phase nur die Gewichte des Diskriminators optimiert.
- In der zweiten Phase trainieren wir den Generator. Erst nutzen wir ihn, um einen weiteren Batch mit künstlichen Bildern zu erzeugen, und erneut wird der Diskriminator verwendet, um zu erfahren, ob die Bilder künstlich oder echt sind. Dieses Mal stecken wir keine echten Bilder in den Batch, und alle Labels werden auf 1 (echt) gesetzt: Mit anderen Worten – wir wollen, dass der Generator Bilder erzeugt, von denen der Diskriminator (fälschlicherweise) glaubt, dass sie echt sind! Entscheidend ist, dass die Gewichte des Diskriminators in diesem Schritt eingefroren werden, damit die Backpropagation nur die Gewichte des Generators beeinflusst.

Der Generator sieht niemals echte Bilder, er lernt nur nach und nach, überzeugende künstliche Bilder zu produzieren! Er bekommt lediglich die Gradienten, die zurück durch den Diskriminator fließen. Je besser der Diskriminator wird, desto mehr Informationen über die echten Bilder sind glücklicherweise in diesen Gradienten aus zweiter Hand enthalten, daher kann der Generator

deutliche Fortschritte erzielen.

Bauen wir nun ein einfaches GAN für Fashion MNIST.

Als Erstes müssen wir den Generator und den Diskriminator bauen. Der Generator entspricht einem Decoder des Autoencoder, während der Diskriminator ein normaler binärer Klassifikator ist (er übernimmt ein Bild als Eingabe und endet mit einer Dense-Schicht mit einer einzelnen Einheit, die die Sigmoid-Aktivierungsfunktion nutzt). Für die zweite Phase jeder Trainingsiteration brauchen wir zudem das vollständige GAN-Modell mit dem Generator, auf den der Diskriminator folgt:

```
codings_size = 30

generator = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[codings_size]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

discriminator = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(1, activation="sigmoid")
])

gan = keras.models.Sequential([generator, discriminator])
```

Als Nächstes müssen wir diese Modelle kompilieren. Da es sich beim Diskriminator um einen binären Klassifikator handelt, können wir den binären Kreuzentropie-Verlust nutzen. Der Generator wird nur über das gan-Modell trainiert, daher müssen wir ihn gar nicht kompilieren. Das gan-Modell ist ebenfalls ein binärer Klassifikator, daher kann es auch den binären Kreuzentropie-Verlust einsetzen. Wichtig ist, dass der Diskriminator nicht während der zweiten Phase trainiert werden soll, daher machen wir ihn nicht trainierbar, bevor wir das gan-Modell kompilieren:

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
```

```
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

Das Attribut `trainable` wird von Keras nur berücksichtigt, wenn es ein Modell kompiliert, daher ist der `discriminator` nach dem Ausführen dieses Codes durchaus trainierbar, wenn wir dessen Methode `fit()` oder `train_on_batch()` aufrufen (was wir tun werden). Er ist hingegen *nicht* trainierbar, wenn wir diese Methoden für das `gan`-Modell aufrufen.

Da die Trainingsschleife ungewöhnlich ist, können wir nicht die normale `fit()`-Methode einsetzen. Stattdessen werden wir eine eigene Trainingsschleife schreiben. Dazu müssen wir erst ein `Dataset` erzeugen, um über die Bilder zu iterieren:

```
batch_size = 32

dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)

dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

Nun können wir die Trainingsschleife schreiben. Verpacken wir sie in einer Funktion `train_gan()`:

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs=50):

    generator, discriminator = gan.layers

    for epoch in range(n_epochs):

        for X_batch in dataset:

            # Phase 1 - Den Diskriminatator trainieren

            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator(noise)

            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)

            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)

            discriminator.trainable = True

            discriminator.train_on_batch(X_fake_and_real, y1)

            # Phase 2 - Den Generator trainieren

            noise = tf.random.normal(shape=[batch_size, codings_size])

            y2 = tf.constant([[1.]] * batch_size)

            discriminator.trainable = False

            gan.train_on_batch(noise, y2)
```

```
train_gan(gan, dataset, batch_size, codings_size)
```

Sie können die beiden Phasen jeder Iteration sehen:

- In Phase 1 füttern wir den Generator mit gaußschem Rauschen, um künstliche Bilder zu erzeugen. Wir ergänzen diesen Batch mit einer gleichen Zahl von echten Bildern. Die Ziele y_1 werden auf 0 für die künstlichen Bilder und auf 1 für echte Bilder gesetzt. Dann trainieren wir den Diskriminatator mit diesem Batch. Beachten Sie, dass wir das Attribut `trainable` des Diskriminators auf `True` setzen – damit vermeiden wir eine Warnung von Keras, wenn es bemerkt, dass `trainable` jetzt `False` ist, während es beim Kompilieren des Modells noch `True` war (oder umgekehrt).
- In Phase 2 füttern wir das GAN mit gaußschem Rauschen. Sein Generator wird künstliche Bilder erzeugen, und der Diskriminatator wird versuchen, herauszufinden, ob diese Bilder echt oder künstlich sind. Wir wollen, dass der Diskriminatator glaubt, diese künstlichen Bilder seien echt, daher werden die Ziele y_2 auf 1 gesetzt. Beachten Sie, dass wir – wieder zum Vermeiden einer Warnung – das Attribut `trainable` auf `False` setzen.

Das ist alles! Zeigen Sie die erzeugten Bilder an (siehe Abbildung 17-16), werden Sie feststellen, dass sie am Ende der ersten Epoche schon langsam wie (sehr verrauschte) Fashion-MNIST-Bilder aussehen.

Leider werden die Bilder auch nicht mehr viel besser, und Sie werden sogar Epochen erleben können, bei denen das GAN vergessen zu haben scheint, was es gelernt hat. Woran liegt das? Nun, es stellt sich heraus, dass das Trainieren eines GAN eine ziemliche Herausforderung sein kann. Schauen wir uns an, woran das liegt.



Abbildung 17-16: Vom GAN nach einer Trainingsepoke generierte Bilder

Schwierigkeiten beim Trainieren von GANs

Während des Trainings versuchen Generator und Diskriminator fortlaufend, sich gegenseitig in einem Nullsummenspiel auszutricksen. Mit fortschreitendem Training kann das Spiel in einem Status enden, den Spieltheoretiker als *Nash-Gleichgewicht* bezeichnen (benannt nach dem Mathematiker John Nash): Dann ist es für keinen der beiden Spieler besser, seine Strategie zu ändern, wenn der andere es auch nicht tut. So wird beispielsweise ein Nash-Gleichgewicht erreicht, wenn jeder auf der linken Seite der Straße fährt: Keinem Fahrer würde es besser ergehen, wenn er alleine auf die andere Seite wechselt. Natürlich gibt es noch ein zweites mögliches Nash-Gleichgewicht – wenn jeder auf der *rechten* Seite der Straße fährt. Unterschiedliche Ausgangszustände und Dynamiken können zu dem einen oder anderen Gleichgewicht führen. In diesem Beispiel gibt es eine einzige optimale Strategie, nachdem ein Gleichgewicht erreicht wurde (also auf derselben Seite zu fahren wie alle anderen), aber zu einem Nash-Gleichgewicht können auch mehrere konkurrierende Strategien gehören (zum Beispiel jagt ein Jäger seine Beute, die Beute versucht, zu entkommen, und keiner wäre besser dran, wenn er seine Strategie ändert).

Wie wirkt sich das nun auf GANs aus? Nun, die Autoren des Artikels haben gezeigt, dass ein GAN nur ein einziges Nash-Gleichgewicht erreichen kann: wenn der Generator perfekte realistische Bilder erzeugt und der Diskriminator gezwungen ist, zu raten (50% echt, 50% künstlich). Das klingt sehr vielversprechend: Sie müssen anscheinend das GAN nur lange genug trainieren, dann wird es dieses Gleichgewicht erreichen, und Sie erhalten einen perfekten Generator. Aber leider ist es nicht so einfach: Niemand garantiert, dass dieses Gleichgewicht jemals erreicht wird.

Die größte Schwierigkeit nennt sich *Mode Collapse* – wenn die Ausgabe des Generators nach und nach immer weniger unterschiedlich wird. Wie kann das geschehen? Stellen Sie sich vor, dass der Generator beim Generieren überzeugender Schuhe besser wird als für alle anderen Kategorien. Dann wird er den Diskriminator mit den Schuhen ein bisschen mehr täuschen, und das wird ihn darin bestärken, noch mehr Bilder mit Schuhen zu erzeugen. Nach und nach wird er vergessen, wie er etwas anderes generieren kann. Währenddessen wird der Diskriminator als künstliche Bilder nur Schuhe zu sehen bekommen und ebenfalls vergessen, wie er künstliche Bilder anderer Kategorien erkennt. Schließlich schafft es der Diskriminator, die künstlichen Schuhbilder von den echten zu unterscheiden, und der Generator ist gezwungen, sich einer anderen Kategorie zuzuwenden. Vielleicht wird er bei Shirts besser, vergisst die Schuhe, und der Diskriminator folgt ihm dorthin. Das GAN dreht sich nach und nach um ein paar Kategorien und wird niemals in einer von ihnen richtig gut.

Und weil Generator und Diskriminator fortlaufend gegeneinander kämpfen, oszillieren ihre Parameter und werden instabil. Das Training beginnt vielleicht ordentlich und divergiert dann aufgrund dieser Instabilitäten plötzlich ohne offensichtlichen Grund. Und weil diese komplexen Dynamiken von vielen Faktoren beeinflusst werden, reagieren GANs sehr empfindlich auf die Hyperparameter: Sie müssen eventuell sehr viel Aufwand treiben, um sie zu optimieren.

Diese Probleme haben Forscher seit 2014 ziemlich beschäftigt. Viele Artikel wurden zu diesem Thema veröffentlicht. Manche haben neue Kostenfunktionen vorgeschlagen¹¹ (allerdings hat ein Artikel (<https://homl.info/ganseqal>)¹² aus dem Jahr 2018 deren Effizienz infrage gestellt),

andere Techniken zum Stabilisieren des Trainings oder zum Vermeiden des Mode Collapse. So besteht beispielsweise eine verbreitete Technik namens *Experience Replay* darin, die vom Generator bei jeder Iteration erzeugten Bilder in einem Replay Buffer abzulegen (und dabei Schritt für Schritt ältere erzeugte Bilder zu verwerfen) und den Diskriminator mit echten Bildern plus künstlichen Bildern aus diesem Buffer zu trainieren (statt nur die vom aktuellen Generator erzeugten künstlichen Bilder zu nutzen). Damit verringert sich die Wahrscheinlichkeit, dass der Diskriminator zu sehr auf die Ausgaben des letzten Generators overfittet. Eine andere verbreitete Technik nennt sich *Mini-Batch Discrimination*: Sie misst, wie ähnlich sich die Bilder eines Batchs sind, und stellt diese Statistik dem Diskriminator zur Verfügung, sodass er problemlos einen ganzen Batch mit künstlichen Bildern verwerfen kann, wenn diese nicht abwechslungsreich genug sind. Damit wird der Generator darin gefördert, variantenreichere Bilder zu erzeugen und damit die Gefahr des Mode Collapse zu reduzieren. Andere Artikel schlagen einfach bestimmte Architekturen vor, die eine gute Performance liefern.

Kurz gesagt, ist das weiterhin ein sehr aktives Forschungsfeld, und die Dynamiken von GANs sind immer noch nicht ganz verstanden. Aber die gute Nachricht ist, dass große Fortschritte gemacht wurden und manche der Ergebnisse wirklich erstaunlich sind! Schauen wir uns also einige der erfolgreichsten Architekturen an, beginnend mit Deep Convolutional GANs, die vor ein paar Jahren State-of-the-Art waren. Dann wenden wir uns zwei aktuelleren (und komplexeren) Architekturen zu.

Deep Convolutional GANs

Der ursprüngliche GAN-Artikel aus dem Jahr 2014 hat mit Convolutional Layers experimentiert, aber nur versucht, kleine Bilder zu erzeugen. Kurz darauf versuchten viele Forscher, GANs zu bauen, die für größere Bilder auf Deeper Convolutional Networks basierten. Das stellte sich als knifflig heraus, da das Training sehr instabil war, aber Alec Radford et al. konnten schließlich Ende 2015 Erfolge vorweisen, nachdem sie mit vielen verschiedenen Architekturen und Hyperparametern experimentiert hatten. Sie nannten ihre Architektur Deep Convolutional GANs (<https://homl.info/dcgan>) (DCGANs).¹³ Dies sind die wichtigsten Richtlinien, die sie zum Bauen stabiler Convolutional GANs vorgeschlagen haben:

- Ersetzen Sie alle Pooling Layers durch Strided Convolutions (im Diskriminator) und Transposed Convolutions (im Generator).
- Nutzen Sie Batchnormalisierung sowohl im Generator wie auch im Diskriminator, außer in der Ausgabeschicht des Generators und in der Eingabeschicht des Diskriminators.
- Entfernen Sie vollständig verbundene verborgene Schichten für tiefere Architekturen.
- Nutzen Sie die ReLU-Aktivierung im Generator für alle Schichten außer für die Ausgabeschicht, bei der tanh zum Einsatz kommen sollte.
- Nutzen Sie die Leaky-RELU-Aktivierung im Diskriminator für alle Schichten.

Diese Richtlinien funktionieren in den meisten Fällen, aber nicht immer, daher werden Sie eventuell trotzdem mit unterschiedlichen Hyperparametern experimentieren müssen (tatsächlich reicht es manchmal aus, den Zufalls-Seed zu ändern und das gleiche Modell erneut zu trainieren). Hier sehen Sie beispielsweise ein kleines DCGAN, das mit Fashion MNIST

ausreichend gut funktioniert:

```
codings_size = 100

generator = keras.models.Sequential([
    keras.layers.Dense(7 * 7 * 128, input_shape=[codings_size]),
    keras.layers.Reshape([7, 7, 128]),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding="same",
                               activation="selu"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2, padding="same",
                               activation="tanh")
])

discriminator = keras.models.Sequential([
    keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
                       activation=keras.layers.LeakyReLU(0.2),
                       input_shape=[28, 28, 1]),
    keras.layers.Dropout(0.4),
    keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
                       activation=keras.layers.LeakyReLU(0.2)),
    keras.layers.Dropout(0.4),
    keras.layers.Flatten(),
    keras.layers.Dense(1, activation="sigmoid")
])

gan = keras.models.Sequential([generator, discriminator])
```

Der Generator hat Codings der Größe 100, projiziert sie auf 6272 Dimensionen ($7 \times 7 \times 128$) und passt die Form des Ergebnisses an, um einen $7 \times 7 \times 128$ -Tensor zu erhalten. Dieser Tensor wird batchnormalisiert und an einen Transposed Convolutional Layer mit einer Schrittweite von 2 übergeben, der ihn von 7×7 auf 14×14 hochrechnet und seine Tiefe von 128 auf 64 reduziert. Das Ergebnis wird erneut batchnormalisiert und an einen weiteren Transposed Convolutional

Layer mit einer Schrittweite von 2 übergeben, der ihn von 14×14 auf 28×28 hochskaliert und die Tiefe von 64 auf 1 verringert. Diese Schicht nutzt tanh als Aktivierungsfunktion, daher liegen die Ausgaben zwischen -1 und 1 . Aus diesem Grund müssen wir den Trainingsdatensatz vor dem Trainieren des GAN auf den gleichen Bereich umskalieren. Zudem müssen wir seine Form anpassen, um die Kanaldimension hinzuzufügen:

```
X_train = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # Reshape und Rescale
```

Der Diskriminator sieht sehr wie ein normales CNN zur Binärklassifikation aus, nur dass er keine Max-Pooling Layers zum Herunterrechnen des Bilds nutzt, sondern Strided Convolutions (**strides=2**). Beachten Sie zudem, dass wir als Aktivierungsfunktion Leaky ReLU verwenden.

Insgesamt haben wir die DCGAN-Richtlinien beachtet, mit Ausnahme der Batch Normalization-Schichten im Diskriminator, die wir durch Dropout-Layers ersetzt haben (ansonsten wäre das Training in diesem Fall instabil gewesen), zudem haben wir im Generator statt ReLU SELU verwendet. Sie dürfen gerne an dieser Architektur herumspielen – Sie werden sehen, wie empfindlich sie auf die Hyperparameter reagiert (insbesondere auf die relativen Lernraten der beiden Netze).

Um schließlich den Datensatz zu bauen und dieses Modell zu kompilieren und zu trainieren, verwenden wir genau den gleichen Code wie zuvor. Nach 50 Trainingsepochen erzeugt der Generator Bilder wie in [Abbildung 17-17](#). Sie sind immer noch nicht perfekt, aber viele davon sind ziemlich überzeugend.



Abbildung 17-17: Vom DCGAN erzeugte Bilder nach 50 Trainingsepochen

Skalieren Sie diese Architektur hoch und trainieren sie mit einem großen Datensatz mit Gesichtern, können Sie recht realistische Bilder erhalten. Tatsächlich können DCGANs auch latente Repräsentationen mit einer gewissen Bedeutung lernen, wie Sie in [Abbildung 17-18](#) sehen: Es wurden viele Bilder erzeugt und neun davon manuell ausgewählt (oben links), unter anderem drei Männer mit Brille, drei Männer ohne Brille und drei Frauen ohne Brille. Für jede

dieser Kategorien wurden die Codings, mit denen die Bilder erzeugt wurden, gemittelt, und basierend auf diesen mittleren Codings wurde ein Bild erzeugt (unten links). Kurz gesagt: Jedes der drei Bilder unten links repräsentiert den Mittelwert der drei Bilder darüber. Aber das ist nicht einfach ein simpler Durchschnitt auf Pixelebene (das würde zu drei überlappenden Gesichtern führen), sondern der Durchschnitt wurde im latenten Raum berechnet, sodass die Bilder immer noch normale Gesichter zeigen. Berechnen Sie nun Männer mit Brille, ziehen Männer ohne Brille ab, addieren Frauen ohne Brille – jeweils in Bezug auf die mittleren Codings – und erzeugen dann das Bild zu diesem Coding, erhalten Sie das Bild in der Mitte des 3×3 -Rasters auf der rechten Seite: eine Frau mit Brille! Die acht anderen Bilder im Raster wurden mit dem gleichen Vektor erzeugt, dem ein bisschen Rauschen beigemischt wurde, um die semantischen Interpolationsfähigkeiten von DCGANs aufzuzeigen. Das Durchführen solcher Rechnungen mit Gesichtern fühlt sich wie Science-Fiction an!

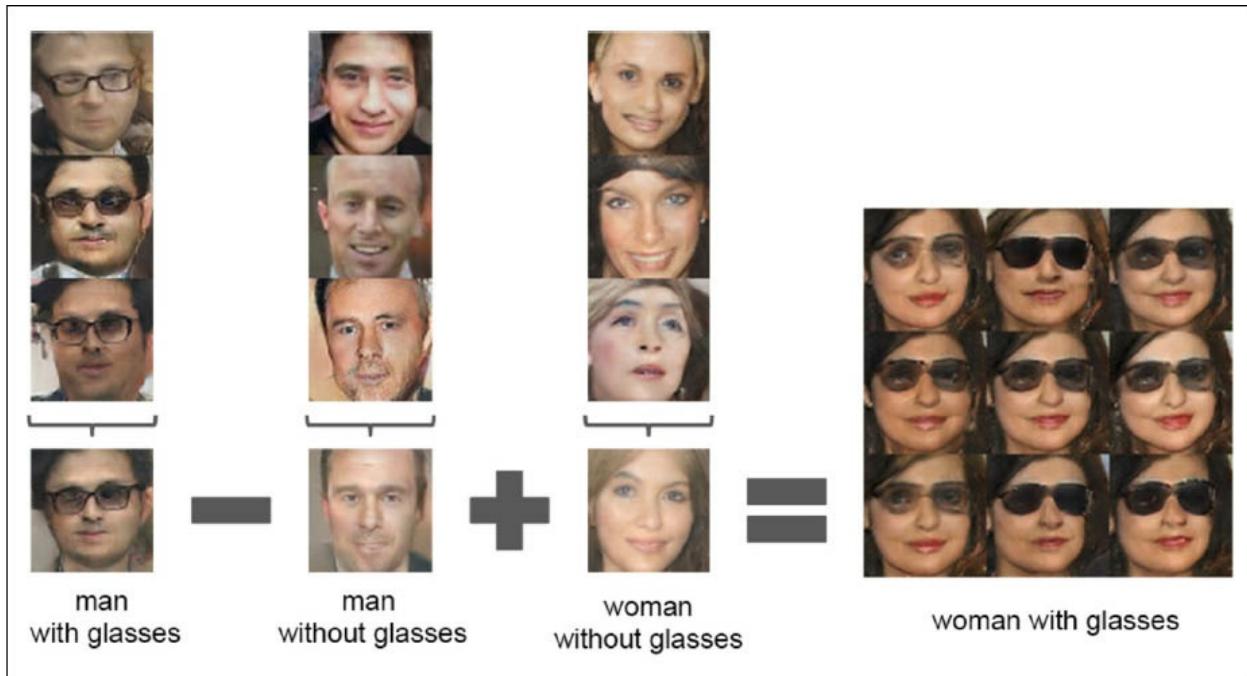


Abbildung 17-18: Vektorarithmetik für visuelle Konzepte (Teil von Abbildung 7 aus dem DCGAN-Artikel)¹⁴

- ☞ Übergeben Sie die Kategorie jedes Bilds als zusätzliche Eingabe an den Generator und den Diskriminatoren, werden beide lernen, wie jede Kategorie aussieht, und Sie werden die Kategorien jedes Bilds steuern können, das vom Generator erzeugt wird. Das nennt sich *Conditional GAN* (<https://homl.info/cgan>) (CGAN)¹⁵.

DCGANs sind aber noch nicht perfekt. Versuchen Sie beispielsweise, sehr große Bilder damit zu erzeugen, erhalten Sie oft lokal überzeugende Merkmale, insgesamt aber Inkonsistenzen (wie zum Beispiel Shirts, bei denen ein Arm viel länger als der andere ist). Wie können Sie das beheben?

Progressive wachsende GANs

Eine wichtige Technik wurde in einem Artikel (<https://homl.info/progan>)¹⁶ von den Nvidia-Forschern Tero Karras et al. aus dem Jahr 2018 vorgeschlagen: Man erzeugt zu Beginn des Trainings kleine Bilder und fügt dann nach und nach Convolutional Layers zu Generator und Diskriminator hinzu, um immer größere Bilder zu erzeugen ($4 \times 4, 8 \times 8, 16 \times 16, \dots, 512 \times 512, 1024 \times 1024$). Dieses Vorgehen ähnelt dem schichtweisen Trainieren der Stacked Autoencoder. Die zusätzlichen Schichten werden am Ende des Generators und am Anfang des Diskriminators hinzugefügt, und die vorher trainierten Schichten bleiben dabei trainierbar.

Lässt man beispielsweise die Ausgaben des Generators von 4×4 auf 8×8 wachsen (siehe Abbildung 17-19), wird zum bestehenden Convolutional Layer eine Upsampling-Schicht (mit Nearest Neighbor Filtering) hinzugefügt, sodass er 8×8 -Feature-Maps ausgibt, die dann an den neuen Convolutional Layer übergeben werden (der "same"-Padding und eine Schrittweite von 1 verwendet, sodass seine Ausgaben ebenfalls 8×8 sind). Auf diese neue Schicht folgt ein neuer Convolutional Layer zur Ausgabe – ein normaler Convolutional Layer mit Kernelgröße 1, der die Ausgaben auf die gewünschte Anzahl an Farbkanälen reduziert (zum Beispiel 3). Um zu vermeiden, dass die trainierten Gewichte des ersten Convolutional Layer unbrauchbar werden, wenn der neue Convolutional Layer hinzugefügt wird, handelt es sich bei der abschließenden Ausgabe um eine gewichtete Summe der ursprünglichen Ausgabeschicht (die nun 8×8 -Feature-Maps ausgibt) und der neuen Ausgabeschicht. Das Gewicht der neuen Ausgaben ist α , während das Gewicht der ursprünglichen Ausgaben $1 - \alpha$ ist, wobei α langsam von 0 nach 1 wächst. Mit anderen Worten: Die neuen Convolutional Layers (in Abbildung 17-19 dargestellt durch gestrichelte Linien) werden nach und nach eingeblendet, während die ursprünglichen Ausgabeschichten nach und nach verschwinden. Eine ähnliche Fade-in-Fade-out-Technik kommt zum Einsatz, wenn der Diskriminator um einen neuen Convolutional Layer ergänzt wird (gefolgt von einem Average-Pooling Layer zum Herunterrechnen).

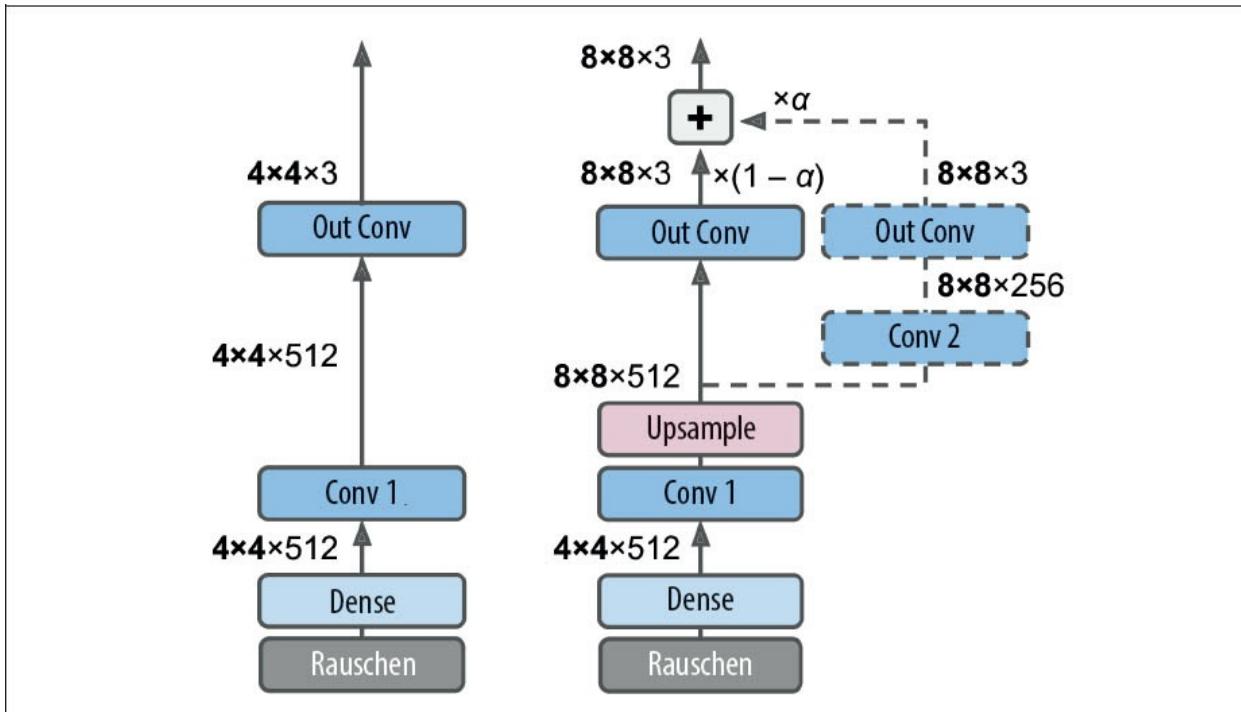


Abbildung 17-19: Progressiv wachsendes GAN: Ein GAN-Generator gibt 4×4 -Farbbilder aus (links); wir erweitern ihn für eine Ausgabe von 8×8 -Bildern (rechts).

In dem Artikel wurden noch eine Reihe weiterer Techniken vorgestellt, die die Diversität der Ausgaben steigern (um einen Mode Collapse zu vermeiden) und das Training stabiler machen sollen:

Mini-Batch-Standardabweichungsschicht

Wird gegen Ende des Diskriminators hinzugefügt. Für jede Position in den Eingaben wird die Standardabweichung über alle Kanäle und alle Instanzen im Batch berechnet ($S = \text{tf.math.reduce_std(inputs, axis=[0, -1])}$). Diese Standardabweichungen werden dann über alle Punkte gemittelt, um einen einzelnen Wert zu erhalten ($v = \text{tf.reduce_mean}(S)$). Schließlich wird eine zusätzliche Feature Map zu jeder Instanz im Batch hinzugefügt und mit dem berechneten Wert gefüllt ($\text{tf.concat}([\text{inputs}, \text{tf.fill}([\text{batch_size}, \text{height}, \text{width}, 1], v)], axis=-1)$). Wie kann das helfen? Nun, wenn der Generator Bilder mit nur wenig Abwechslung erzeugt, wird es eine kleine Standardabweichung über die Feature Maps im Diskriminator geben. Dank dieser Schicht kann der Diskriminat or problemlos auf diese Statistik zugreifen und, er lässt sich weniger leicht von einem Generator austricksen, der zu wenig Diversität erzeugt. Das wird den Generator darin bestärken, abwechslungsreichere Ausgaben zu generieren und damit das Risiko eines Mode Collapse zu verringern.

Ausgeglichene Lernrate

Initialisiert alle Gewichte nicht mit der He-Initialisierung, sondern mit einer einfachen Normalverteilung mit dem Mittelwert 0 und der Standardabweichung 1. Aber die Gewichte werden zur Laufzeit (also jedes Mal, wenn die Schicht ausgeführt wird) um den gleichen Faktor wie bei der He-Initialisierung herunterskaliert: Sie werden durch $\sqrt{2/n_{\text{inputs}}}$ geteilt, wobei n_{inputs} die Anzahl an Eingaben für die Schicht ist. Der Artikel hat gezeigt, dass diese Technik die Leistung des GAN deutlich steigert, wenn RMSProp, Adam oder andere adaptive Gradientenoptimierer zum Einsatz kommen. Tatsächlich normalisieren diese Optimierer die Gradientenaktualisierungen mit ihrer geschätzten Standardabweichung (siehe [Kapitel 11](#)), daher müssen Parameter mit einem größeren Dynamikumfang¹⁷ länger trainiert werden, während Parameter mit einem kleinen Dynamikumfang eventuell zu schnell aktualisiert werden, was zu Instabilitäten führen kann. Durch ein Umskalieren der Gewichte als Teil des Modells selbst statt nur bei der Initialisierung stellt dieser Ansatz sicher, dass der Dynamikumfang für alle Parameter während des Trainings der gleiche ist, sodass sie alle mit der gleichen Geschwindigkeit lernen. Das beschleunigt und stabilisiert das Training.

Normalisierungsschicht auf Pixelebene

Wird nach jedem Convolutional Layer im Generator hinzugefügt. Sie normalisiert jede Aktivierung basierend auf allen Aktivierungen im gleichen Bild und an der gleichen Position, aber über alle Kanäle (geteilt durch die Wurzel der mittleren quadrierten Aktivierung). In TensorFlow-Code ist das `inputs / tf.sqrt(tf.reduce_mean(tf.square(x), axis=-1, keepdims=True) + 1e-8)` (der Glättungsterm `1e-8` wird benötigt, um eine Division durch null zu vermeiden). Diese

Technik vermeidet Explosionen in den Aktivierungen aufgrund eines exzessiven Wettbewerbs zwischen Generator und Diskriminator.

Die Kombination all dieser Techniken hat es den Autoren erlaubt, außerordentlich überzeugende, hochauflöste Bilder von Gesichtern zu generieren (<https://homl.info/progandemo>). Aber was genau meinen wir mit »überzeugend«? Die Evaluation ist eine der großen Herausforderungen bei der Arbeit mit GANs: Es ist zwar möglich, die Diversität der generierten Bilder zu evaluieren, aber es ist eine viel schwierigere und subjektivere Aufgabe, deren Qualität zu bewerten. Eine Technik ist der Einsatz menschlicher Bewerter, aber das ist teuer und zeitaufwendig. Daher schlugen die Autoren vor, die Ähnlichkeit zwischen der lokalen Bildstruktur der generierten und der Trainingsbilder zu messen und dabei jede Skala zu berücksichtigen. Diese Idee führte zu einer weiteren bahnbrechenden Innovation: StyleGANs.

StyleGANs

Das Generieren hochauflösender Bilder wurde wieder einmal vom gleichen Nvidia-Team mit einem Artikel (<https://homl.info/stylegan>)¹⁸ aus dem Jahr 2018 weiter vorangebracht. Darin wird die beliebte StyleGAN-Architektur vorgestellt. Die Autoren nutzten *Style-Transfer*-Techniken im Generator, um sicherzustellen, dass die generierten Bilder auf jeder Skalenebene die gleiche lokale Struktur wie die Trainingsbilder besitzen, womit die Qualität der erzeugten Bilder sehr stark verbessert wurde. Der Diskriminator und die Verlustfunktion wurden nicht verändert – nur der Generator. Schauen wir uns das StyleGAN an. Es besteht aus zwei Netzwerken (siehe Abbildung 17-20).

Mapping Network

Ein MLP mit acht Schichten bildet die latenten Repräsentationen \mathbf{z} (also die Codings) auf einen Vektor \mathbf{w} ab. Dieser Vektor wird dann durch mehrere *affine Transformationen* geschickt (also Dense-Schichten ohne Aktivierungsfunktionen, die in Abbildung 17-20 durch die A-Kästchen symbolisiert sind), die mehrere Vektoren erzeugen. Diese Vektoren steuern den Stil des generierten Bilds auf unterschiedlichen Stufen, von feingranularen Texturen (zum Beispiel der Haarfarbe) bis hin zu High-Level-Features (zum Beispiel Erwachsene oder Kinder). Kurz gesagt, das Mapping Network bildet die Codings auf mehrere Stilvektoren ab.

Synthesis Network

Verantwortlich für das Generieren der Bilder. Es besitzt eine konstante erlernte Eingabe (zur Klarstellung: die Eingabe wird *nach* dem Training konstant sein, während des Trainings wird sie per Backpropagation angepasst) und verarbeitet diese Eingabe wie zuvor über mehrere Convolutional und Upsampling Layers, aber es gibt zwei Veränderungen: Die Eingabe und alle Ausgaben der Convolutional Layers werden mit etwas Rauschen versehen (vor der Aktivierungsfunktion). Und auf jede Rauschschicht folgt eine *Adaptive-Instance-Normalization* -(AdaIN)-Schicht: Sie standardisiert jede Feature Map unabhängig voneinander (indem sie den Mittelwert der Feature Map von ihr abzieht und durch ihre Standardabweichung teilt), dann nutzt sie den Stilvektor, um Skala und Offset jeder Feature Map zu bestimmen (der Stilvektor enthält eine Skala und einen Bias-Term für jede Feature

Map).

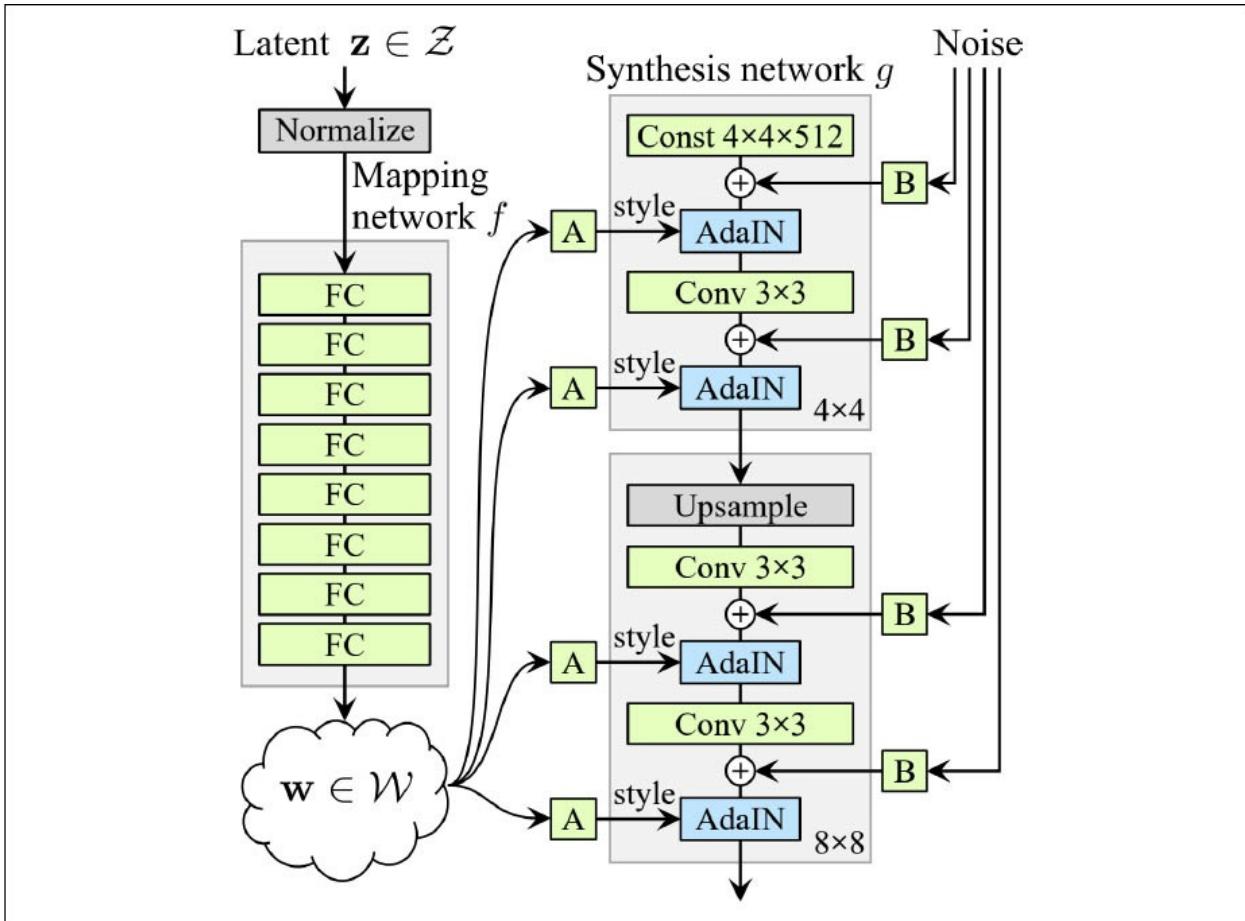


Abbildung 17-20: Die Generator-Architektur der StyleGANs (Teil von Abbildung 1 aus dem StyleGAN-Artikel)¹⁹

Es ist sehr wichtig, das Rauschen unabhängig von den Codings hinzuzufügen. Manche Teile eines Bilds sind ziemlich zufällig, wie zum Beispiel die genaue Position jedes Fleckens oder jedes Haars. In früheren GANs musste diese Zufälligkeit entweder aus den Codings oder aus pseudozufälligem Rauschen kommen, das vom Generator selbst erzeugt wurde. Kam sie aus den Codings, hieß das, dass der Generator einen signifikanten Teil der Repräsentation der Codings abzweigen musste, um Rauschen zu steuern – eine ziemliche Verschwendungen. Zudem musste das Rauschen durch das Netz fließen und die letzten Schichten des Generators erreichen können: Das schien eine unnötige Einschränkung zu sein, die das Training vermutlich verlangsamt hat. Und schließlich tauchen manche visuellen Artefakte eventuell auf, weil das gleiche Rauschen auf unterschiedlichen Ebenen genutzt wurde. Wenn der Generator stattdessen versucht, sein eigenes pseudozufälliges Rauschen zu erzeugen, mag dieses Rauschen nicht sehr überzeugend aussehen und zu mehr visuellen Artefakten führen. Zudem würden Teile der Gewichte des Generators für das Erzeugen des pseudozufälligen Rauschens eingesetzt werden, was ebenfalls nach Verschwendungen aussieht. Durch zusätzliche Rauscheingaben werden all diese Probleme vermieden – das GAN kann das bereitgestellte Rauschen für die richtige Menge an Stochastizität

in jedem Teil des Bilds einsetzen.

Das hinzugefügte Rauschen ist für jede Ebene anders. Jede Rauscheingabe besteht aus einer einzelnen Feature Map voll mit gaußschem Rauschen, das an alle Feature Maps (der gegebenen Stufe) verteilt und mithilfe gelernter Per-Feature-Skalierungsfaktoren angepasst wird (dargestellt in Abbildung 17-20 durch die B-Boxen).

Schließlich setzt StyleGAN noch eine Technik namens *Mixing Regularization* (oder *Style Mixing*) ein, bei der ein Prozentsatz der generierten Bilder mit zwei verschiedenen Codings erzeugt wird. Genauer gesagt, werden die Codings c_1 und c_2 durch das Mapping Network geschickt, was zu zwei Stilvektoren w_1 und w_2 führt. Dann generiert das Synthesis Network ein Bild basierend auf den Stilen w_1 für die ersten Ebenen und mit den Stilen w_2 für die restlichen Ebenen. Die Grenze wird zufällig ausgewählt. Das verhindert, dass das Netz davon ausgeht, dass Stile auf aneinander grenzenden Ebenen in Korrelation zueinander stehen, was wiederum eine Lokalität im GAN unterstützt – jeder Stilvektor beeinflusst nur eine begrenzte Zahl von Eigenschaften im generierten Bild.

Es gibt so viele verschiedene GANs da draußen, dass ein ganzes Buch erforderlich wäre, um sie alle abzudecken. Hoffentlich hat Ihnen diese Einführung die wichtigsten Ideen vermitteln können – und vor allem den Wunsch, mehr darüber zu lernen. Kämpfen Sie mit einem mathematischen Konzept, gibt es vermutlich Blogposts, die Ihnen helfen, es besser zu verstehen. Dann versuchen Sie, Ihr eigenes GAN zu implementieren. Seien Sie nicht enttäuscht, wenn das Lernen zuerst schwierig ist: Leider ist das normal, und Sie brauchen ein bisschen Geduld, bis es funktioniert, aber die Ergebnisse sind es wert. Schlagen Sie sich mit einem Implementierungsdetail herum, gibt es viele Keras- oder TensorFlow-Implementierungen, die Sie sich anschauen können. Wollen Sie nur schnell Ergebnisse haben, können Sie auch ein vortrainiertes Modell nutzen (es gibt zum Beispiel vortrainierte StyleGAN-Modelle für Keras).

Im nächsten Kapitel werden wir uns einem ganz anderen Zweig des Deep Learning zuwenden – dem Deep Reinforcement Learning.

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 Autoencoder evaluieren?
4. Was sind unvollständige und übervollständige Autoencoder? Welches Hauptrisiko besteht bei einem extrem unvollständigen Autoencoder? Welches bei einem übervollständigen Autoencoder?
5. Wie lassen sich die Gewichte eines Stacked Autoencoder miteinander koppeln? Warum tut man dies?

6. Was ist ein generatives Modell? Können Sie eine Untergruppe der generativen Autoencoder benennen?
7. Was ist ein GAN? Können Sie ein paar Aufgaben benennen, bei denen GANs brillieren?
8. Was sind die größten Schwierigkeiten beim Trainieren von GANs?
9. Versuchen Sie, einen Denoising Autoencoder zum Vortrainieren eines Bildklassifikators einzusetzen. Sie können MNIST verwenden (die einfachste Option) oder einen komplexeren Bilddatensatz wie CIFAR10 (<https://homl.info/122>), wenn Sie eine größere Herausforderung suchen. Folgen Sie unabhängig vom Datensatz diesen Schritten:
 - 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 Bilder, die jedes Neuron in der codierenden Schicht am stärksten aktivieren.
 - Konstruieren Sie ein DNN zur Klassifikation unter Verwendung der ersten Schichten des Autoencoder. Trainieren Sie es mit nur 500 Bildern des Trainingsdatensatzes. Funktioniert es besser mit oder ohne Vortraining?
10. Trainieren Sie einen Variational Autoencoder mit einem Bilddatensatz Ihrer Wahl und nutzen Sie ihn, um Bilder zu generieren. Alternativ können Sie versuchen, einen ungelabelten Datensatz zu finden, an dem Sie interessiert sind, und neue Beispiele zu generieren.
11. Trainieren Sie ein DCGAN, um einen Bilddatensatz zu verarbeiten und damit Bilder zu generieren. Fügen Sie Experience Replay hinzu, um zu sehen, ob es hilft. Verwandeln Sie es in ein Conditional GAN, bei dem Sie die generierten Klassen steuern können.

Lösungen zu diesen Aufgaben finden Sie in [Anhang A](#).

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 Laufe 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 (<https://homl.info/dqn>) vorführten, das jedes Atari-Spiel von null auf erlernen konnte² und irgendwann Menschen überlegen war (<https://homl.info/dqn2>).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 März 2016 im Sieg des Systems AlphaGo über Lee Sedol gipfelte, einer Legende im Brettspiel *Go*, und im Mai 2017 gegen den Weltmeister Ke Jie. 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 durch neue Ideen in einer großen Anwendungsbreite. DeepMind wurde 2014 von Google für über 500 Millionen Dollar aufgekauft.

Wie hat DeepMind das alles geschafft? 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, um ein Modell zum Balancieren einer Stange auf einem beweglichen Wagen zu entwickeln, dann werde ich die TF-Agents-Bibliothek vorstellen, die aktuellste Algorithmen nutzt, mit denen sich das Bauen leistungsfähiger RL-Systeme stark vereinfachen lässt. Wir werden diese Bibliothek nutzen, um einen Agenten zu bauen, der das berühmte Atari-Spiel *Breakout* spielt. Beenden werde ich das Kapitel mit einem Blick auf einige der aktuellen Entwicklungen in diesem Bereich.

Lernen zum Optimieren von Belohnungen

Beim Reinforcement Learning sammelt ein Software-Agent *Beobachtungen*, führt *Aktionen* innerhalb einer *Umwelt* oder *Umgebung* durch und erhält dafür *Belohnungen*. Das Ziel des Agenten ist es, 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 18-1](#)):

1. Der Agent könnte ein Programm sein, das einen 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 oder in die falsche Richtung läuft.
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 verstettern 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.

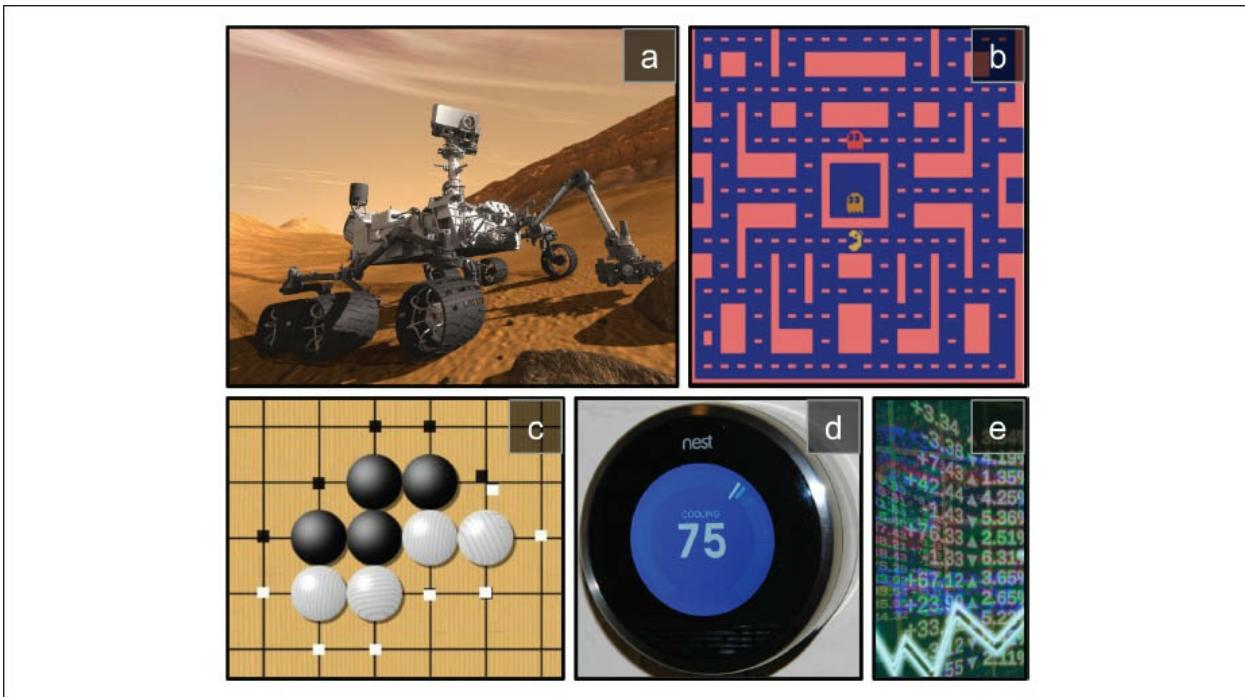


Abbildung 18-1: Beispiele für Reinforcement Learning: (a) Robotik, (b) Ms. Pac-Man, (c) Go-Spieler, (d) Thermostat, (e) automatischer Börsenspekulant⁵

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, Empfehlungssysteme, 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 zur 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 18-2).

Die Policy kann ein beliebiger Algorithmus sein, der nicht einmal deterministisch arbeiten muss. Manchmal ist es gar nicht erforderlich, die Umwelt zu beobachten. 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?

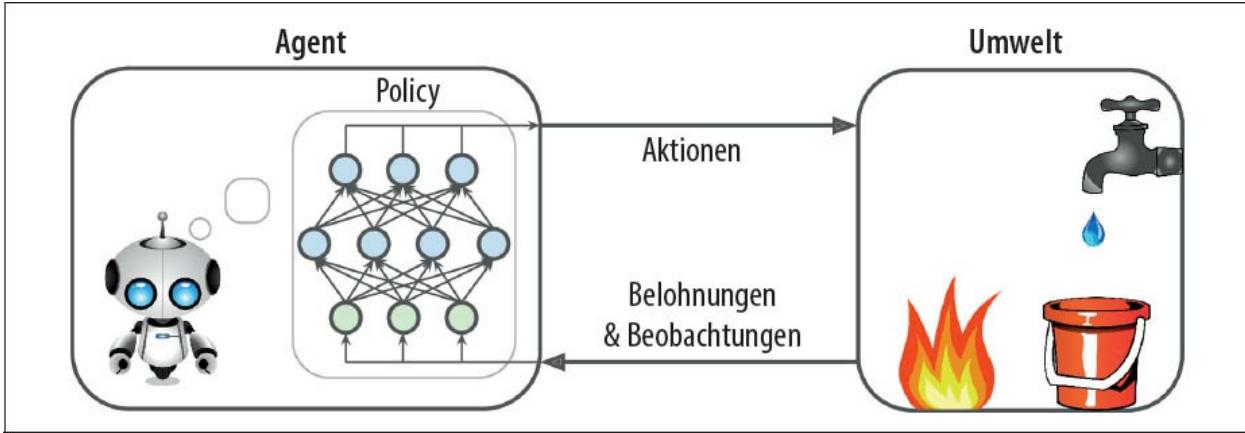


Abbildung 18-2: Reinforcement Learning mit einem neuronalen Netz als Policy

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 18-3). Das ist ein Beispiel für *Policy-Suche*, in diesem Fall mit 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 schlechtesten 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.⁸ 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.⁹ Diesen Ansatz nennt man *Policy-Gradienten* (PG). Wir werden ihn später in diesem Kapitel im Detail besprechen. Im Beispiel des Staubsaugerroboters 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.

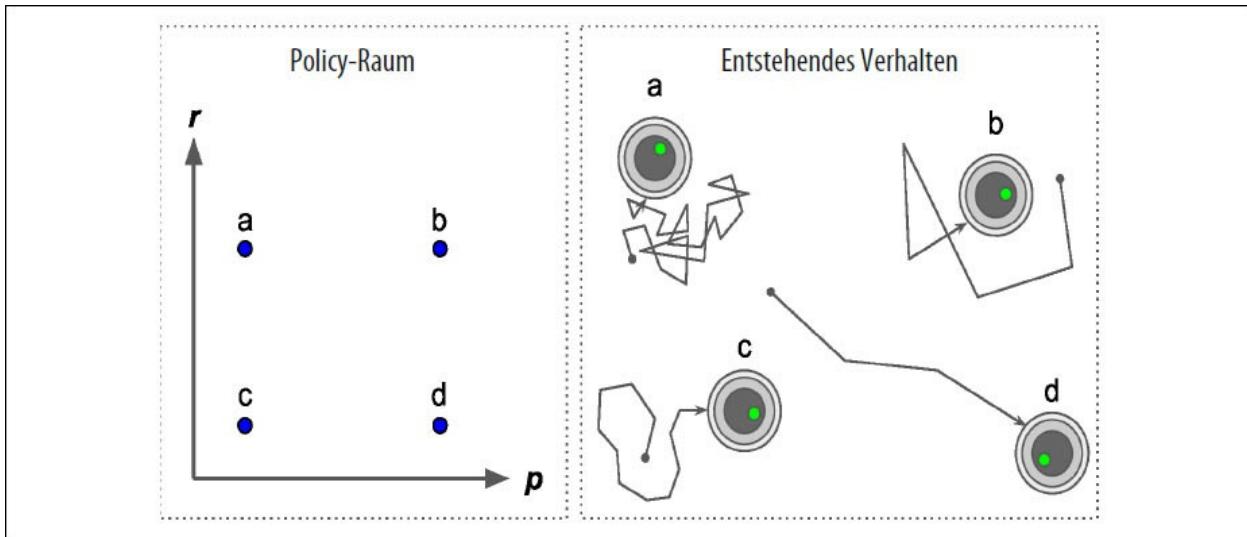


Abbildung 18-3: Vier Punkte im Policy-Raum (links) und das entsprechende Verhalten des Agenten (rechts)

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, 1.000 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. Sie können beispielsweise eine Bibliothek wie PyBullet (<https://pybullet.org/>) oder MuJoCo (<http://www.mujoco.org/>) für eine 3-D-Physiksimulation nutzen.

OpenAI Gym (<https://gym.openai.com/>)¹⁰ 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.

Falls Sie eine isolierten Umgebung mit `virtualenv` erstellt haben, müssen Sie diese erst aktivieren, bevor Sie das Toolkit installieren:

```
$ cd $ML_PATH                      # Ihr ML-Arbeitsverzeichnis (z. B. $HOME/ml)

$ source my_env/bin/activate # unter Linux oder MacOS

$ .\my_env\Scripts\activate # unter Windows
```

Als Nächstes installieren Sie OpenAI Gym (wenn Sie keine virtuelle Umgebung verwenden,

werden Sie die Option `--user` ergänzen müssen, oder Sie benötigen Administratorrechte):

```
$ python3 -m pip install --upgrade gym
```

Abhängig von Ihrem System müssen Sie eventuell auch die Mesa- OpenGL-Utility-(GLU-)Bibliothek installieren (zum Beispiel müssen Sie unter Ubuntu 18.04 `apt install libglu1-mesa` ausführen). Diese Bibliothek ist zum Rendern der ersten Umgebung notwendig. Anschließend öffnen Sie eine Python-Shell oder ein Jupyter-Notebook und erstellen eine Umwelt durch `make()`:

```
>>> import gym  
>>> env = gym.make("CartPole-v1")  
>>> obs = env.reset()  
>>> obs  
array([-0.01258566, -0.00156614, 0.04207708, -0.00180545])
```

Hier haben wir eine CartPole-Umwelt erzeugt. Dabei handelt es sich um eine zweidimensionale Simulation, in der ein Auto nach links oder rechts beschleunigt werden kann, um eine Stange darauf zu balancieren (siehe [Abbildung 18-4](#)). Sie erhalten eine Liste mit allen verfügbaren Umgebungen durch `gym.envs.registry.all()`. Nachdem die Umwelt erstellt wurde, müssen Sie sie mit der Methode `reset()` initialisieren. Das gibt die erste Beobachtung zurück. Beobachtungen hängen von der Art der Umwelt ab. Bei der CartPole-Umwelt handelt es sich bei jeder Beobachtung um ein eindimensionales NumPy-Array mit vier Gleitkommazahlen: Diese stehen für die horizontale Position des Autos (0.0 = Mitte), die Geschwindigkeit (positiv bedeutet rechts), den Winkel der Stange (0.0 = vertikal) und die Winkelgeschwindigkeit (positiv bedeutet im Uhrzeigersinn).

Zeigen wir jetzt die Umwelt durch einen Aufruf der Methode `render()` an (siehe [Abbildung 18-4](#)). Unter Windows müssen Sie dazu erst einen X Server wie VcXsrv oder Xming installieren:

```
>>> env.render()  
True
```

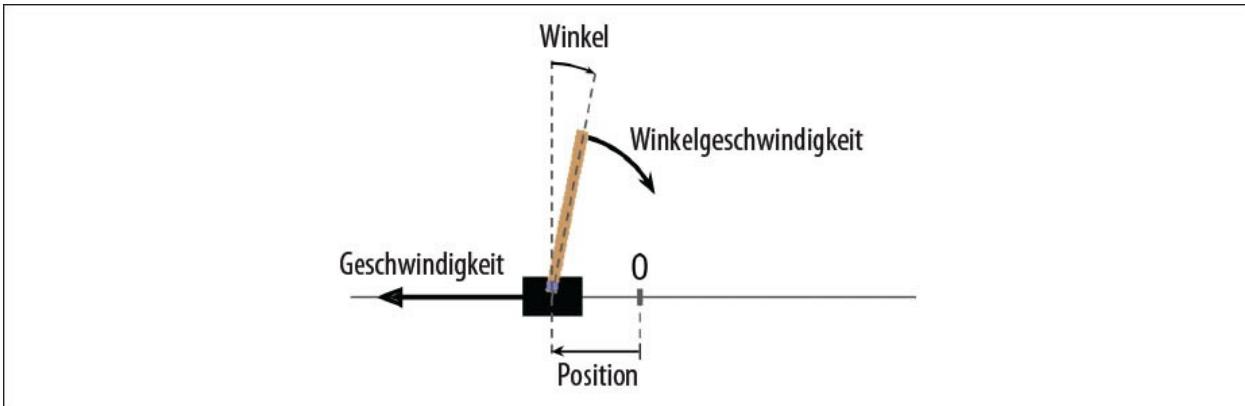


Abbildung 18-4: Die CartPole-Umwelt

 Nutzen Sie einen Headless Server (also ohne Bildschirm), wie zum Beispiel eine virtuelle Maschine in der Cloud, wird das Rendern fehlschlagen. Das können Sie nur mit einem Fake-X-Server wie Xvfb oder Xdummy vermeiden. Sie können beispielsweise Xvfb installieren (`apt install xvfb` unter Ubuntu oder Debian) und Python mit folgendem Befehl starten: `xvfb-run -s "-screen 0 1400x900x24" python3`. Alternativ installieren Sie Xvfb und die pyvirtualdisplay-Bibliothek (<https://homl.info/pyvd>) (die Xvfb verpackt) und führen am Anfang Ihres Programms `pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()` aus.

Wenn Sie `render()` verwenden möchten, um ein gerendertes Bild als NumPy-Array abzulegen, können Sie `mode="rgb_array"` setzen (schrägerweise wird diese Umgebung die Umwelt ebenfalls auf dem Bildschirm rendern):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape # Höhe, Breite, Farbkanäle (3 = Rot, Grün, Blau)
(800, 1200, 3)
```

Befragen wir unsere Umwelt, welche Aktionen möglich sind:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` bedeutet, dass die möglichen Aktionen die Integer-Zahlen 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 (`obs[2] > 0`), beschleunigen wir den Wagen nach rechts:

```
>>> action = 1 # Beschleunigung nach rechts
>>> obs, reward, done, info = env.step(action)
```

```
>>> obs  
array([-0.01261699, 0.19292789, 0.04204097, -0.28092127])  
>>> 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, die Episode so lange wie möglich im Spiel zu bleiben.

`done`

Dieser Wert wird `True`, wenn die Episode vorbei ist. Das passiert, sobald die Stange kippt, vom Bildschirm verschwindet oder nach 200 Schritten (in dem Fall haben Sie gewonnen). Danach muss die Umgebung zurückgesetzt werden, um neu verwendet werden zu können.

`info`

Dieses umgebungsspezifische Dictionary kann zusätzliche Informationen zum Debuggen enthalten. So findet sich hier in manchen Spielen, wie viele Leben der Agent noch hat.



Ist Ihre Arbeit mit einer Umgebung abgeschlossen, sollten Sie deren Methode `close()` aufrufen, um Ressourcen freizugeben.

Programmieren wir eine einfache Policy, die nach links beschleunigt, wenn sich die Stange 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(200):

        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)

(41.718, 8.858356280936096, 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/hanson-ml2>) 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 18-5). Bei der CartPole-Umgebung gibt es lediglich zwei mögliche Aktionen (links und rechts). Daher benötigen wir nur ein Ausgabeneuron, 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.

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 vorherige Beobachtung berücksichtigen, um die Geschwindigkeit zu bestimmen. Ein anderes Beispiel ist, dass 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.

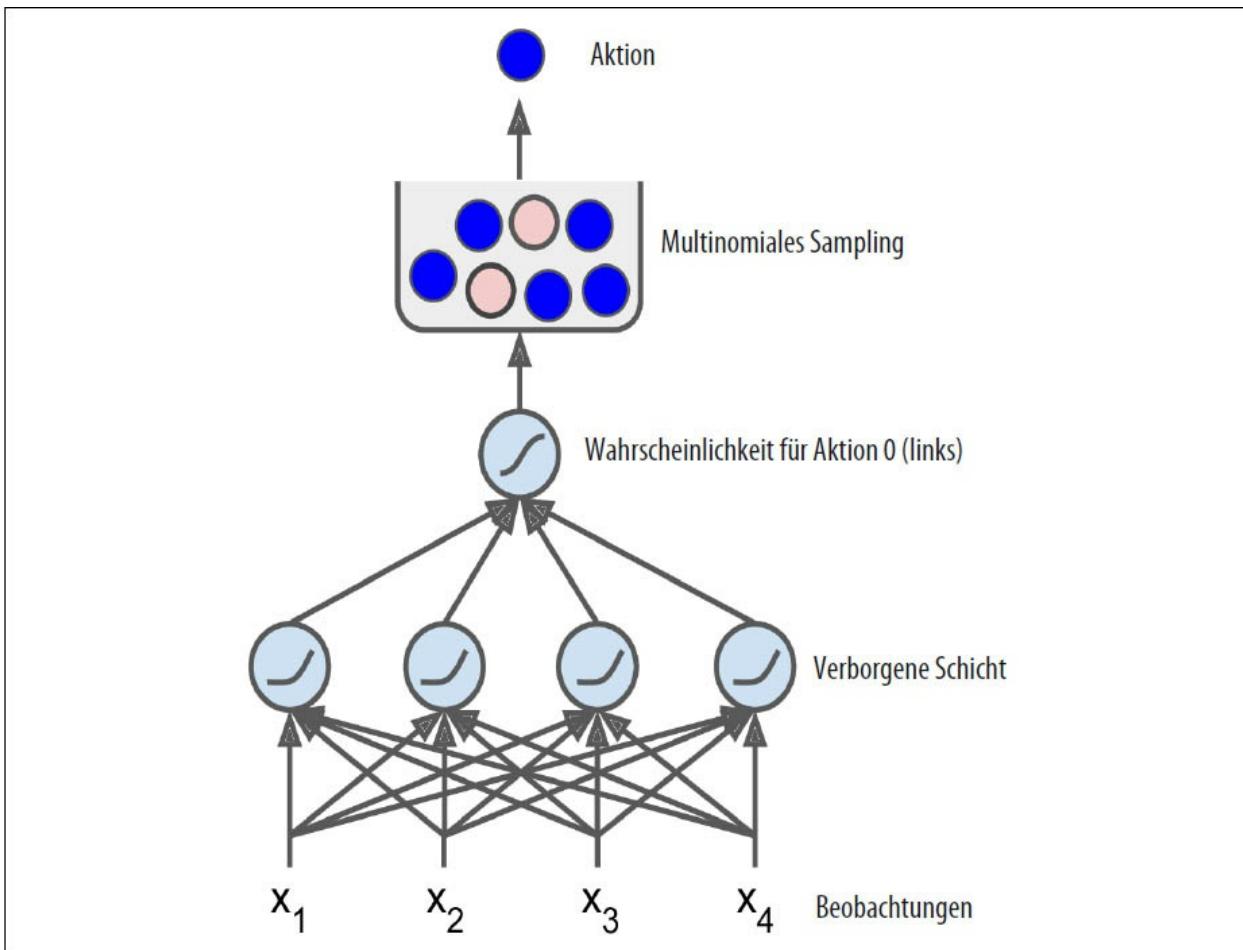


Abbildung 18-5: Ein neuronales Netz als Policy

Der folgende Code erstellt das neuronale Netz für diese Policy mit tf.keras:

```
import tensorflow as tf

from tensorflow import keras

n_inputs = 4 # == env.observation_space.shape[0]

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])


```

Nach den Imports nutzen wir ein einfaches Sequential-Modell, um das Policy-Netz zu definieren. Die Anzahl an Eingaben ist die Größe des Beobachtungsraums (beim CartPole ist das 4), und wir haben nur fünf verborgene Einheiten, weil es sich um ein einfaches Problem handelt. Schließlich wollen wir eine einzelne Wahrscheinlichkeit ausgeben (die Wahrscheinlichkeit, nach links zu fahren), daher haben wir ein einzelnes Ausgabeneuron, das die Sigmoid-Aktivierungsfunktion verwendet. Gäbe es mehr als zwei mögliche Aktionen, bräuchten wir ein Ausgabeneuron pro Aktion, und wir müssten stattdessen die Softmax-Aktivierungsfunktion einsetzen.

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 Wahrscheinlichkeitsverteilung und der Zielwahrscheinlichkeitsverteilung 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 einen *Discount-Faktor* γ an. Diese Summe von Belohnungen mit Discount nennen wir den Return dieser Aktion. 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 18-6), erhalten wir mit einem Discount-Faktor $\gamma = 0,8$ für die erste Aktion einen Return von $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$. Wenn der Discount-Faktor nahe 0 ist, spielen langfristige Belohnungen im Vergleich zu unmittelbaren eine geringe Rolle. Wenn umgekehrt der Discount-Faktor nahe 1 liegt, zählen Belohnungen in ferner Zukunft fast so viel wie unmittelbare. Typische Discount-Faktoren liegen zwischen 0,95 und 0,99. Mit einem Discount-Faktor 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 einem Discount-Faktor 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 ein Discount-Faktor von 0,95 angemessen.

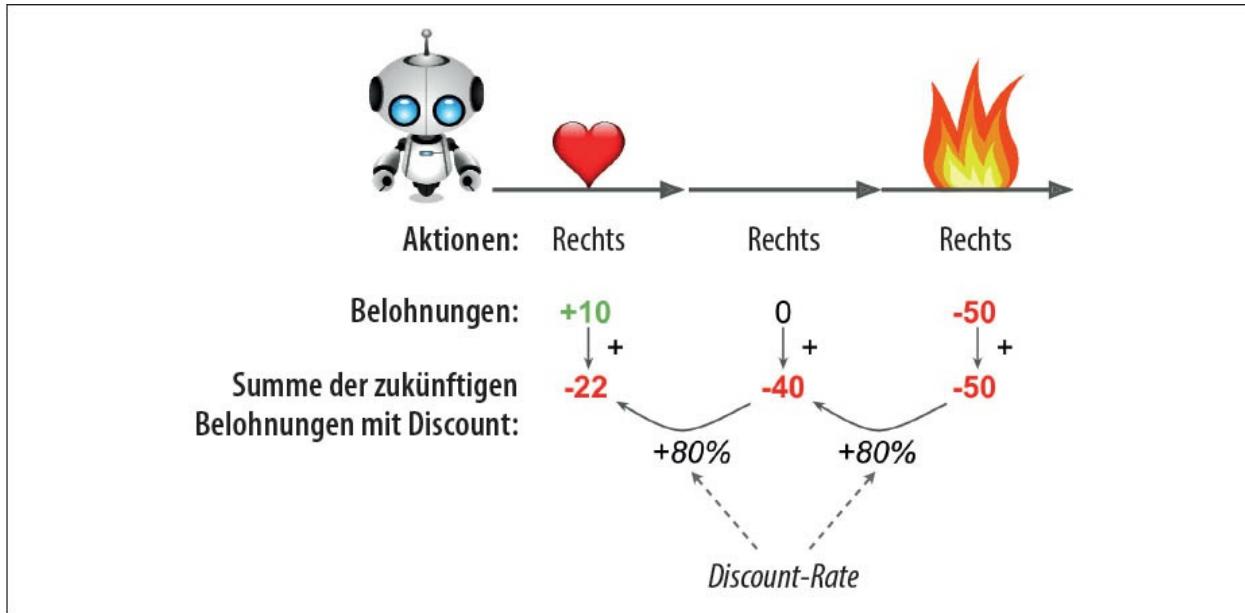


Abbildung 18-6: Den Return einer Aktion berechnen: die Summe der zukünftigen Belohnungen mit Discount

Natürlich können auf eine gute Aktion mehrere schlechte folgen, wodurch die Stange am Ende schnell umfällt und die gute Aktion einen schlechten Return 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 höheren Return als schlechte. Wir wollen schätzen, wie viel besser oder schlechter eine Aktion verglichen mit den anderen möglichen Aktionen im Mittel ist. Das wird als *Aktionsvorteil* bezeichnet. Dazu müssen wir viele Episoden durchführen und sämtliche Returns der Aktionen normalisieren (den Mittelwert abziehen und durch die Standardabweichung teilen). Danach können wir davon ausgehen, dass Aktionen mit einem negativen Vorteil schlecht und Aktionen mit positivem Vorteil gut waren. Perfekt – da wir die Aktionen nun evaluieren können, sollten wir unseren ersten Agenten mit Policy-Gradienten

trainieren. Sehen wir, wie das geht.

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://homl.info/132>).¹¹ Hier folgt eine verbreitete Variante:

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 Vorteil jeder Aktion (nach dem im vorigen Abschnitt beschriebenen Verfahren).
3. Wenn der Vorteil einer Aktion positiv ist, war die Aktion vermutlich gut, und der zuvor berechnete Gradient sollte angewendet werden, um diese Aktion in der Zukunft wahrscheinlicher zu machen. Wenn der Vorteil jedoch negativ ist, war die Aktion vermutlich 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 Vorteil der Aktion multipliziert.
4. Schließlich werden alle erhaltenen Gradientenvektoren gemittelt, und ein Schritt im Gradientenverfahren wird damit durchgeführt.

Implementieren wir diesen Algorithmus mit tf.keras. Wir werden die weiter oben gebaute Neuronale-Netz-Policy trainieren, sodass sie lernt, die Stange auf dem Auto auszubalancieren. Zuerst brauchen wir eine Funktion, die einen Schritt spielt. Wir tun erst einmal so, als wäre jede durchgeführte Aktion die richtige, sodass wir den Verlust und seine Gradienten berechnen können (diese Gradienten werden nur eine Weile lang gespeichert, und wir werden sie später anpassen – abhängig davon, ob sich die Aktion als gut oder schlecht herausgestellt hat):

```
def play_one_step(env, obs, model, loss_fn):  
  
    with tf.GradientTape() as tape:  
  
        left_proba = model(obs[np.newaxis])  
  
        action = (tf.random.uniform([1, 1]) > left_proba)  
  
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)  
  
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))  
  
        grads = tape.gradient(loss, model.trainable_variables)  
  
        obs, reward, done, info = env.step(int(action[0, 0].numpy()))  
  
        return obs, reward, done, grads
```

Schauen wir uns diese Funktion genauer an:

- Im `GradientTape`-Block (siehe [Kapitel 12](#)) beginnen wir damit, das Modell aufzurufen und ihm eine einzelne Beobachtung mitzugeben (wir passen die Beobachtung so an, dass sie zu einem Batch mit einer einzelnen Instanz wird, da das Modell einen Batch erwartet). Ausgegeben wird die Wahrscheinlichkeit, nach links zu gehen.
- Als Nächstes erzeugen wir eine zufällige Gleitkommazahl zwischen 0 und 1 und prüfen, ob sie größer als `left_proba` ist. Die `action` wird mit einer Wahrscheinlichkeit `left_proba` den Wert `False` und mit einer Wahrscheinlichkeit $1 - \text{left_proba}$ den Wert `True` haben. Nach dem Umwandeln dieses booleschen Werts in einen Float wird die Aktion mit der entsprechenden Wahrscheinlichkeit 0 (links) oder 1 (rechts) sein.
- Nun definieren wir die Zielwahrscheinlichkeit dafür, nach links zu gehen: Sie ist 1 minus die Aktion (gecastet in einem Float). Ist die Aktion 0 (links), ist die Zielwahrscheinlichkeit dafür, nach links zu gehen, 1. Ist die Aktion 1 (rechts), ist die Zielwahrscheinlichkeit 0.
- Dann berechnen wir mit der gegebenen Verlustfunktion den Verlust und nutzen das Tape, um die Gradienten des Verlusts in Bezug auf die trainerbaren Variablen des Modells zu berechnen. Diese Gradienten werden später vor der Anwendung noch angepasst – abhängig davon, als wie gut oder schlecht sich die Aktion herausgestellt hat.
- Schließlich spielen wir die ausgewählte Aktion und geben die neue Beobachtung sowie die Belohnung zurück und dazu die Information, ob die Episode beendet ist, und natürlich die gerade berechneten Gradienten.

Erstellen wir nun eine weitere Funktion, die auf `play_one_step()` aufbaut, mehrere Episoden spielt und alle Belohnungen und Gradienten für jede Episode und jeden Schritt zurückgibt:

```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):  
    all_rewards = []  
    all_grads = []  
    for episode in range(n_episodes):  
        current_rewards = []  
        current_grads = []  
        obs = env.reset()  
        for step in range(n_max_steps):  
            obs, reward, done, grads = play_one_step(env, obs, model, loss_fn)  
            current_rewards.append(reward)  
            current_grads.append(grads)  
            if done:  
                break  
    return all_rewards, all_grads
```

```

        break

    all_rewards.append(current_rewards)

    all_grads.append(current_grads)

return all_rewards, all_grads

```

Dieser Code gibt eine Liste mit Belohnungslisten zurück (eine Belohnungsliste pro Episode mit einer Belohnung pro Schritt), dazu eine Liste der Gradientenlisten (eine Gradientenliste pro Episode mit jeweils einem Gradiententupel pro Schritt und einem Gradiententensor pro trainierbare Variable im Tupel).

Der Algorithmus wird die Funktion `play_multiple_episodes()` nutzen, um das Spiel mehrfach zu spielen (zum Beispiel zehn Mal), dann wird er zurückkehren und sich alle Belohnungen anschauen, den Discount darauf anwenden und sie normalisieren. Dazu brauchen wir ein paar Funktionen mehr: Die erste wird die Summe der zukünftigen Belohnungen mit Discount bei jedem Schritt berechnen, die zweite wird alle diese Belohnungen (Returns) mit Discount über viele Episoden normalisieren, indem sie den Mittelwert abzieht und durch die Standardabweichung teilt:

```

def discount_rewards(rewards, discount_factor):

    discounted = np.array(rewards)

    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor

    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):

    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                                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_factor=0.8)
```

```

array([-22, -40, -50])

>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
...                               discount_factor=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 18-6](#)). Sie können überprüfen, dass die Funktion `discount_and_normalize_rewards()` tatsächlich die normierten Vorteile für jede Aktion in beiden Episoden zurückgibt. Beachten Sie, dass die erste Episode viel schlechter als die zweite war. Daher sind alle ihre normierten Vorteile negativ; alle Aktionen aus der ersten Episode würden als schlecht betrachtet, umgekehrt wären alle Aktionen aus der zweiten Episode gut.

Wir sind nun fast so weit, dass wir den Algorithmus laufen lassen können! Definieren wir jetzt die Hyperparameter. Wir werden 150 Trainingsiterationen durchführen, 10 Episoden pro Iteration spielen, und jede Episode wird höchstens 200 Schritte dauern. Wir werden einen Discount-Faktor von 0,95 nutzen:

```

n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95

```

Wir benötigen auch noch einen Optimierer und die Verlustfunktion. Ein normaler Adam-Optimierer mit einer Lernrate von 0,01 wird ausreichen, und wir werden die binäre Kreuzentropie als Verlustfunktion nutzen, weil wir einen binären Klassifizierer trainieren (es gibt zwei mögliche Aktionen: links oder rechts):

```

optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.binary_crossentropy

```

Bauen wir jetzt die Trainingsschleife und führen wir sie aus!

```

for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)

```

```

all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                    discount_factor)

all_mean_grads = []

for var_index in range(len(model.trainable_variables)):

    mean_grads = tf.reduce_mean(
        [final_reward * all_grads[episode_index][step][var_index]
         for episode_index, final_rewards in enumerate(all_final_rewards)
         for step, final_reward in enumerate(final_rewards)], axis=0)

    all_mean_grads.append(mean_grads)

optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))

```

Schauen wir uns nun den Code an:

- Bei jeder Trainingsiteration ruft diese Schleife die Funktion `play_multiple_episodes()` auf, die das Spiel zehn Mal spielt und alle Belohnungen und Gradienten für jede Episode und jeden Schritt zurückgibt.
- Dann rufen wir `discount_and_normalize_rewards()` auf, um den normalisierten Vorteil jeder Aktion zu berechnen (die wir im Code als `final_reward` bezeichnen). Das ergibt, wie gut oder schlecht eine Aktion im Nachhinein tatsächlich war.
- Als Nächstes berechnen wir für jede trainierbare Variable das gewichtete Mittel der Gradienten für diese Variable über alle Episoden und alle Schritte, gewichtet mit dem `final_reward`.
- Schließlich wenden wir diese mittleren Gradienten mithilfe des Optimierers an: Die trainierbaren Variablen des Modells werden angepasst, und die Policy wird hoffentlich ein bisschen besser werden.

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 im Abschnitt »Policy Gradients« in den Jupyter-Notebooks überprüfen). Die mittlere Belohnung pro Episode wird sehr nahe an 200 liegen (was in dieser Umgebung standardmäßig das Maximum ist). Das ist ein Erfolg!



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, weil dies das Training erheblich beschleunigt. Da Sie beispielsweise wissen, dass die Stange so vertikal wie möglich sein sollte, könnten Sie vielleicht eine negative Belohnung hinzufügen, die proportional zu dem Winkel der Stange ist. Dadurch werden die Belohnungen besser, und das Training wird beschleunigt. Auch wenn Sie bereits eine halbwegs gute Policy haben (z.B. eine hartcodierte), sollten Sie diese zunächst durch das neuronale Netz abbilden lassen, bevor Sie es mit Policy-Gradienten verbessern.

Der einfache Algorithmus mit den Policy-Gradienten hat zwar die CartPole-Aufgabe gemeistert, aber bei größeren und komplexeren Aufgaben skaliert er nicht sehr gut. Tatsächlich ist er ausgesprochen *Sample-ineffizient* – er muss also das Spiel sehr lange beobachten, bevor er einen signifikanten Fortschritt erzielen kann. Das liegt daran, dass er viele Episoden laufen lassen muss, um den Vorteil jeder Aktion zu schätzen. Aber er dient als Grundlage für leistungsfähigere Algorithmen, wie zum Beispiel *Actor-Critic*-Algorithmen (die wir am Ende des Kapitels kurz 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, den zu erwartenden Return 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 (darum sagen wir, das System besitze kein Gedächtnis).

[Abbildung 18-7](#) zeigt ein Beispiel einer Markov-Kette mit vier Zuständen.

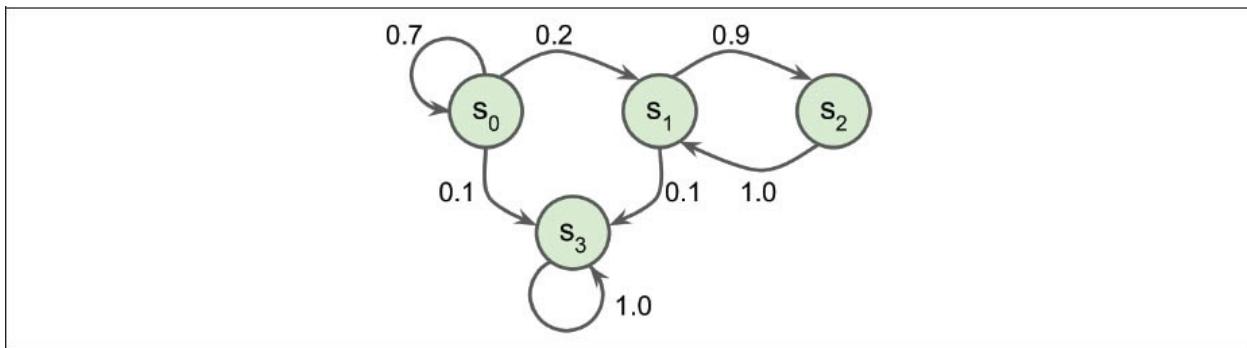


Abbildung 18-7: Beispiel einer Markov-Kette

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.

Markov-Entscheidungsprozesse wurden erstmalig in den 1950ern von Richard Bellman beschrieben (<https://homl.info/133>).¹² 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 18-8 dargestellte MDP drei Zustände (durch Kreise dargestellt) und bei jedem Schritt bis zu drei mögliche diskrete Aktionen (durch Rauten dargestellt).

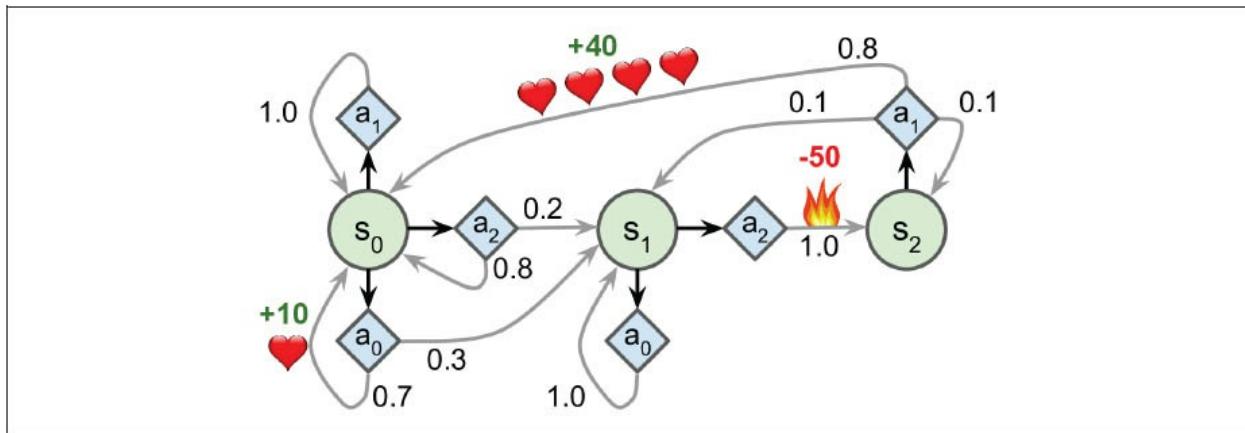


Abbildung 18-8: Beispiel für einen Markov-Entscheidungsprozess

Wenn es im Zustand s_0 beginnt, kann der Agent zwischen den Aktionen a_0 , a_1 und 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. 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 . 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 18-1](#)). 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 18-1: Optimalitätsprinzip von Bellman

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V^*(s')] \text{ für alle } s$$

- $T(s, a, s')$ ist die Übergangswahrscheinlichkeit von Zustand s zu Zustand s' , vorausgesetzt, der Agent wählt Aktion a . So ist beispielsweise in [Abbildung 18-8](#) $T(s_2, a_1, s_0) = 0,8$.
- $R(s, a, s')$ ist die Belohnung, die der Agent beim Übergang von Zustand s in Zustand s' erhält, vorausgesetzt, der Agent wählt Aktion a . In [Abbildung 18-8](#) ist zum Beispiel $R(s_2, a_1, s_0) = 40$.
- 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 18-2](#)). Ein bemerkenswertes Ergebnis ist, dass diese Schätzungen mit genug Zeit garantiert zu den optimalen Zustandswerten konvergieren, die der optimalen Policy entsprechen.

Formel 18-2: Value-Iteration-Algorithmus

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V_k(s')] \text{ für alle } s$$

$V_k(s)$ ist in dieser Gleichung der geschätzte Wert des Zustands s in der k . Iteration des Algorithmus.



Dieser Algorithmus ist ein Beispiel für *dynamische Programmierung*, die ein komplexes Problem in bearbeitbare Teilprobleme aufteilt, die sich iterativ abarbeiten lassen.

Die optimalen Zustandswerte zu kennen, ist vor allem zum Evaluieren einer Policy nützlich. Es liefert uns aber nicht die optimale Policy für den Agenten. Glücklicherweise fand Bellman einen sehr ähnlichen Algorithmus zum Abschätzen der optimalen *Zustands-Aktions-Werte*, genannt *Q-Werte* (Qualitätswerte). Der optimale Q-Wert eines Zustand-Aktion-Paars (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 18-3](#)).

Formel 18-3: Q-Wert-Iterationsalgorithmus

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \cdot \max_{a'} Q_k(s', a') \right] \quad \text{für alle } (s, a)$$

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: $\pi^*(s) = \operatorname{argmax} Q^*(s, a)$.

Wenden wir diesen Algorithmus auf das in [Abbildung 18-8](#) gezeigte MDP an. Zunächst definieren wir das MDP:

```
transition_probabilities = [ # shape=[s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
    [None, [0.8, 0.1, 0.1], None]]
rewards = [ # shape=[s, a, s']
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Um zum Beispiel die Übergangswahrscheinlichkeit von s_2 nach s_0 nach dem Ausspielen von Aktion a_1 zu kennen, schauen wir uns `transition_probabilities[2][1][0]` an (mit dem Wert 0,8). Genauso können wir die zugehörige Belohnung über `rewards[2][1][0]` erhalten (+40). Und für die Liste der möglichen Aktionen in s_2 schauen wir auf `possible_actions[2]` (in diesem Fall ist nur die Aktion a1 möglich). Als Nächstes müssen wir alle Q-Werte mit 0 initialisieren (außer für die unmöglichen Aktionen, für die wir die Q-Werte auf $-\infty$ setzen):

```
Q_values = np.full((3, 3), -np.inf) # -np.inf für unmögliche Aktionen
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0 # für alle möglichen Aktionen
```

Führen wir nun den Q-Wert-Iterationsalgorithmus aus. Er wendet wiederholt [Formel 18-3](#) für alle Q-Werte, jeden Status und jede mögliche Aktion an:

```
gamma = 0.90 # Discount-Faktor
for iteration in range(50):
    Q_prev = Q_values.copy()
```

```

for s in range(3):

    for a in possible_actions[s]:

        Q_values[s, a] = np.sum([
            transition_probabilities[s][a][sp]
            * (rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))]

        for sp in range(3)])

```

Das ist alles! Es ergeben sich folgende Q-Werte:

```

>>> Q_values

array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.          , -inf, -4.87971488],
       [-inf, 50.13365013, -inf]])

```

Befindet sich der Agent beispielsweise im Zustand s_0 und wählt er Aktion a_1 , ist die erwartete Summe der zukünftigen Belohnungen mit Discount beispielsweise in etwa 17,0.

Schauen wir uns jetzt für jeden Zustand die Aktion mit dem höchsten Q-Wert an:

```

>>> np.argmax(Q_values, axis=1) # optimale Aktion für jeden Zustand

array([0, 0, 1])

```

Damit erhalten wir eine optimale Policy für dieses MDP mit einer Discount-Rate von 0,90: Wähle im Zustand s_0 Aktion a_0 , im Zustand s_1 Aktion a_0 (bleibe standhaft) und im Zustand s_2 Aktion a_1 (die einzige Möglichkeit). Interessanterweise ändert sich die Policy, wenn Sie die Discount-Rate auf 0,95 erhöhen: Im Zustand s_1 wird a_2 die beste Aktion (gehe durchs Feuer!). Das ergibt Sinn, weil mit einem höheren Gewicht auf der Gegenwart die Aussicht auf zukünftige Belohnungen den unmittelbaren Schmerz nicht aufwiegt.

Temporal Difference Learning

Reinforcement-Learning-Aufgaben mit diskreten Aktionen lassen sich oft als Markov-Entscheidungsprozesse modellieren. Allerdings ist dem Agenten zu Beginn die Übergangswahrscheinlichkeiten unbekannt (er kennt $T(s, a, s')$ nicht), ebenso wie die Belohnungen (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 dem 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 18-4](#)).

Formel 18-4: TD Learning-Algorithmus

$$V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

oder äquivalent:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r + s')$$

$$\text{mit } \delta_k(s, r + s') = r + \gamma \cdot V_{k+1}(s') - V_k(s)$$

In dieser Gleichung gilt:

- α ist die Lernrate (z.B. 0,01).
- $r + \gamma \cdot V_k(s')$ wird als *TD-Ziel* bezeichnet.
- $\delta_k(s, r, s')$ ist der *TD-Fehler*.

Die erste Form dieser Gleichung lässt sich kürzer auch als $a \xleftarrow{\alpha} b$ schreiben, was $a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k$ bedeutet. Damit kann die erste Zeile von [Formel 18-4](#) wie folgt geschrieben werden:

$$V(s) \xleftarrow{\alpha} r + \gamma \cdot V(s')$$

- ☞ 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 hüpf 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).

Q-Learning

Ähnlich dazu ist der Q-Learning-Algorithmus eine Anpassung des Q-Wert-Iterationsalgorithmus auf Situationen mit zunächst unbekannten Übergangswahrscheinlichkeiten und Belohnungen (siehe [Formel 18-5](#)). Q-Learning beobachtet dazu das Spiel eines Agenten (zum Beispiel ein zufälliges Spielen) und verbessert dabei nach und nach seine Schätzungen der Q-Werte. Hat es exakte (oder ausreichend exakte) Q-Wert-Schätzungen, besteht die optimale Policy darin, die Aktion auszuwählen, die den höchsten Q-Wert besitzt (also die Greedy-Policy).

Formel 18-5: Q-Learning-Algorithmus

$$Q(s, a) \xleftarrow{\alpha} r + \gamma \cdot \max_{a'} Q(s', a')$$

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) . Um diese Summe zu schätzen, nehmen wir das Maximum der Q-Wert-Schätzungen für den nächsten Status s' , da wir davon ausgehen, dass die Zielpolicy von dort optimal handelt.

Implementieren wir den Q-Learning-Algorithmus. Zuerst müssen wir einen Agenten die Umgebung erforschen lassen. Dazu brauchen wir eine Schrittfunktion, sodass der Agent eine Aktion ausführen und den sich daraus ergebenden Zustand und die Belohnung erhalten kann:

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

Implementieren wir jetzt die Policy des Agenten zum Erforschen. Da der Zustandsraum ziemlich klein ist, wird eine einfache Zufallspolicy ausreichen. Lassen wir den Algorithmus lange genug laufen, wird der Agent jeden Zustand mehrfach besucht haben und auch jede mögliche Aktion oft ausprobieren:

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Als Nächstes sind wir nach dem Initialisieren der Q-Werte wie zuvor dazu bereit, den Q-Learning-Algorithmus mit einer abfallenden Lernrate auszuführen (per Power Scheduling, siehe dazu [Kapitel 11](#)):

```
alpha0 = 0.05 # initiale Lernrate
decay = 0.005 # Abfall der Lernrate
gamma = 0.90 # discount-Faktor
state=0#initialer Zustand

for iteration in range(10000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = np.max(Q_values[next_state])
```

```

alpha = alpha0 / (1 + iteration * decay)

Q_values[state, action] *= 1 - alpha

Q_values[state, action] += alpha * (reward + gamma * next_value)

state = next_state

```

Dieser Algorithmus wird zu den optimalen Q-Werten konvergieren, aber es werden sehr viele Iterationen und möglicherweise ziemlich viel Tuning der Hyperparameter notwendig sein. Wie Sie in [Abbildung 18-9](#) sehen, konvergiert der Q-Wert-Iterationsalgorithmus (links) sehr schnell in weniger als 20 Iterationen, während der Q-Learning-Algorithmus (rechts) über 8.000 Iterationen zum Konvergieren braucht. Offensichtlich wird das Finden der optimalen Policy viel schwerer, wenn die Übergangswahrscheinlichkeiten nicht bekannt sind!

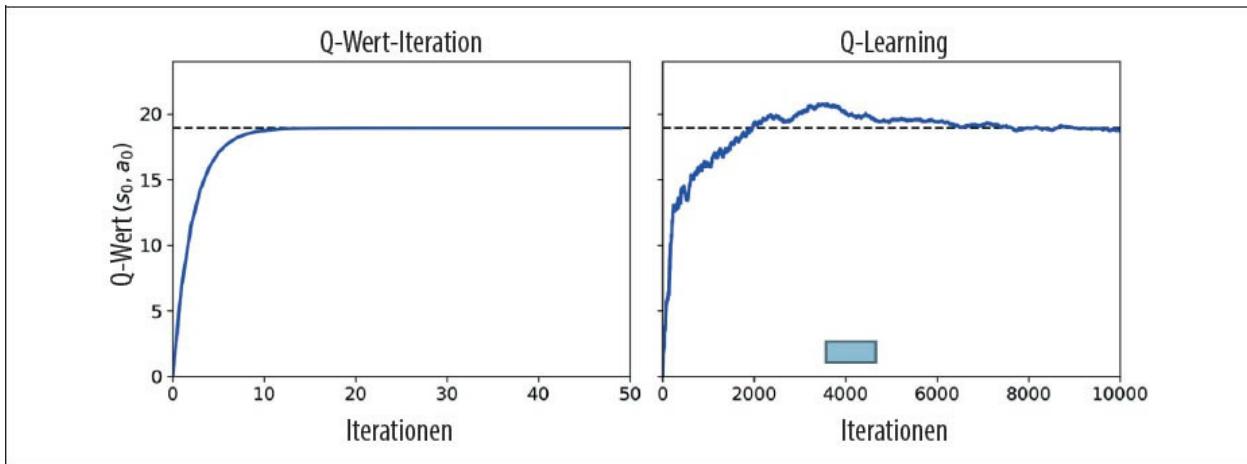


Abbildung 18-9: Der Q-Wert-Iterationsalgorithmus (links) und der Q-Learning-Algorithmus (rechts)

Der Q-Learning-Algorithmus wird als *Off-Policy*-Algorithmus bezeichnet, weil die zu trainierende Policy nicht notwendigerweise die ist, die ausgeführt wird: Im vorherigen Codebeispiel ist die ausgeführte Policy (die Erforschungspolicy) komplett zufallsbasiert, während die zu trainierende Policy immer die Aktion mit den höchsten Q-Werten wählt. Umgekehrt ist der Policy-Gradienten-Algorithmus ein *On-Policy*-Algorithmus: Er erforscht die Welt mit der Policy, die trainiert wird. Es mag ein wenig überraschen, dass Q-Learning die optimale Policy alleine dadurch erlernen kann, dass er dabei zusieht, wie ein Agent zufällig agiert (stellen Sie sich vor, Sie lernen das Golfspielen bei einem betrunkenen Affen als Lehrer). Geht das vielleicht noch besser?

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 für das Besuchen unbekannter Regionen des MDP übrig ist. 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 18-6](#) zeigt.

Formel 18-6: Q-Learning mit einer Erkundungsfunktion

$$Q(s, a) \leftarrow r + \gamma \cdot \max_{a'} f(Q(s', a'), N(s', a'))$$

- $N(s', a')$ zählt, wie oft Action a' im Zustand s' ausgewählt wurde.
- $f(Q, N)$ ist eine *Erkundungsfunktion* wie $f(Q, N) = Q + \kappa/(1 + N)$, wobei der Hyperparameter κ die Neugierde festlegt, also wie stark der Agent vom Unbekannten angezogen wird.

Approximativer Q-Learning und Deep-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 (siehe [Abbildung 18-1](#)). Es gibt etwa 150 Punkte, die Ms. Pac-Man essen kann. Jeder davon kann vorhanden oder bereits aufgegessen sein. Die Anzahl möglicher Zustände übersteigt $2^{150} \approx 10^{45}$. Addieren Sie alle möglichen Positions kombinationen für alle Geister und Ms. Pac-Man, wird die Zahl möglicher Zustände größer als die Anzahl der Atome unserer Erde, es gibt also keine Möglichkeit, eine Schätzung für jeden einzelnen Q-Wert zu speichern.

Die Lösung ist daher, die Q-Werte eines Paares (s, a) 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 *approximativer 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 (<https://homl.info/dqn>) konnte jedoch im Jahr 2013 demonstrieren, dass Deep Neural Networks 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 approximativer 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 18-7](#).

Formel 18-7: Ziel-Q-Wert

$$Q_{\text{target}}(s, a) = r + \gamma \cdot \max_{a'} Q_{\theta}(s', a')$$

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 (oder den Huber-Verlust, um die Empfindlichkeit des Algorithmus für große Fehler zu verringern). Und das ist für den grundlegenden Deep-Q-Learning-Algorithmus auch schon alles! Schauen wir, wie wir ihn implementieren können, um das CartPole-Problem zu lösen.

Deep-Q-Learning implementieren

Als Erstes benötigen wir ein Deep-Q-Netz. Theoretisch brauchen Sie ein neuronales Netz, das ein Zustand-Aktion-Paar übernimmt und einen genäherten Q-Wert ausgibt, aber in der Praxis ist es viel effizienter, ein neuronales Netz zu verwenden, das einen Zustand übernimmt und einen genäherten Q-Wert für jede mögliche Aktion liefert. Um die CartPole-Umgebung zu lösen, brauchen wir kein allzu kompliziertes neuronales Netz – es reichen ein paar verborgene Schichten:

```
env = gym.make("CartPole-v0")

input_shape = [4] # == env.observation_space.shape

n_outputs = 2 # == env.action_space.n

model = keras.models.Sequential([
    keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    keras.layers.Dense(32, activation="elu"),
    keras.layers.Dense(n_outputs)
])
```

Um mit diesem DQN eine Aktion auszuwählen, entscheiden wir uns für diejenige mit dem größten vorhergesagten Q-Wert. Damit wir sicherstellen, dass der Agent die Umgebung erkundet, werden wir eine ε -greedy-Policy nutzen (wir wählen also zufällig eine Aktion mit der

Wahrscheinlichkeit ε):

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

Statt das DQN nur mit den letzten Erfahrungen zu trainieren, legen wir alle Erfahrungen in einem *Replay Buffer* (oder *Replay Memory*) ab und sampeln daraus bei jeder Trainingsiteration einen zufälligen Trainingsbatch. Das hilft dabei, die Korrelationen zwischen den Erfahrungen in einem Trainingsbatch zu verringern, was das Training massiv besser macht. Dazu brauchen wir nur eine Deque-Liste:

```
from collections import deque

replay_buffer = deque(maxlen=2000)
```

Eine *Deque* ist eine verkettete Liste, bei der jedes Element auf das nächste und auf das vorherige Element verweist. Das Einfügen und Löschen von Elementen ist dadurch sehr schnell, aber je länger die Liste wird, desto langsamer ist der freie Zugriff auf ein Element. Benötigen Sie einen sehr großen Replay Buffer, verwenden Sie einen Circular Buffer (siehe den Abschnitt »Deque vs Rotating List« im Notebook).

Jede Erfahrung wird aus fünf Elementen bestehen: einem Zustand, der vom Agenten umgesetzten Aktion, der daraus entstandenen Belohnung, dem nächsten erreichten Zustand und schließlich einem booleschen Wert, der angibt, ob die Episode an diesem Punkt beendet wurde (*done*). Wir werden eine kleine Funktion brauchen, um einen zufälligen Batch mit Erfahrungen aus dem Replay Buffer zu sampeln. Diese wird fünf NumPy-Arrays zurückgeben, die den fünf Erfahrungselementen entsprechen:

```
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(5)]
```

```
    return states, actions, rewards, next_states, dones
```

Erstellen wir auch noch eine Funktion, die einen einzelnen Schritt ausführt, dabei die ϵ -greedy-Policy nutzt und die sich daraus ergebende Erfahrung im Replay Buffer ablegt:

```
def play_one_step(env, state, epsilon):  
    action = epsilon_greedy_policy(state, epsilon)  
    next_state, reward, done, info = env.step(action)  
    replay_buffer.append((state, action, reward, next_state, done))  
    return next_state, reward, done, info
```

Bauen wir schließlich eine letzte Funktion, die einen Batch mit Erfahrungen aus dem Replay Buffer sampelt und das DQN durch einen einzelnen Gradientenschritt für diesen Batch trainiert:

```
batch_size = 32  
  
discount_factor = 0.95  
  
optimizer = keras.optimizers.Adam(lr=1e-3)  
loss_fn = keras.losses.mean_squared_error  
  
def training_step(batch_size):  
    experiences = sample_experiences(batch_size)  
    states, actions, rewards, next_states, dones = experiences  
    next_Q_values = model.predict(next_states)  
    max_next_Q_values = np.max(next_Q_values, axis=1)  
    target_Q_values = (rewards +  
                       (1 - dones) * discount_factor * max_next_Q_values)  
    target_Q_values = target_Q_values.reshape(-1, 1)  
    mask = tf.one_hot(actions, n_outputs)  
    with tf.GradientTape() as tape:  
        all_Q_values = model(states)  
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)  
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
```

```

grads = tape.gradient(loss, model.trainable_variables)
optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

Schauen wir uns diesen Code genauer an:

- Zuerst definieren wir ein paar Hyperparameter und erstellen den Optimierer und die Verlustfunktion.
- Dann erstellen wir die Funktion `training_step()`. Sie sammelt zuerst einen Batch mit Erfahrungen, dann nutzt sie das DQN, um den Q-Wert für jede mögliche Aktion in jedem nächsten Zustand der Erfahrung vorherzusagen. Da wir davon ausgehen, dass der Agent optimal spielen wird, merken wir uns nur den maximalen Q-Wert für jeden nächsten Zustand. Als Nächstes nutzen wir [Formel 18-7](#), um den Ziel-Q-Wert für jedes Zustand-Aktion-Paar einer Erfahrung zu berechnen.
- Nun wollen wir das DQN nutzen, um den Q-Wert für jedes erfahrene Zustand-Aktion-Paar zu bestimmen. Aber das DQN gibt auch die Q-Werte für andere mögliche Aktionen aus, nicht nur für die vom Agenten gewählte. Also müssen wir alle Q-Werte ausblenden, die wir nicht benötigen. Die Funktion `tf.one_hot()` erleichtert das Umwandeln eines Arrays mit Aktionsindizes in solch eine Maske. Enthalten beispielsweise die ersten drei Erfahrungen die Aktionen 1, 1, 0, wird die Maske mit `[[0, 1], [0, 1], [1, 0], ...]` beginnen. Wir können dann die Ausgabe des DQN mit dieser Maske multiplizieren, was alle Q-Werte auf null setzen wird, die wir nicht haben wollen. Nun summieren wir über die erste Achse auf, um alle Nullen loszuwerden, und behalten nur die Q-Werte der erfahrenen Zustand-Aktion-Paare. Damit erhalten wir den Tensor `Q_values` mit einem vorhergesagten Q-Wert für jede Erfahrung im Batch.
- Jetzt berechnen wir den Verlust: Das ist der mittlere quadratische Fehler zwischen den vorhergesagten und den Ziel-Q-Werten für die erfahrenen Zustand-Aktion-Paare.
- Schließlich führen wir einen Gradientenschritt durch, um den Verlust in Bezug auf die trainierbaren Variablen des Modells zu minimieren.

Das war der schwerste Teil. Das Trainieren des Modells ist jetzt nicht mehr kompliziert:

```

for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    if episode > 50:
        training_step(batch_size)

```

Wir lassen 600 Episoden mit jeweils maximal 200 Schritten laufen. Bei jedem Schritt berechnen wir erst den ϵ -Wert für die ϵ -greedy-Policy: Er wird linear in etwas weniger als 500 Episoden von 1 bis auf 0,01 heruntergehen. Dann rufen wir die Funktion `play_one_step()` auf, die mithilfe der ϵ -greedy-Policy eine Aktion auswählt, sie ausführt und die Erfahrung im Replay Buffer ablegt. Ist die Episode abgeschlossen, verlassen wir die Schleife. Sind wir schließlich über die 50. Episode hinaus, rufen wir die Funktion `training_step()` auf, um das Modell mit einem Batch zu trainieren, der aus dem Replay Buffer gesampelt wurde. Wir spielen 50 Episoden ohne Training, um dem Replay Buffer etwas Zeit zu geben, gefüllt zu werden (wenn wir nicht lange genug warten, wird es im Buffer nicht genug Diversität geben). Und das ist es schon – wir haben gerade den Deep-Q-Learning-Algorithmus implementiert!

In Abbildung 18-10 sehen Sie die gesamten Belohnungen, die der Agent bei jeder Episode erhält.

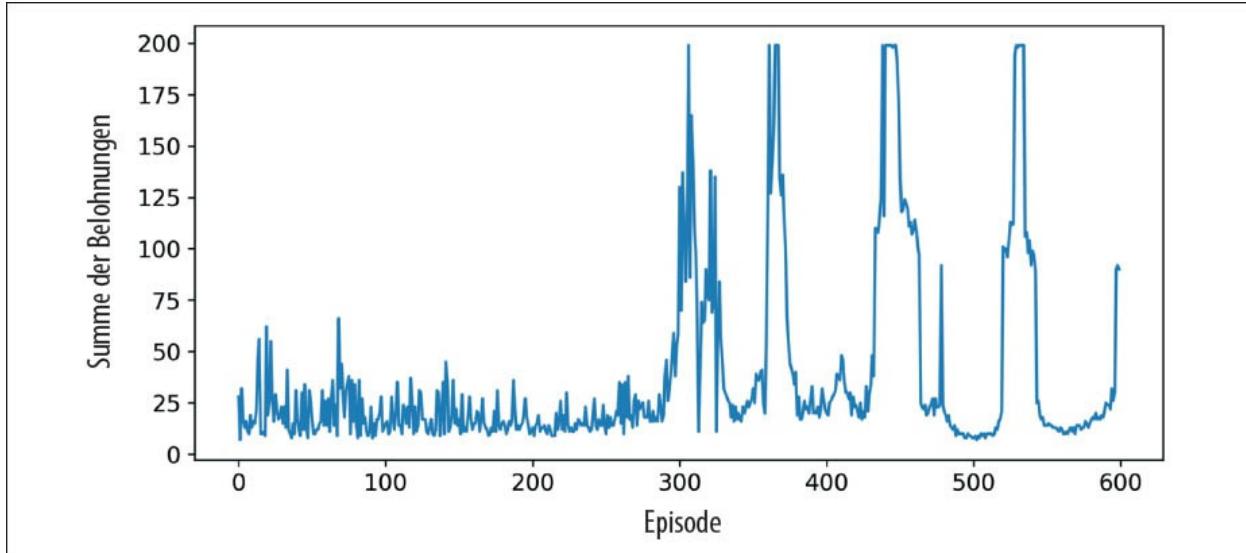


Abbildung 18-10: Lernkurve des Deep-Q-Learning-Algorithmus

Wie Sie sehen, hat der Algorithmus fast 300 Episoden lang kaum Fortschritte gemacht (was teilweise daran liegt, dass ϵ zu Beginn sehr hoch war), dann schoss die Performance plötzlich bis auf 200 hoch (was in dieser Umgebung das maximal Mögliche ist). Das ist super: Der Algorithmus hat gut funktioniert und lief sogar viel schneller als der Policy-Gradient-Algorithmus! Aber halt ... nur ein paar Episoden später hat er alles vergessen, was er wusste, und seine Performance fiel auf unter 25! Das nennt sich *katastrophales Vergessen* und ist eines der großen Probleme, mit dem so gut wie alle RL-Algorithmen zu kämpfen haben: Während der Agent seine Umgebung erforscht, passt er seine Policy an, aber was er in einem Teil der Umgebung lernt, kann das kaputt machen, was er in anderen Teilen zuvor gelernt hat. Die Erfahrungen sind recht eng miteinander korreliert, und die Lernumgebung ändert sich fortlaufend – das ist für die Gradientenmethode nicht ideal! Vergrößern Sie den Replay Buffer, wird der Algorithmus weniger Probleme damit haben. Auch ein Verringern der Lernrate kann helfen. Aber es lässt sich nicht leugnen: Reinforcement Learning ist schwer. Das Training ist oft instabil, und Sie müssen möglicherweise viele Hyperparameterwerte und Zufalls-Seeds ausprobieren,

bevor Sie eine Kombination finden, die gut funktioniert. Ändern Sie beispielsweise im vorherigen Beispiel die Anzahl der Neuronen pro Schicht von 32 auf 30 oder 34, wird die Performance nie über 100 gehen (das DQN ist eventuell mit nur einer verborgenen Schicht stabiler als mit zweien).

 Reinforcement Learning ist knifflig, vor allem aufgrund der Instabilitäten beim Training und der starken Empfindlichkeit bei der Wahl der Hyperparameter und der Zufalls-Seeds.¹³ Wie es der Forscher Andrej Karpathy ausdrückt: »[Überwachtes Lernen] will arbeiten. [...] RL muss zur Arbeit gezwungen werden.« Sie werden Zeit, Geduld, Beharrlichkeit und vielleicht auch ein bisschen Glück brauchen. Das ist einer der Hauptgründe dafür, dass RL nicht so verbreitet ist wie normales Deep Learning (zum Beispiel Convolutional Networks). Aber es gibt ein paar echte Anwendungsfälle über AlphaGo und Atari-Spiele hinaus: So nutzt beispielsweise Google RL zum Optimieren seiner Kosten für die Datacenters, und es kommt in manchen Robotikanwendungen zum Einsatz, zum Tunen von Hyperparametern und in Empfehlungssystemen.

Sie fragen sich vielleicht, warum wir den Verlust nicht grafisch ausgegeben haben. Es stellt sich heraus, dass der Verlust ein schlechter Indikator für die Leistung eines Modells ist. Er kann heruntergehen, aber trotzdem kann sich der Agent schlecht entscheiden (das kann beispielsweise geschehen, wenn der Agent in einem kleinen Bereich der Umgebung festhängt und das DQN diese Region langsam overfittet). Umgekehrt kann der Verlust nach oben gehen, obwohl der Agent gute Entscheidungen trifft (wenn das DQN zum Beispiel die Q-Werte zu niedrig schätzt und damit beginnt, seine Vorhersagen zu erhöhen, wird sich der Agent vermutlich besser verhalten und mehr Belohnung bekommen, aber der Verlust steigt eventuell trotzdem, weil das DQN auch die Ziele setzt, die ebenfalls höher sein können).

Der einfache Deep-Q-Learning-Algorithmus, den wir bisher genutzt haben, wäre zu instabil, um Atari-Spiele zu erlernen. Wie hat DeepMind das dann gemacht? Nun, sie haben am Algorithmus gedreht!

Deep-Q-Learning-Varianten

Schauen wir uns ein paar Varianten des Deep-Q-Learning-Algorithmus an, die das Training stabilisieren und beschleunigen können.

Feste Q-Wert-Ziele

Im einfachen Deep-Q-Learning-Algorithmus wurde das Modell sowohl für die Vorhersagen wie auch für das Setzen der eigenen Ziele eingesetzt. Das kann dann zu einer Situation wie bei einem Hund führen, der seinen eigenen Schwanz jagt. Diese Feedback-Schleife kann das Netz instabil machen: Sie kann auseinanderlaufen, oszillieren, einfrieren und so weiter. Um dieses Problem zu lösen, haben die Deep-Mind-Forscher in ihrem Artikel aus dem Jahr 2013 statt einem nun zwei DQNs verwendet: Das erste ist das *Onlinemodell*, das bei jedem Schritt lernt und genutzt wird, um den Agenten zu bewegen, während das andere das *Zielmodell* ist, das nur zum Definieren der Ziele dient. Letzteres ist einfach ein Klon des Onlinemodells:

```

target = keras.models.clone_model(model)

target.set_weights(model.get_weights())

```

In der Funktion `training_step()` müssen wir nur eine Zeile ändern, sodass zum Berechnen der Q-Werte der nächsten Zustände das Zielmodell statt des Onlinemodells genutzt wird:

```
next_Q_values = target.predict(next_states)
```

Und schließlich müssen wir in der Trainingsschleife regelmäßig die Gewichte aus dem Onlinemodell in das Zielmodell kopieren (zum Beispiel alle 50 Episoden):

```

if episode % 50 == 0:

    target.set_weights(model.get_weights())

```

Da das Zielmodell seltener als das Onlinemodell aktualisiert wird, sind die Q-Wert-Ziele stabiler, die weiter oben besprochene Feedback-Schleife ist beruhigt, und die Effekte sind weniger stark. Dieser Ansatz war einer der wichtigsten Beiträge der DeepMind-Forscher in ihrem Artikel aus dem Jahr 2013, denn damit konnten Agenten Atari-Spiele nur über die Pixel erlernen. Um das Training zu stabilisieren, verwendeten sie eine winzige Lernrate von 0,00025, aktualisierten das Zielmodell nur alle 10.000 Schritte (statt wie im obigen Codebeispiel alle 50 Schritte) und setzten einen sehr großen Replay Buffer mit 1 Million Erfahrungen ein. `epsilon` wurde nur sehr langsam verringert – von 1 auf 0,1 in 1 Million Schritte, zudem wurde der Algorithmus 50 Millionen Schritte laufen gelassen.

Weiter unten werden wir die TF-Agents-Bibliothek nutzen, um einen DQN-Agenten darin zu trainieren, mit diesen Hyperparametern *Breakout* zu spielen, aber zuvor wollen wir uns erst mal eine andere DQN-Variante anschauen, die die Latte noch höher gelegt hat.

Double DQN

In einem Artikel (<https://hml.info/doubledqn>)¹⁴ aus dem Jahr 2015 haben Deep-Mind-Forscher ihren DQN-Algorithmus modifiziert, seine Leistung verbessert und das Training ein wenig stabilisiert. Sie nannten diese Variante *Double DQN*. Die Aktualisierung entstammte der Beobachtung, dass das Zielnetzwerk anfällig dafür ist, Q-Werte zu überschätzen. Stellen Sie sich vor, alle Aktionen wären gleich gut: Die vom Zielmodell geschätzten Q-Werte sollten dann identisch sein, aber da es sich um Näherungen handelt, sind einige vielleicht rein zufällig etwas größer als andere. Das Zielmodell wählt immer den größten Q-Wert, der etwas größer als der mittlere Q-Wert ist, womit der wahre Q-Wert sehr wahrscheinlich überschätzt wird (das ist ein bisschen so, als würde man die Höhe der höchsten zufälligen Welle messen, wenn man die Tiefe eines Pools erfahren möchte). Um das zu beheben, schlügen die Autoren vor, beim Wählen der besten Aktion für die nächsten Zustände das Onlinemodell statt des Zielmodells zu nutzen und mit dem Zielmodell nur die Q-Werte für diese besten Aktionen zu schätzen. Hier die aktualisierte Funktion `training_step()`:

```

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
    target_Q_values = (rewards +
        (1 - dones) * discount_factor * next_best_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    [...] # der Rest ist wie zuvor

```

Nur ein paar Monate später wurde eine weitere Verbesserung des DQN-Algorithmus vorgeschlagen.

Priorisiertes Experience Replay

Statt Erfahrungen *einheitlich* aus dem Replay Buffer zu sampeln, könnten wir wichtige Erfahrungen doch häufiger auswählen. Diese Idee nennt sich *Importance Sampling* (IS) oder *priorisiertes Experience Replay* (PER), und sie wurde in einem Artikel (<https://homl.info/prioreplay>)¹⁵ ebenfalls aus dem Jahr 2015 vorgestellt – wieder von DeepMind-Forschern.

Genauer gesagt, werden Erfahrungen als »wichtig« betrachtet, wenn sie wahrscheinlicher zu schnellerem Lernfortschritt führen. Aber wie können wir das abschätzen? Ein vernünftiger Ansatz ist das Messen des Ausmaßes des TD-Fehlers $\delta = r + \gamma \cdot V(s') - V(s)$. Ein großer TD-Fehler zeigt an, dass ein Übergang (s, r, s') sehr überraschend ist und es sich vermutlich lohnt, daraus zu lernen.¹⁶ Wird eine Erfahrung im Replay Buffer aufgezeichnet, wird ihre Priorität auf einen sehr großen Wert gesetzt, um sicherzustellen, dass sie mindestens einmal gesampelt wird. Aber nachdem sie einmal ausgewählt wurde (und bei jedem weiteren Sampeln), wird der TD-Fehler δ berechnet und die Priorität dieser Erfahrung auf $p = |\delta|^\zeta$ gesetzt (plus einer kleinen Konstanten, um sicherzustellen, dass jede Erfahrung eine Sample-Wahrscheinlichkeit ungleich null hat). Die Wahrscheinlichkeit P , eine Erfahrung mit der Priorität p zu sampeln, ist proportional zu p^ζ mit ζ als Hyperparameter, der steuert, wie gierig das Importance Sampling sein soll: Ist $\zeta = 0$, erhalten wir ein gleichverteiltes Sampling, ist $\zeta = 1$, ist das Importance Sampling voll aktiv. Im Artikel haben die Autoren $\zeta = 0,6$ gewählt, aber der optimale Wert wird von der Aufgabe abhängen.

Es gibt allerdings einen Nachteil: Da die Samples einen Bias hin zu wichtigen Erfahrungen haben, müssen wir diesen beim Training kompensieren, indem wir die Erfahrungen nach ihrer

Wichtigkeit heruntergewichten, denn ansonsten wird das Modell die wichtigen Erfahrungen overfitten. Wir wollen durchaus wichtige Erfahrungen häufiger sampeln, aber wir müssen ihnen dann beim Training ein niedrigeres Gewicht geben. Dazu definieren wir das Trainingsgewicht jeder Erfahrung als $w = (n P)^{-\beta}$ mit n als Anzahl der Erfahrungen im Replay Buffer und β als Hyperparameter, der steuert, wie stark wir den Bias durch Importance Sampling kompensieren wollen (0 bedeutet gar nicht, 1 vollständig). Im Artikel haben die Autoren zu Beginn des Trainings $\beta = 0,4$ gewählt und es linear bis zu $\beta = 1$ am Ende des Trainings gesteigert. Auch hier wird der optimale Wert von der Aufgabe abhängen, aber wenn Sie den einen Wert erhöhen, werden Sie im Allgemeinen auch den anderen erhöhen wollen.

Schauen wir uns jetzt noch eine letzte wichtige Variante des DQN-Algorithmus an.

Dueling DQN

Der *Dueling-DQN*-Algorithmus (DDQN, nicht zu verwechseln mit Double DQN, auch wenn sich beide Techniken leicht kombinieren lassen) wurde in noch einem Artikel (<https://homl.info/ddqn>)¹⁷ aus dem Jahr 2015 von DeepMind-Forschern vorgestellt. Um zu verstehen, wie er funktioniert, müssen wir uns zuerst darüber klar sein, dass der Q-Wert eines Zustand-Aktion-Paars (s, a) als $Q(s, a) = V(s) + A(s, a)$ ausgedrückt werden kann, wobei $V(s)$ der Wert des Zustands s und $A(s, a)$ der Vorteil (Advantage) ist, Aktion a im Zustand s zu wählen – verglichen mit allen anderen möglichen Aktionen in diesem Zustand. Zudem ist der Wert eines Zustands gleich dem Q-Wert der besten Aktion a^* für diesen Zustand (da wir davon ausgehen, dass die optimale Policy die beste Aktion wählen wird), daher ist $V(s) = Q(s, a^*)$, was nahelegt, dass $A(s, a^*) = 0$. In einem Dueling DQN schätzt das Modell die Werte für den Zustand und den Vorteil jeder möglichen Aktion. Da die beste Aktion einen Vorteil von 0 haben sollte, zieht das Modell den maximal vorhergesagten Vorteil von allen vorhergesagten Vorteilen ab. Hier ein einfaches Dueling-DQN-Modell, das mit der Functional API implementiert wurde:

```
K = keras.backend

input_states = keras.layers.Input(shape=[4])

hidden1 = keras.layers.Dense(32, activation="elu")(input_states)

hidden2 = keras.layers.Dense(32, activation="elu")(hidden1)

state_values = keras.layers.Dense(1)(hidden2)

raw_advantages = keras.layers.Dense(n_outputs)(hidden2)

advantages = raw_advantages - K.max(raw_advantages, axis=1, keepdims=True)

Q_values = state_values + advantages

model = keras.Model(inputs=[input_states], outputs=[Q_values])
```

Der Rest des Algorithmus arbeitet wie zuvor. Sie können auch ein Double Dueling DQN bauen und es mit dem priorisierten Experience Replay kombinieren! Im Allgemeinen lassen sich viele

RL-Techniken kombinieren, wie DeepMind in einem Artikel (<https://hml.info/rainbow>)¹⁸ aus dem Jahr 2017 gezeigt hat. Die Autoren des Artikels haben sechs verschiedene Techniken in einem Agenten namens *Rainbow* kombiniert, der sich deutlich besser als die damals besten geschlagen hat.

Leider kann das Implementieren all dieser Techniken, ihr Debuggen, Anpassen und natürlich das Trainieren der Modelle viel Arbeit sein. Statt also das Rad neu zu erfinden, ist es oft besser, skalierbare und gut getestete Bibliotheken wie TF-Agents zu verwenden.

Die TF-Agents-Bibliothek

Die TF-Agents-Bibliothek (<https://github.com/tensorflow/agents>) ist eine Bibliothek zum Reinforcement Learning, die auf TensorFlow basiert, bei Google entwickelt und 2018 als Open Source veröffentlicht wurde. Wie OpenAI Gym stellt sie viele fertige Umgebungen bereit (unter anderem Wrapper für alle OpenAI-Gym-Umgebungen), zudem unterstützt sie die PyBullet-Bibliothek (zur 3-D-Physik-Simulation), die DM-Control-Bibliothek von DeepMind (basierend auf der Physik-Engine von MuJoCo) und die ML-Agents-Bibliothek von Unity (die viele 3-D-Umgebungen simuliert). Außerdem implementiert sie viele RL-Algorithmen, unter anderem REINFORCE, DQN und DDQN, sowie verschiedene RL-Komponenten wie effiziente Replay Buffer und Metriken.

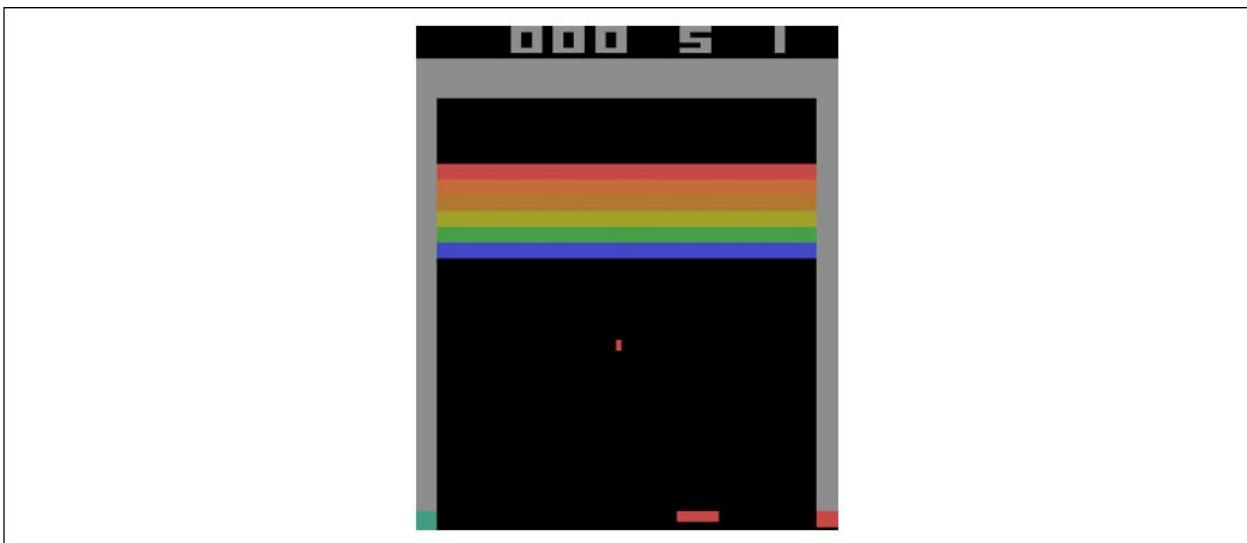


Abbildung 18-11: Das berühmte Breakout

Sie ist schnell, skalierbar, einfach zu verwenden und anpassbar: Sie können Ihre eigenen Umgebungen und neuronalen Netze erstellen und so gut wie jede Komponente anpassen. In diesem Abschnitt werden wir TF-Agents nutzen, um einen Agenten zum Spielen von *Breakout* zu trainieren. Dabei handelt es sich um ein berühmtes Atari-Spiel (siehe Abbildung 18-11). Verwendet wird zum Training der DQN-Algorithmus (wenn Sie möchten, können Sie problemlos zu einem anderen Algorithmus wechseln).

TF-Agents installieren

Beginnen wir mit der Installation von TF-Agents. Das können Sie über pip erledigen (wie immer müssen Sie beim Verwenden einer virtuellen Umgebung sicherstellen, dass diese auch aktiviert ist – wenn nicht, müssen Sie die Option `--user` nutzen oder Administratorrechte haben):

```
$ python3 -m pip install --upgrade tf-agents
```

Aktuell ist TF-Agents noch ziemlich neu und erfährt jeden Tag Verbesserungen, daher kann

sich die API ein wenig verändert haben, wenn Sie diese Zeilen lesen – aber im Großen und Ganzen sollte es gleich bleiben, wie auch ein Großteil des Codes. Wenn etwas nicht mehr funktioniert, werde ich das Jupyter-Notebook entsprechend anpassen – werfen Sie also einen Blick darauf.

Als Nächstes erstellen wir eine TF-Agents-Umgebung, die einfach die Breakout-Umgebung von OpenAI Gym verpackt. Dazu müssen Sie zuerst die Atari-Dependencies von OpenAI Gym installieren:

```
$ python3 -m pip install --upgrade 'gym[atari]'
```

Neben weiteren Bibliotheken wird dieser Befehl `atari-py` installieren, bei dem es sich um eine Python-Schnittstelle zum Arcade Learning Environment (ALE) handelt – einem Framework, das auf dem Atari-2600-Emulator Stella aufbaut.

TF-Agents-Umgebungen

Hat alles funktioniert, sollten Sie TF-Agents importieren und eine Breakout-Umgebung erstellen können:

```
>>> from tf_agents.environments import suite_gym  
>>> env = suite_gym.load("Breakout-v4")  
>>> env  
  
<tf_agents.environments.wrappers.TimeLimit at 0x10c523c18>
```

Das ist nur ein Wrapper um eine OpenAI-Gym-Umgebung, auf die Sie über das Attribut `gym` zugreifen können:

```
>>> env.gym  
  
<gym.envs.atari.atari_env.AtariEnv at 0x24dcab940>
```

TF-Agents-Umgebungen ähneln den OpenAI-Gym-Umgebungen stark, aber es gibt auch ein paar Unterschiede. So gibt die Methode `reset()` keine Beobachtung zurück – stattdessen liefert sie ein `TimeStep`-Objekt, das die Beobachtung zusammen mit ein paar zusätzlichen Informationen verpackt:

```
>>> env.reset()  
  
TimeStep(step_type=array(0, dtype=int32),  
         reward=array(0., dtype=float32),  
         discount=array(1., dtype=float32),  
         observation=array([[[0., 0., 0.], [0., 0., 0.], ...]]], dtype=float32))
```

Die Methode `step()` gibt ebenfalls ein `TimeStep`-Objekt zurück:

```
>>> env.step(1) # Schuss  
TimeStep(step_type=array(1, dtype=int32),  
         reward=array(0., dtype=float32),  
         discount=array(1., dtype=float32),  
         observation=array([[0., 0., 0.], [0., 0., 0.], ...]], dtype=float32))
```

Die Attribute `reward` und `observation` sind selbsterklärend und entsprechen denen für OpenAI Gym (nur dass `reward` als NumPy-Array repräsentiert wird). Das Attribut `step_type` ist für den ersten Schritt in der Episode gleich 0, für die Schritte in der Episode gleich 1 und für den letzten Schritt gleich 2. Sie können die Methode `is_last()` des `TimeStep` aufrufen, um zu prüfen, ob es sich um den letzten Schritt handelt. Und schließlich gibt das Attribut `discount` den Discount-Faktor an, der bei diesem Zeitschritt genutzt wird. In diesem Beispiel ist er 1, daher wird es keinen Discount geben. Sie können ihn beim Laden der Umgebung über das Setzen des Parameters `discount` festlegen.



Sie können auf den aktuellen Zeitschritt der Umgebung jederzeit über deren Methode `current_time_step()` zugreifen.

Umgebungsspezifikationen

Praktischerweise stellt eine TF-Agents-Umgebung die Spezifikationen der Beobachtungen, Aktionen und Zeitschritte bereit, unter anderem ihre Formen, Datentypen und -namen, aber auch die minimalen und maximalen Werte:

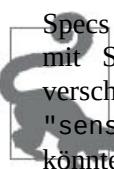
```
>>> env.observation_spec()  
  
BoundedArraySpec(shape=(210, 160, 3), dtype=dtype('float32'), name=None,  
                  minimum=[[0. 0. 0.], [0. 0. 0.], ...]],  
                  maximum=[[255., 255., 255.], [255., 255., 255.], ...])  
  
>>> env.action_spec()  
  
BoundedArraySpec(shape=(), dtype=dtype('int64'), name=None,  
                  minimum=0, maximum=3)  
  
>>> env.time_step_spec()  
  
TimeStep(step_type=ArraySpec(shape=(), dtype=dtype('int32'), name='step_type'),  
         reward=ArraySpec(shape=(), dtype=dtype('float32'), name='reward'),
```

```
discount=BoundedArraySpec(shape=(), ..., minimum=0.0, maximum=1.0),  
observation=BoundedArraySpec(shape=(210, 160, 3), ...))
```

Wie Sie sehen, handelt es sich bei den Beobachtungen einfach um Screenshots des Atari-Bildschirms, dargestellt als NumPy-Array der Form [210, 160, 3]. Um eine Umgebung zu rendern, können Sie `env.render(mode="human")` aufrufen, und wenn Sie das Bild in Form eines NumPy-Arrays haben wollen, nutzen Sie einfach `env.render(mode="rgb_array")` (anders als in OpenAI Gym ist das der Standardmodus).

Es gibt vier mögliche Aktionen. Die Atari-Umgebung von Gym besitzt eine eigene Methode, die Sie aufrufen können, um zu erfahren, womit jede Aktion korrespondiert:

```
>>> env.gym.get_action_meanings()  
['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```

 Specs können Instanzen einer Spezifikationsklasse, einer verschachtelten Liste oder Dictionaries mit Specs sein. Ist die Spezifikation verschachtelt, muss das spezifizierte Objekt zur verschachtelten Struktur der Spezifikation passen. Ist die Beobachtungs-Spec beispielsweise `"sensors": ArraySpec(shape=[2]), "camera": ArraySpec(shape=[100, 100])}`, könnte eine gültige Beobachtung `{"sensors": np.array([1.5, 3.5]), "camera": np.array(...)}` sein. Das Paket `tf.nest` stellt Tools zum Umgang mit solchen verschachtelten Strukturen bereit.

Die Beobachtungen sind ziemlich groß, daher werden wir sie herunterrechnen und in Graustufen umwandeln. Damit wird das Training beschleunigt und weniger RAM benötigt. Hierfür können wir einen *Umgebungswrapper* einsetzen.

Umgebungswrapper und Atari-Vorverarbeitung

TF-Agents stellt diverse Umgebungswrapper im Paket `tf_agents.environments.wrappers` bereit. Wie der Name schon andeutet, wird damit eine Umgebung verpackt, und jeder Aufruf wird dorthin weitergeleitet, aber es wird noch zusätzliche Funktionalität ergänzt. Dies sind ein paar der verfügbaren Wrapper:

ActionClipWrapper

Beschneidet die Aktionen auf die Aktions-Spec.

ActionDiscretizeWrapper

Quantisiert einen kontinuierlichen Aktionsraum auf einen diskreten Aktionsraum. Ist beispielsweise der ursprüngliche Aktionsraum der kontinuierliche Raum von -1,0 bis +1,0 und Sie wollen einen Algorithmus verwenden, der nur diskrete Aktionsräume unterstützt (wie zum Beispiel DQN), können Sie die Umgebung mit `discrete_env = ActionDiscretizeWrapper(env, num_actions=5)` verpacken, um mit `discrete_env` einen diskreten Aktionsraum mit fünf möglichen Aktionen zur Verfügung zu haben: 0, 1, 2, 3, 4. Diese Aktionen entsprechen in der ursprünglichen Umgebung den

Aktionen $-1,0, -0,5, 0,0, 0,5$ und $1,0$.

ActionRepeat

Wiederholt jede Aktion für n Schritte und addiert die Belohnungen auf. Damit lässt sich das Training in vielen Umgebungen deutlich beschleunigen.

RunStats

Zeichnet Umgebungsstatistiken wie zum Beispiel die Anzahl der Schritte und die Anzahl der Episoden auf.

TimeLimit

Unterbricht die Umgebung, wenn sie länger als eine angegebene Anzahl an Schritten läuft.

Um eine verpackte Umgebung zu erzeugen, müssen Sie einen Wrapper erstellen und die zu verpackende Umgebung an den Konstruktor übergeben. Das ist alles! Der folgende Code verpackt beispielsweise unsere Umgebung in einem ActionRepeat-Wrapper, sodass jede Aktion vier Mal wiederholt wird:

```
from tf_agents.environments.wrappers import ActionRepeat

repeating_env = ActionRepeat(env, times=4)
```

OpenAI Gym besitzt einige eigene Umgebungswrapper im Paket `gym.wrappers`. Sie sind aber dazu gedacht, Gym-Umgebungen zu verpacken. Um sie in TF-Agents-Umgebungen zu verwenden, müssen Sie daher erst die Gym-Umgebung mit einem Gym-Wrapper verpacken und die daraus entstandene Umgebung wiederum mit einem TF-Agents-Wrapper umhüllen. Die Funktion `suite_gym.wrap_env()` erledigt das für Sie, sofern Sie ihr eine Gym-Umgebung und eine Liste mit Gym-Wrappern und/oder TF-Agents-Wrappern mitgeben. Alternativ erzeugt die Funktion `suite_gym.load()` die Gym-Umgebung und verpackt sie auch gleich für Sie. Jeder Wrapper wird ohne Argumente erzeugt – wollen Sie also Argumente setzen, müssen Sie ein `lambda` mitgeben. Der folgende Code erzeugt beispielsweise eine Breakout-Umgebung, die in jeder Episode für maximal 10.000 Schritte läuft, und jede Aktion wird vier Mal wiederholt:

```
from gym.wrappers import TimeLimit

limited_repeating_env = suite_gym.load(
    "Breakout-v4",
    gym_env_wrappers=[lambda env: TimeLimit(env, max_episode_steps=10000)],
    env_wrappers=[lambda env: ActionRepeat(env, times=4)])
```

Für Atari-Umgebungen werden in den meisten entsprechenden Artikeln ein paar Vorverarbeitungsschritte angewendet, daher stellt TF-Agents einen praktischen Wrapper `AtariPreprocessing` bereit, der sie implementiert. Dies ist die Liste der unterstützten Schritte:

Graustufen und Downsampling

Beobachtungen werden in Graustufen umgewandelt und heruntergerechnet (standardmäßig auf 84×84 Pixel).

Max-Pooling

Auf die letzten beiden Frames des Spiels wird Max-Pooling mit einem 1×1 -Filter angewendet. Das soll das Flackern entfernen, das in manchen Atari-Spielen aufgrund der begrenzten Anzahl von Sprites auftritt, die der Atari 2600 in jedem Frame anzeigen kann.

Frame Skipping

Der Agent sieht nur jeden n . Frame des Spiels (standardmäßig ist $n = 4$), und seine Aktionen werden für jeden Frame wiederholt, wobei alle Belohnungen aufsummiert werden. Das beschleunigt das Spiel effektiv aus Sicht des Agenten und beschleunigt auch das Training, weil die Belohnungen weniger verzögert werden.

End of Life Lost

In manchen Spielen basieren die Belohnungen nur auf dem Score, daher erhält der Agent keine sofortige Bestrafung für das Verlieren eines Lebens. Eine Lösung ist, das Spiel sofort zu beenden, wenn ein Leben verloren wurde. Ob diese Strategie wirkliche Vorteile bietet, wird noch diskutiert, daher ist sie standardmäßig abgeschaltet.

Da die Standard-Atari-Umgebung schon zufälliges Frame Skipping und Max-Pooling anwendet, müssen wir die reine, nicht skippende Variante namens "Breakout NoFrameskip-v4" laden. Zudem reicht ein einzelner Frame aus dem *Breakout*-Spiel nicht aus, um Richtung und Geschwindigkeit des Balls zu kennen, womit es für den Agenten sehr schwierig ist, das Spiel ordentlich zu spielen (sofern es sich nicht um einen RNN-Agenten handelt, der sich zwischen den Schritten einen internen Status merkt). Eine Möglichkeit, damit umzugehen, ist der Einsatz eines Umgebungswrappers, der Beobachtungen aus mehreren Frames ausgibt, die entlang der Kanaldimension gestapelt sind. Diese Strategie wird mit dem `FrameStack4`-Wrapper implementiert und gibt Stacks mit vier Frames zurück. Erstellen wir also die verpackte Atari-Umgebung!

```
from tf_agents.environments import suite_atari
from tf_agents.environments.atari_preprocessing import AtariPreprocessing
from tf_agents.environments.atari_wrappers import FrameStack4

max_episode_steps = 27000 # <=> 108k ALE frames since 1 step = 4 frames
environment_name = "BreakoutNoFrameskip-v4"

env = suite_atari.load(
    environment_name,
    max_episode_steps=max_episode_steps,
```

```
gym_env_wrappers=[AtariPreprocessing, FrameStack4])
```

Das Ergebnis dieser ganzen Vorverarbeitung sehen Sie in [Abbildung 18-12](#). Die Auflösung ist viel geringer, reicht aber aus, um das Spiel zu spielen. Zudem sind Frames entlang der Kanaldimension gestapelt, wobei rot der Frame vor drei Schritten, grün der vor zwei Schritten und blau der von einem Schritt ist, während der aktuelle Frame in Lila zu sehen ist.¹⁹ Aus dieser einen Beobachtung kann der Agent erkennen, dass sich der Ball in Richtung der linken unteren Ecke bewegt und dass er das Paddle weiterhin nach links bewegen sollte (wie er es schon in den vorherigen Schritten getan hat).

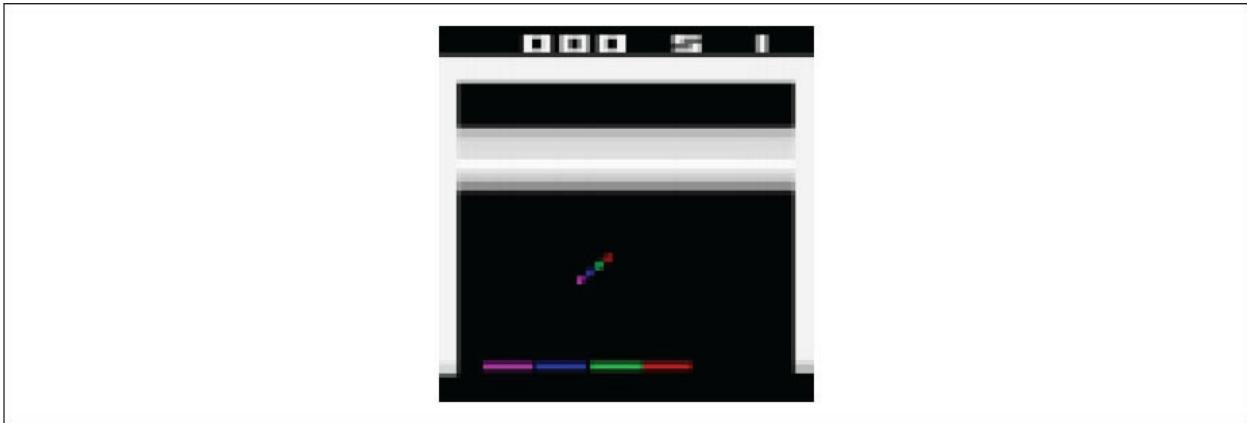


Abbildung 18-12: Vorverarbeitete Breakout-Beobachtung

Schließlich können wir die Umgebung noch in eine TFPyEnvironment verpacken:

```
from tf_agents.environments.tf_py_environment import TFPyEnvironment

tf_env = TFPyEnvironment(env)
```

Damit lässt sich die Umgebung in einem TensorFlow-Graphen verwenden (unter der Motorhaube basiert diese Klasse auf `tf.py_function()`, die es einem Graphen erlaubt, beliebigen Python-Code aufzurufen). Dank der Klasse TFPyEnvironment unterstützt TF-Agents sowohl reine Python-Umgebungen wie auch TensorFlowbasierte Umgebungen. Ganz allgemein unterstützt TF-Agents sowohl Python- als auch TensorFlow-basierte Komponenten (Agenten, Replay Buffer, Metriken und so weiter).

Jetzt haben wir eine nette Breakout-Umgebung mit all der nötigen Vorverarbeitung und Unterstützung durch TensorFlow und müssen nun den DQN-Agenten und die anderen Komponenten zum Trainieren erstellen. Schauen wir uns die Architektur des Systems an, das wir bauen werden.

Trainingsarchitektur

Ein Trainingsprogramm in TF-Agents wird normalerweise in zwei parallel laufen Teile unterteilt, wie Sie in [Abbildung 18-13](#) sehen: Links erkundet ein *Fahrer* die *Umgebung* mithilfe

einer *Collect Policy* zur Wahl von Aktionen und sammelt *Trajektorien* (also Erfahrungen), die er an einen *Beobachter* schickt, der sie in einem *Replay Buffer* sichert; rechts holt sich ein *Agent* Batches mit Trajektorien aus dem Replay Buffer und trainiert *Netze*, die die Collect Policy nutzt. Kurz gesagt, erforscht der linke Teil die Umgebung und sammelt Trajektorien, während der rechte Teil die Collect Policy lernt und aktualisiert.

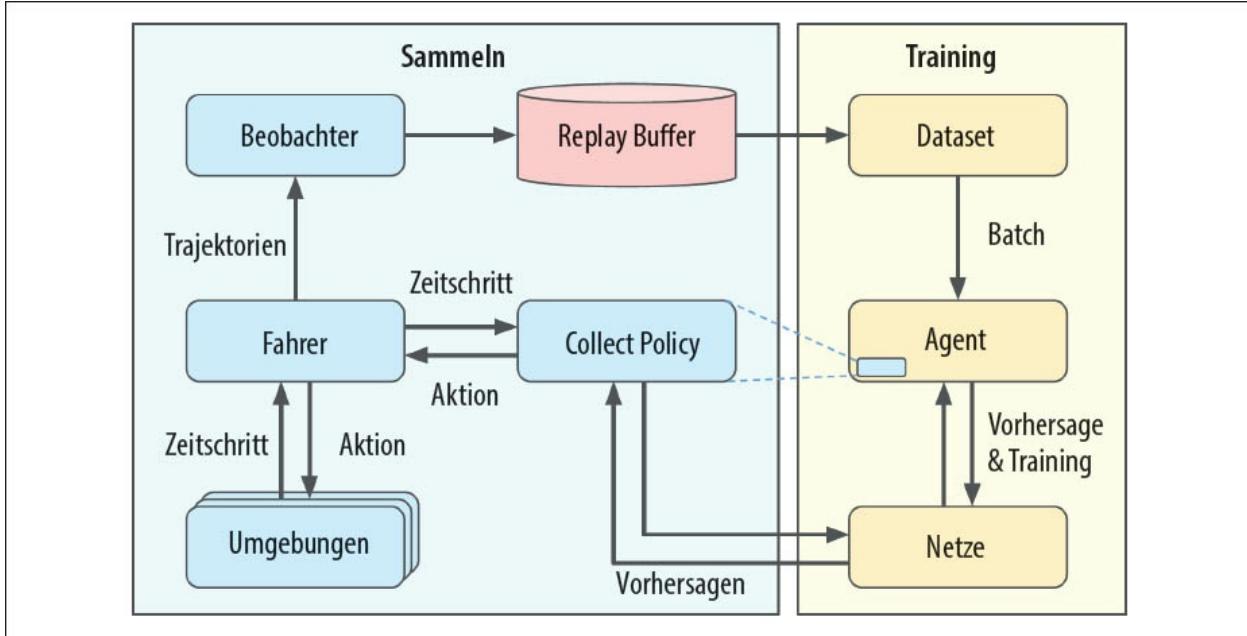
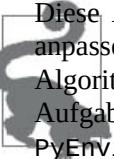


Abbildung 18-13: Eine typische Trainingsarchitektur in TF-Agents

Zur Abbildung stellen sich ein paar Fragen, die ich beantworten möchte:

- Warum gibt es mehrere Umgebungen? Statt eine einzelne Umgebung zu erforschen, wollen Sie im Allgemeinen, dass der Fahrer mehrere Kopien der Umgebung parallel erkundet, weil er so all Ihre CPU-Cores ausnutzen kann, die Training-GPUs beschäftigt bleiben und der Trainingsalgorithmus mit weniger korrelierten Trajektorien versorgt wird.
- Was ist eine *Trajektorie*? Dabei handelt es sich um eine Repräsentation eines *Übergangs* von einem Zeitschritt zum nächsten oder um eine Sequenz aufeinanderfolgender Übergänge von Zeitschritt n zu Zeitschritt $n + t$. Die vom Fahrer gesammelten Trajektorien werden an den Beobachter übergeben, der sie im Replay Buffer ablegt, wo sie später vom Agenten gesampelt und für das Training genutzt werden.
- Warum brauchen wir einen Beobachter? Kann der Fahrer die Trajektorien nicht direkt speichern? Nun, das könnte er, aber damit wäre die Architektur weniger flexibel. Was wäre beispielsweise, wenn Sie keinen Replay Buffer nutzen wollen? Oder wenn Sie die Trajektorien für etwas anderes verwenden möchten (etwa zum Berechnen von Metriken)? Tatsächlich ist ein Beobachter nur eine beliebige Funktion, die eine Trajektorie als Argument übernimmt. Sie können einen Beobachter nutzen, um die Trajektorien in einem Replay Buffer abzulegen, sie in einer TFRecord-Datei zu sichern (siehe [Kapitel 13](#)), Metriken zu berechnen oder etwas ganz anderes damit zu tun. Zudem

können Sie mehrere Beobachter an den Fahrer übergeben, und die Trajektorien werden an alle verteilt.

 Diese Architektur kommt zwar am häufigsten zum Einsatz, aber Sie können sie nach Bedarf anpassen und auch Komponenten daraus ersetzen. Sofern Sie nicht gerade an neuen RL-Algorithmen forschen, werden Sie sogar sehr wahrscheinlich eine eigene Umgebung für Ihre Aufgabe verwenden. Dazu müssen Sie nur eine eigene Klasse erstellen, die von der Klasse PyEnvironment im Paket tf_agents.environments.py_environment erbt und die entsprechenden Methoden wie action_spec(), observation_spec(), _reset() oder _step() überschreibt (im Abschnitt »Creating a Custom TF_Agents Environment« des Notebooks finden Sie ein Beispiel).

Jetzt werden wir alle diese Komponenten erstellen: zuerst das Deep-Q-Netz, dann den DQN-Agenten (der sich um das Erstellen der Collect Policy kümmern wird), den Replay Buffer und den Beobachter, um dort hinein zu schreiben, dann ein paar Trainingsmetriken, den Treiber und schließlich das Dataset. Haben wir alle Komponenten zusammen, werden wir den Replay Buffer mit ein paar initialen Trajektorien bestücken und die Haupt-Trainingsschleife starten. Beginnen wir also mit dem Erstellen des Deep-Q-Netzes.

Deep-Q-Netz erstellen

Die TF-Agents-Bibliothek stellt im Paket tf_agents.networks und dessen Unterpaketen viele Netze zur Verfügung. Wir werden die Klasse tf_agents.networks.q_networks.QNetwork verwenden:

```
from tf_agents.networks.q_network import QNetwork

preprocessing_layer = keras.layers.Lambda(
    lambda obs: tf.cast(obs, np.float32) / 255.)

conv_layer_params=[(32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1)]
fc_layer_params=[512]

q_net = QNetwork(
    tf_env.observation_spec(),
    tf_env.action_spec(),
    preprocessing_layers=preprocessing_layer,
    conv_layer_params=conv_layer_params,
    fc_layer_params=fc_layer_params)
```

Dieses QNetwork übernimmt eine Beobachtung als Eingabe und gibt einen Q-Wert pro Aktion

zurück, daher müssen wir ihm die Spezifikationen der Beobachtungen und der Aktionen mitgeben. Es beginnt mit einer Vorverarbeitungsschicht – einer einfachen Lambda-Schicht, die die Beobachtungen in 32-Bit-Floats castet und sie normalisiert (die Werte liegen dann zwischen 0,0 und 1,0). Die Beobachtungen enthalten vorzeichenlose Bytes, die viermal weniger Platz als 32-Bit-Floats verbrauchen. Daher haben wir sie nicht zuvor in 32-Bit-Floats gecastet – wir wollen RAM im Replay Buffer sparen. Als Nächstes wendet das Netz drei Convolutional Layers an: Der erste hat 32 8×8 -Filter und nutzt eine Schrittweite von 4, der zweite hat 64 4×4 -Filter und eine Schrittweite von 2, und der dritte hat 64 3×3 -Filter mit einer Schrittweite von 1. Schließlich wird noch eine Dense-Schicht mit 512 Einheiten angewendet, gefolgt von einer Dense-Ausgabeschicht mit vier Einheiten – einer pro auszugebendem Q-Wert (also einer pro Aktion). Alle Convolutional Layers und alle Dense-Schichten außer der Ausgabeschicht nutzen standardmäßig die ReLU-Aktivierungsfunktion (Sie können das mit dem Argument `activation_fn` ändern). Die Ausgabeschicht nutzt keine Aktivierungsfunktion.

Hinter den Kulissen besteht ein QNetwork aus zwei Teilen – einem Encoding-Netz zum Verarbeiten der Beobachtungen, gefolgt von einer Dense-Ausgabeschicht, die einen Q-Wert pro Aktion ausgibt. Die TF-Agents-Klasse `EncodingNetwork` implementiert eine neuronale Netzarchitektur, die sich in vielen Agenten findet (siehe Abbildung 18-14).

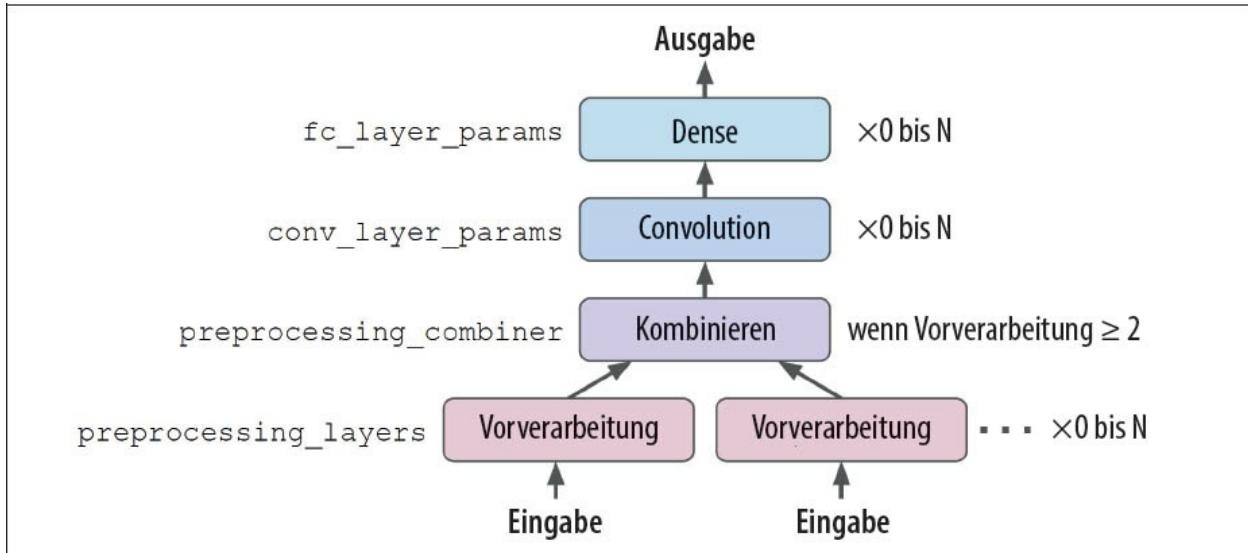
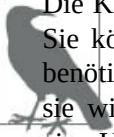


Abbildung 18-14: Architektur eines Encoding-Netzes

Es kann eine oder mehrere Eingaben haben. Besteht jede Beobachtung beispielsweise aus Sensordaten und einem Kamerabild, werden Sie zwei Eingaben haben. Für jede Eingabe können Vorverarbeitungsschritte erforderlich sein. Dann können Sie als Argument `preprocessing_layers` eine Liste mit Keras-Schichten mitgeben – eine Schicht pro Eingabe –, und das Netz wird jede Schicht auf die entsprechende Eingabe anwenden (benötigt eine Eingabe mehrere Vorverarbeitungsschichten, können Sie ein ganzes Modell mitgeben, da ein Keras-Modell immer auch als Schicht verwendet werden kann). Gibt es zwei oder mehr Eingaben, müssen Sie als Argument `preprocessing_combiner` auch noch eine zusätzliche Schicht mitgeben, die die Ausgaben der Vorverarbeitungsschichten einer einzelnen Eingabe

zusammenführt.

Als Nächstes wendet das Encoding-Netz optional sequenziell eine Liste von Convolutional Layers an, sofern Sie deren Parameter über das Argument `conv_layer_params` spezifiziert haben. Dabei muss es sich um eine Liste mit 3-Tupeln handeln (eines pro Convolutional Layer), die die Anzahl an Filtern, die Kernelgröße und die Schrittweite angeben. Nach diesen Convolutional Layers wird das Encoding-Netz optional auch noch eine Sequenz von Dense-Schichten anwenden, wenn Sie das Argument `fc_layer_params` mitgeben: Das ist eine Liste mit der Anzahl an Neuronen für jede Dense-Schicht. Optional können Sie mit dem Argument `dropout_layer_params` auch noch eine Liste mit Drop-out-Raten (eine pro Dense-Schicht) festlegen, wenn Sie nach jeder Dense-Schicht Drop-out anwenden wollen. Das `QNetwork` nimmt die Ausgabe dieses Encoding-Netzes und übergibt sie an die Dense-Ausgabeschicht (mit einer Einheit pro Aktion).



Die Klasse `QNetwork` ist flexibel genug, um viele verschiedene Architekturen aufzubauen, aber Sie können auch immer Ihre eigene Netzklasse erstellen, wenn Sie zusätzliche Funktionalität benötigen: Erweitern Sie die Klasse `tf_agents.networks.Network` und implementieren Sie sie wie eine normale eigene Keras-Schicht. Die Klasse `tf_agents.networks.Network` ist eine Unterklasse von `keras.layers.Layer`, und sie besitzt zusätzliche Funktionalität, die von manchen Agenten benötigt wird, wie zum Beispiel die Möglichkeit, flache Kopien des Netzes zu erstellen (also die Architektur ohne die Gewichte zu kopieren). Dies nutzt beispielsweise der `DQNAgent`, um eine Kopie des Onlinemodells zu erzeugen.

Nachdem wir nun das DQN haben, können wir den DQN-Agenten bauen.

DQN-Agenten erstellen

Die TF-Agents-Bibliothek implementiert viele Arten von Agenten im Paket `tf_agents.agents` und dessen Unterpaketen. Wir werden die Klasse `tf_agents.agents.dqn.DqnAgent` verwenden:

```
from tf_agents.agents.dqn.dqn_agent import DqnAgent

train_step = tf.Variable(0)

update_period = 4 # das Modell alle 4 Schritte trainieren

optimizer = keras.optimizers.RMSprop(lr=2.5e-4, rho=0.95, momentum=0.0,
                                      epsilon=0.00001, centered=True)

epsilon_fn = keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=1.0, # initiales  $\epsilon$ 
    decay_steps=250000 // update_period, # <=> 1,000,000 ALE-Frames
    end_learning_rate=0.01) # finales  $\epsilon$ 

agent = DqnAgent(tf_env.time_step_spec(),
```

```

        tf_env.action_spec(),

        q_network=q_net,
        optimizer=optimizer,
        target_update_period=2000, # <=> 32,000 ALE-Frames
        td_errors_loss_fn=keras.losses.Huber(reduction="none"),
        gamma=0.99, # Discount-Faktor
        train_step_counter=train_step,
        epsilon_greedy=lambda: epsilon_fn(train_step))

agent.initialize()

```

Gehen wir diesen Code durch:

- Erst erstellen wir eine Variable, die die Anzahl an Trainingsschritten zählt.
- Dann bauen wir den Optimierer mit den Hyperparametern aus dem DQN-Artikel aus dem Jahr 2015.
- Anschließend erstellen wir ein `PolynomialDecay`-Objekt, das den ε -Wert für die ε -greedy-Collect-Policy für den aktuellen Trainingsschritt berechnet (es wird normalerweise genutzt, um die Lernrate absinken zu lassen, daher der Name der Argumente, aber es funktioniert auch gut, um andere Werte absinken zu lassen). Er wird von 1,0 in 1 Million ALE-Frames (entsprechend 250.000 Schritten, da wir ein Frame Skipping mit einer Periode von 4 verwenden) auf 0,01 sinken. Zudem trainieren wir den Agenten alle vier Schritte (also 16 ALE-Frames), daher wird ε eigentlich in über 62.500 *Trainingsschritten* absinken.
- Dann bauen wir den `DQNAgent`, übergeben ihm die Specs für den Zeitschritt und die Aktionen, das zu trainierende `QNetwork`, den Optimierer, die Anzahl an Trainingsschritten zwischen den Aktualisierungen für das Zielmodell, die Verlustfunktion, den Discount-Faktor, die Variable `train_step` und eine Funktion, die den ε -Wert zurückgibt (sie darf kein Argument übernehmen, daher nutzen wir ein Lambda, um den `train_step` zu übergeben).

Beachten Sie, dass die Verlustfunktion nicht den mittleren Fehler, sondern einen Fehler pro Instanz zurückgeben muss. Darum setzen wir `reduction="none"`.

- Schließlich initialisieren wir den Agenten.

Bauen wir jetzt den Replay Buffer und den Beobachter, der ihn füllen wird.

Replay Buffer und Beobachter erstellen

Die TF-Agents-Bibliothek stellt im Paket `tf_agents.replay_buffers` verschiedene Implementierungen für Replay Buffer bereit. Manche sind komplett in Python geschrieben (ihre

Modulnamen beginnen mit `py_`), andere basieren auf TensorFlow (dort beginnen die Modulnamen mit `tf_`). Wir werden die Klasse `TFUniformReplay Buffer` aus dem Paket `tf_agents.replay_buffers.tf_uniform_replay_buffer` verwenden. Sie bietet eine leistungsfähige Implementierung eines Replay Buffer mit gleichverteiltem Sampling.²⁰

```
from tf_agents.replay_buffers import tf_uniform_replay_buffer

replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=tf_env.batch_size,
    max_length=1000000)
```

Schauen wir uns die Argumente an:

`data_spec`

Die Spezifikation der Daten, die im Replay Buffer gespeichert werden. Der DQN-Agent weiß, wie die eingesammelten Daten aussehen werden, und stellt die Daten-Spec über sein Attribut `collect_data_spec` zur Verfügung, daher übergeben wir dieses an den Replay Buffer.

`batch_size`

Die Anzahl an Trajektorien, die bei jedem Schritt hinzugefügt werden. In unserem Fall ist das eine, da der Fahrer nur eine Aktion pro Schritt ausführen und eine Trajektorie einsammeln wird. Würde es sich bei der Umgebung um eine *Batched-Umgebung* handeln, also eine, die bei jedem Schritt einen Batch mit Aktionen vornimmt und einen Batch mit Beobachtungen zurückgibt, müsste der Fahrer bei jedem Schritt einen Batch mit Trajektorien sichern. Da wir einen TensorFlow-Replay-Buffer nutzen, muss dieser die Größe der zu verarbeitenden Batches kennen (um den Rechengraphen zu erstellen). Ein Beispiel einer Batched-Umgebung ist `ParallelPyEnvironment` (aus dem Paket `tf_agents.environments_parallel_py_environment`): Sie führt in eigenen Prozessen mehrere Umgebungen parallel aus (sie können verschieden sein, solange sie die gleichen Aktions- und Beobachtungs-Specs besitzen), und bei jedem Schritt übernimmt sie einen Batch mit Aktionen und führt diese in den Umgebungen aus (eine Aktion pro Umgebung), um danach alle sich daraus ergebenden Beobachtungen zurückzuliefern.

`max_length`

Die maximale Größe des Replay Buffer. Wir haben einen großen Buffer erstellt, der eine Million Trajektorien speichern kann (wie im 2015er-DQN-Artikel). Das erfordert viel RAM.

Legen wir zwei aufeinanderfolgende Trajektorien ab, enthalten sie zwei aufeinanderfolgende Beobachtungen mit jeweils vier Frames (da wir den Wrapper `FrameStack4` eingesetzt haben), und leider sind drei der vier Frames in der zweiten Beobachtung redundant (sie sind schon in der ersten Beobachtung vorhanden). Mit anderen Worten: Wir verwenden etwa viermal mehr RAM als nötig. Um das zu vermeiden, können Sie stattdessen einen `PyHashed ReplayBuffer` aus dem Paket `tf_agents.replay_buffers.py_hashed_replay_buffer` verwenden: Er dedupliziert Daten in den gespeicherten Trajektorien entlang der letzten Beobachtungsachse.

Jetzt können wir den Beobachter erstellen, der die Trajektorien in den Replay Buffer schreiben wird. Ein Beobachter ist einfach eine Funktion (oder ein aufrufbares Objekt), die ein `trajectory`-Argument übernimmt, daher können wir die Methode `add_batch()` (gebunden an das Objekt `replay_buffer`) direkt als unseren Beobachter verwenden:

```
replay_buffer_observer = replay_buffer.add_batch
```

Wollen Sie Ihren eigenen Beobachter erstellen, können Sie eine beliebige Funktion mit einem Argument `trajectory` schreiben. Muss sie einen Status haben, können Sie eine Klasse mit einer Methode `__call__(self, trajectory)` erstellen. Dies ist beispielsweise ein einfacher Beobachter, der bei jedem Aufruf einen Zähler erhöht (außer wenn die Trajektorie eine Grenze zwischen zwei Episoden darstellt, was nicht als Schritt zählt) und alle 100 Schritte den Fortschritt in Bezug auf einen gegebenen Gesamtwert ausgibt (das Carriage Return \r zusammen mit `end=""` sorgt dafür, dass der angezeigte Zähler auf der gleichen Zeile bleibt):

```
class ShowProgress:

    def __init__(self, total):
        self.counter = 0
        self.total = total

    def __call__(self, trajectory):
        if not trajectory.is_boundary():
            self.counter += 1
        if self.counter % 100 == 0:
            print("\r{}/{}".format(self.counter, self.total), end="")
```

Erstellen wir nun ein paar Trainingsmetriken.

Trainingsmetriken erstellen

TF-Agents implementiert diverse RL-Metriken im Paket `tf_agents.metrics` – manche rein in Python, andere auf TensorFlow basierend. Erstellen wir ein paar, um die Anzahl an Episoden und Schritten, aber vor allem den mittleren Return pro Episode und die durchschnittliche Episodenlänge zu erfassen:

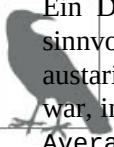
```
from tf_agents.metrics import tf_metrics

train_metrics = [
    tf_metrics.NumberOfEpisodes(),
    tf_metrics.EnvironmentSteps(),
```

```

    tf_metrics.AverageReturnMetric(),
    tf_metrics.AverageEpisodeLengthMetric(),
]


```

 Ein Discounting der Belohnungen ist beim Training oder zum Implementieren einer Policy sinnvoll, da Sie so die Wichtigkeit sofortiger Belohnungen und zukünftiger Belohnungen austarieren können. Aber ist eine Episode vorbei, können wir auswerten, wie gut sie insgesamt war, indem wir die Belohnungen *ohne Discount* aufsummieren. Aus diesem Grund berechnet die Average ReturnMetric die Summe der Belohnungen ohne Discount für jede Episode und merkt sich den gleitenden Durchschnitt dieser Summen über alle Episoden.

Sie können sich jederzeit den Wert einer dieser Metriken über den Aufruf ihrer Methode `result()` ausgeben lassen (zum Beispiel `train_metrics[0].result()`). Alternativ können Sie alle Metriken über einen Aufruf von `log_metrics(train_metrics)` protokollieren (diese Funktion finden Sie im Paket `tf_agents.eval.metric_utils`):

```

>>> from tf_agents.eval.metric_utils import log_metrics
>>> import logging
>>> logging.getLogger().setLevel(logging.INFO)
>>> log_metrics(train_metrics)
[...]
NumberOfEpisodes = 0
EnvironmentSteps = 0
AverageReturn = 0.0
AverageEpisodeLength = 0.0

```

Als Nächstes wollen wir den Collect-Fahrer erstellen.

Collect-Fahrer erstellen

Wie wir in [Abbildung 18-13](#) gesehen haben, handelt es sich bei einem Fahrer um ein Objekt, das mit einer gegebenen Policy eine Umgebung erforscht, Erfahrungen sammelt und sie an Beobachter weitergibt. Bei jedem Schritt geschieht Folgendes:

- Der Fahrer übergibt den aktuellen Zeitschritt an die Collect Policy, die ihn nutzt, um eine Aktion auszuwählen, und ein *Aktionsschritt*-Objekt mit der Aktion zurückgibt.
- Der Fahrer übergibt dann die Aktion an die Umgebung, die den nächsten Schritt zurückgibt.
- Schließlich erstellt der Fahrer ein Trajektorienobjekt, um diesen Übergang zu

repräsentieren und ihn an alle Beobachter weiterzugeben.

Manche Policies, wie zum Beispiel RNN-Policies, sind zustandsbehaftet: Sie wählen eine Aktion aufgrund des gegebenen Zeitschritts und ihres eigenen internen Zustands aus. Zustandsbehaftete Policies geben im Aktionsschritt neben der gewählten Aktion auch ihren eigenen Status mit. Der Fahrer übergibt dann diesen Status beim nächsten Zeitschritt wieder zurück an die Policy. Zudem sichert er den Policy-Zustand in der Trajektorie (im Feld `policy_info`), womit er im Replay Buffer landet. Das ist für das Trainieren einer zustandsbehafteten Policy entscheidend: Sampelt der Agent eine Trajektorie, muss er den Status der Policy in den Status versetzen, in dem sie sich beim gesampelten Zeitschritt befand.

Wie zuvor schon besprochen, kann die Umgebung zudem eine Batched-Umgebung sein – dann übergibt der Fahrer einen *Batched-Zeitschritt* an die Policy (also ein Zeitschrittobjekt mit einem Batch an Beobachtungen, einem Batch an Schrittypen, einem Batch an Belohnungen und einem Batch mit Discounts – alle vier Batches mit der gleichen Größe). Der Fahrer übergibt zudem einen Batch mit vorherigen Policy-Zuständen. Die Policy gibt dann einen *Batched-Aktionsschritt* mit einem Batch an Aktionen und einem Batch mit Policy-Zuständen zurück. Schließlich erzeugt der Fahrer eine *Batched-Trajektorie* (also eine Trajektorie mit einem Batch an Schrittypen, einem Batch an Beobachtungen, einem Batch an Aktionen, einem Batch an Belohnungen und ganz allgemein einem Batch für jedes Trajektorienattribut, wobei alle Batches die gleiche Größe besitzen).

Es gibt zwei zentrale Fahrerklassen: `DynamicStepDriver` und `DynamicEpisodeDriver`. Die erste sammelt Erfahrungen für eine gegebene Anzahl an Schritten ein, die zweite für eine gegebene Anzahl an Episoden. Wir wollen Erfahrungen in jeder Trainingsiteration für vier Schritte einsammeln (wie im 2015er DQN-Artikel), daher erstellen wir einen `DynamicStepDriver`:

```
from tf_agents.drivers.dynamic_step_driver import DynamicStepDriver

collect_driver = DynamicStepDriver(
    tf_env,
    agent.collect_policy,
    observers=[replay_buffer_observer] + training_metrics,
    num_steps=update_period) # in jeder Trainingsiteration 4 Schritte einsammeln
```

Wir übergeben die Umgebung, in der gespielt werden soll, die Collect Policy des Agenten, eine Liste mit Beobachtern (einschließlich des Replay-Buffer-Beobachters und der Trainingsmetriken) und schließlich die Anzahl an auszuführenden Schritten (in diesem Fall vier). Jetzt könnten wir ihn durch den Aufruf seiner Methode `run()` starten, aber es ist besser, den Replay Buffer erst mit Erfahrungen vorzuwärmern, die mithilfe einer rein zufälligen Policy gesammelt wurden. Dazu können wir die Klasse `RandomTFPolicy` nutzen und einen zweiten Fahrer erzeugen, der diese Policy 20.000 Schritte lang ausführt (was 80.000 Simulator-Frames

entspricht, wie das auch im DQN-Artikel aus dem Jahr 2015 getan wurde). Wir können unseren ShowProgress-Beobachter verwenden, um den Fortschritt anzuzeigen:

```
from tf_agents.policies.random_tf_policy import RandomTFPolicy

initial_collect_policy = RandomTFPolicy(tf_env.time_step_spec(),
                                         tf_env.action_spec())

init_driver = DynamicStepDriver(
    tf_env,
    initial_collect_policy,
    observers=[replay_buffer.add_batch, ShowProgress(20000)],
    num_steps=20000) # <=> 80,000 ALE frames

final_time_step, final_policy_state = init_driver.run()
```

Jetzt sind wir fast so weit, die Trainingsschleife laufen zu lassen. Es fehlt aber noch eine letzte Komponente: das Dataset.

Dataset erstellen

Um einen Batch mit Trajektorien aus dem Replay Buffer zu sampeln, rufen Sie dessen Methode `get_next()` auf. Diese gibt den Batch mit Trajektorien und ein `Buffer_Info`-Objekt zurück, das die Sample-Kennungen und deren Sampling-Wahrscheinlichkeiten enthält (das kann für manche Algorithmen wie PER nützlich sein). Der folgende Code sampelt beispielsweise einen kleinen Batch mit zwei Trajektorien (Unterepisoden), die jeweils drei aufeinanderfolgende Schritte enthalten. Diese Unterepisoden sind in [Abbildung 18-15](#) zu sehen (jede Zeile enthält drei aufeinanderfolgende Schritte aus einer Episode):

```
>>> trajectories, buffer_info = replay_buffer.get_next(
...     sample_batch_size=2, num_steps=3)
...
>>> trajectories._fields
('step_type', 'observation', 'action', 'policy_info',
'next_step_type', 'reward', 'discount')
>>> trajectories.observation.shape
TensorShape([2, 3, 84, 84, 4])
>>> trajectories.step_type.numpy()
```

```
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

Das `trajectories`-Objekt ist ein benanntes Tupel mit sieben Feldern. Jedes Feld enthält einen Tensor, dessen erste beiden Dimensionen 2 und 3 sind (da es zwei Trajektorien mit jeweils drei Schritten gibt). Das erklärt, warum die Form des `observation`-Felds $[2, 3, 84, 84, 4]$ ist: zwei Trajektorien mit jeweils drei Schritten, und die Beobachtung jedes Schritts ist $84 \times 84 \times 4$. Analog dazu hat der `step_type`-Tensor die Form $[2, 3]$: In diesem Beispiel enthalten beide Trajektorien drei aufeinanderfolgende Schritte in der Mitte einer Episode (Typen 1, 1, 1). In der zweiten Trajektorie können Sie den Ball unten links in der ersten Beobachtung kaum sehen, und in den nächsten beiden Beobachtungen ist er ganz verschwunden. Der Agent wird also ein Leben verlieren, aber die Episode wird nicht sofort enden, weil immer noch genug Leben vorhanden sind.

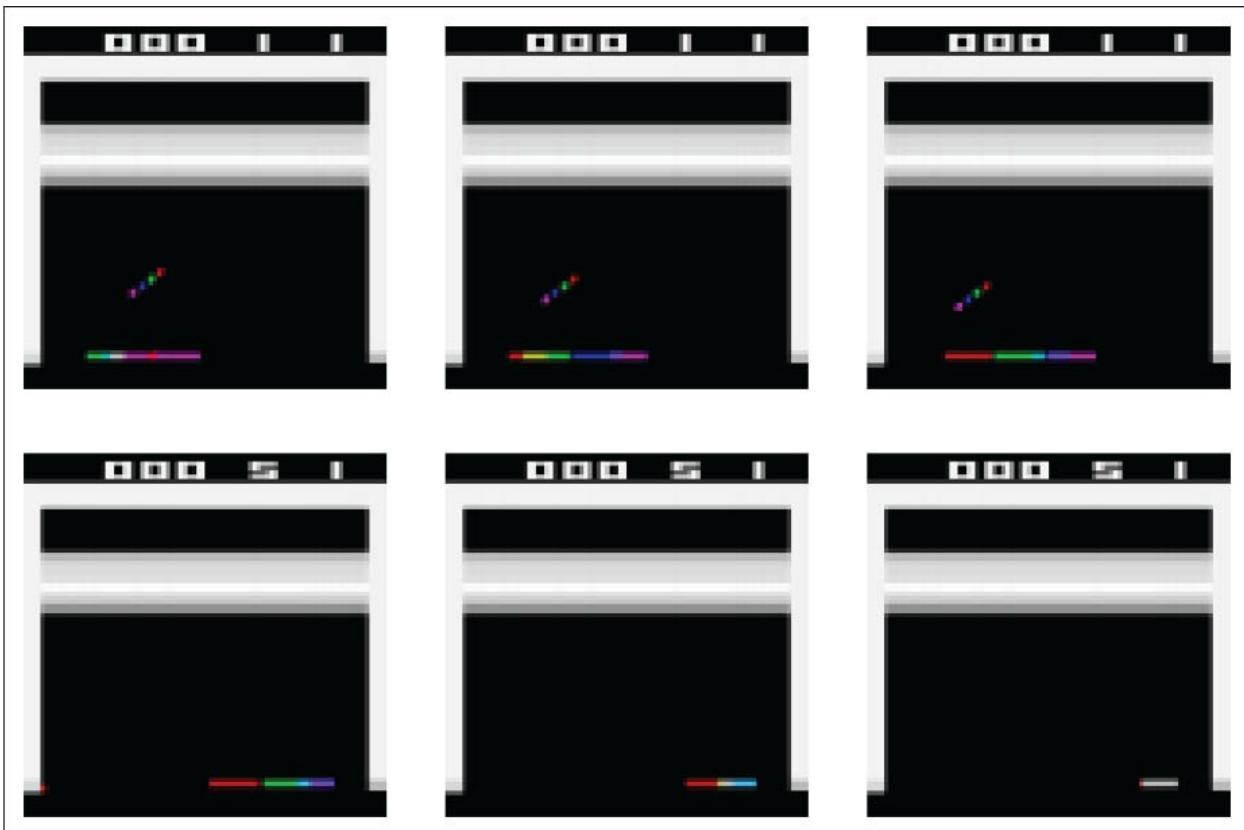


Abbildung 18-15: Zwei Trajektorien enthalten jeweils drei aufeinanderfolgende Schritte.

Jede Trajektorie ist eine knappe Repräsentation einer Sequenz aufeinanderfolgender Zeitschritte und Aktionsschritte, die so designt wurde, dass Redundanz vermieden wird. Wie? Nun, wie Sie in [Abbildung 18-16](#) sehen können, besteht Übergang n aus Zeitschritt n , Aktionsschritt n und Zeitschritt $n + 1$, während Übergang $n + 1$ aus Zeitschritt $n + 1$, Aktionsschritt $n + 1$ und Zeitschritt $n + 2$ besteht. Würden wir diese beiden Übergänge direkt im Replay Buffer ablegen, wäre Zeitschritt $n + 1$ doppelt vorhanden. Um das zu vermeiden, enthält der n .

Trajektorienschritt nur den Typ und die Beobachtung von Zeitschritt n (nicht seine Belohnung oder den Discount), aber nicht die Beobachtung aus Zeitschritt $n + 1$ (er enthält allerdings eine Kopie des Typs des nächsten Zeitschritts – das ist die einzige Redundanz).

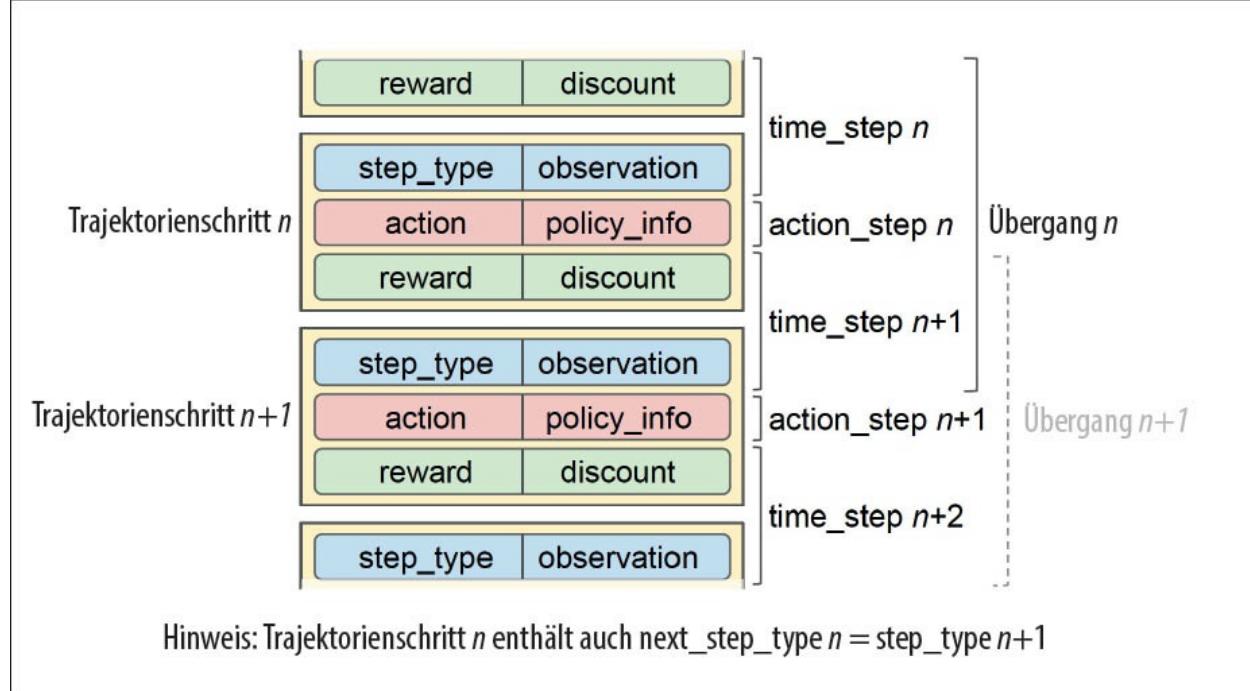


Abbildung 18-16: Trajektorien, Übergänge, Zeitschritte und Aktionsschritte

Haben Sie also einen Batch mit Trajektorien, bei denen jede Trajektorie $t + 1$ Schritte enthält (von Zeitschritt n bis Zeitschritt $n + t$), enthält er alle Daten der Zeitschritte n bis $n + t$ mit Ausnahme der Belohnung und des Discounts für Zeitschritt n (aber er enthält die Belohnung und den Discount von Zeitschritt $n + t + 1$). Das repräsentiert t Übergänge (n nach $n + 1$, $n + 1$ nach $n + 2$, ..., $n + t - 1$ nach $n + t$).

Die Funktion `to_transition()` im Modul `tf_agents.trajectories.trajectory` wandelt eine Batched-Trajektorie in eine Liste mit einem Batched-time_step, einem Batched-action_step und einem Batched-next_time_step um. Beachten Sie, dass die zweite Dimension 2 statt 3 ist, da es t Übergänge zwischen $t + 1$ Zeitschritten gibt (machen Sie sich keine Sorgen, wenn Sie jetzt ein bisschen verwirrt sind – Sie werden sich daran gewöhnen):

```
>>> from tf_agents.trajectories.trajectory import to_transition
>>> time_steps, action_steps, next_time_steps = to_transition(trajectories)
>>> time_steps.observation.shape
TensorShape([2, 2, 84, 84, 4]) # 3 Zeitschritte = 2 Übergänge
```

Eine gesampelte Trajektorie kann über zwei (oder mehr) Episoden gehen! In diesem Fall wird sie **Grenzübergänge** (Boundary Transitions) enthalten, also solche mit einem `step_type` gleich 2

(Ende) und einem `next_step_type` gleich 0 (Start). Natürlich geht TF-Agents korrekt mit solchen Trajektorien um (zum Beispiel setzt es den Policy-Status zurück, wenn es auf eine Grenze trifft). Die Methode `is_boundary()` der Trajektorie gibt einen Tensor zurück, der für jeden Schritt angibt, ob er eine Grenze ist oder nicht.

Für unsere Haupt-Trainingsschleife rufen wir nicht die Methode `get_next()` auf, sondern wir werden ein `tf.data.Dataset` verwenden. So können wir von der Leistungsfähigkeit der Data-API profitieren (zum Beispiel Parallelisierung und Prefetching). Dazu rufen wir die Methode `as_dataset()` des Replay Buffer auf:

```
dataset = replay_buffer.as_dataset(  
    sample_batch_size=64,  
    num_steps=2,  
    num_parallel_calls=3).prefetch(3)
```

Wir werden bei jedem Trainingsschritt Batches mit 64 Trajektorien sampeln (wie im 2015er-DQN-Artikel), jeweils mit zwei Schritten (zwei Schritte = ein vollständiger Übergang einschließlich der Beobachtung des nächsten Schritts). Dieses Dataset wird drei Elemente parallel verarbeiten und drei Batches prefetchen.

Für On-Policy-Algorithmen (wie Policy-Gradienten) sollte jede Erfahrung einmal gesampelt, im Training verwendet und dann verworfen werden. In diesem Fall können Sie trotzdem einen Replay Buffer einsetzen, aber statt ein `Dataset` zu verwenden, würden Sie bei jeder Trainingsiteration die Methode `gather_all()` des Replay Buffer nutzen, um einen Tensor mit allen bisher aufgezeichneten Trajektorien zu erhalten, diesen dann zum Durchführen eines Trainingsschritts verwenden und schließlich den Replay Buffer durch Aufruf seiner Methode `clear()` leeren.

Nachdem wir nun alle Komponenten zusammen haben, können wir das Modell trainieren!

Trainingsschleife erstellen

Um das Training zu beschleunigen, werden wir die Hauptfunktionen in Tensor-Flow Functions umwandeln. Dazu nutzen wir die Funktion `tf_agents.utils.common.function()`, die `tf.function()` mit ein paar zusätzlichen experimentellen Optionen verpackt:

```
from tf_agents.utils.common import function  
  
collect_driver.run = function(collect_driver.run)  
agent.train = function(agent.train)
```

Erstellen wir eine kleine Funktion, die die Haupt-Trainingsschleife für `n_iterations` ausführt:

```
def train_agent(n_iterations):
```

```

time_step = None

policy_state = agent.collect_policy.get_initial_state(tf_env.batch_size)

iterator = iter(dataset)

for iteration in range(n_iterations):

    time_step, policy_state = collect_driver.run(time_step, policy_state)

    trajectories, buffer_info = next(iterator)

    train_loss = agent.train(trajectories)

    print("\r{} loss:{:.5f}".format(
        iteration, train_loss.loss.numpy()), end="")

    if iteration % 1000 == 0:

        log_metrics(train_metrics)

```

Die Funktion fragt zuerst bei der Collect Policy nach ihrem initialen Status (für die gegebene Umgebungs-Batchgröße, die hier 1 ist). Da die Policy zustandslos ist, wird ein leeres Tupel zurückgegeben (wir hätten also auch `policy_state = ()` schreiben können). Als Nächstes erstellen wir einen Iterator über das Dataset und führen die Trainingsschleife aus. Bei jeder Iteration rufen wir die Methode `run()` des Fahrers auf, übergeben den aktuellen Zeitschritt (initial `None`) und den aktuelle Policy-Status. Sie wird die Collect Policy ausführen, die Erfahrung aus vier Schritten (wie zuvor konfiguriert) einsammeln und die eingesammelten Trajektorien an den Replay Buffer und die Metriken weitergeben. Als Nächstes sammeln wir einen Batch mit Trajektorien aus dem Dataset und übergeben es an die Methode `train()` des Agenten. Diese gibt ein `train_loss`-Objekt zurück, das vom Typ des Agenten abhängig sein kann. Nun zeigen wir die Iteration und den Trainingsverlust an, und alle 1.000 Iterationen protokollieren wir alle Metriken. Jetzt können Sie einfach `train_agent()` mit einer Anzahl an Iterationen aufrufen und zusehen, wie der Agent nach und nach lernt, *Breakout* zu spielen!

```
train_agent(10000000)
```

Das wird ziemlich viel Rechenleistung beanspruchen und Ihnen Geduld abverlangen (es kann abhängig von Ihrer Hardware Stunden oder sogar Tage dauern), zudem müssen Sie den Algorithmus eventuell mehrfach mit unterschiedlichen Zufalls-Seeds laufen lassen, um gute Ergebnisse zu erhalten. Ist das aber einmal geschafft, wird der Agent übermenschlich sein (zumindest bei *Breakout*). Sie können auch versuchen, diesen DQN-Agenten mit anderen Atari-Spielen zu trainieren: Er kann in den meisten Actionspielen übermenschlich werden, ist aber nicht so gut in Spielen mit längeren Geschichten.²¹

Überblick über beliebte RL-Algorithmen

Bevor wir dieses Kapitel beenden, wollen wir uns noch schnell ein paar beliebte RL-Algorithmen anschauen:

Actor-Critic-Algorithmus

Eine Familie von RL-Algorithmen, die Policy-Gradienten mit Deep-Q-Netzen kombinieren. Ein Actor-Critic-Agent besteht aus zwei neuronalen Netzen – einem Policy-Netz und einem DQN. Das DQN wird normal trainiert, indem es von den Erfahrungen des Agenten lernt. Das Policy-Netz lernt anders (und viel schneller) als in normalen PGs: Statt den Wert jeder Aktion abzuschätzen, indem es mehrere Episoden durchläuft, dann die zukünftigen Belohnungen mit Discount für jede Aktion aufsummiert und sie schließlich normalisiert, basiert der Agent (Actor) auf den Aktionswerten, die vom DQN geschätzt wurden. Es ist ein bisschen wie ein Athlet (der Agent), der mit der Hilfe eines Coachs (dem DQN) lernt.

*Asynchronous Advantage Actor-Critic (<https://homl.info/a3c>) (A3C)*²²

Eine wichtige Variante des Actor-Critic-Algorithmus, die 2016 von Deep-Mind-Forschern vorgestellt wurde. Mehrere Agenten lernen parallel und erforschen die Umgebung in eigenen Kopien. Regelmäßig, aber asynchron (daher der Name), befördert jeder Agent Aktualisierungen der Gewichte in ein Master-Netz, dann holt er die neuesten Gewichte dort wieder heraus. So trägt der Agent dazu bei, das Master-Netz zu verbessern, während er davon profitiert, was die anderen Agenten gelernt haben. Zudem schätzt das DQN nicht die Q-Werte, sondern den Vorteil jeder Aktion (daher das zweite A im Namen), was das Training stabilisiert.

Advantage Actor-Critic (<https://homl.info/a2c>) (A2C)

Eine Variante des A3C-Algorithmus ohne Asynchronität. Alle Modellaktualisierungen geschehen synchron, daher werden Gradientenaktualisierungen auf größeren Batches durchgeführt, was es dem Modell erlaubt, die GPU besser auszunutzen.

*Soft Actor-Critic (<https://homl.info/sac>) (SAC)*²³

Eine 2018 von Tuomas Haarnoja und anderen Forschern der UC Berkeley vorgeschlagene Actor-Critic-Variante. Sie lernt nicht nur Belohnungen, sondern auch, die Entropie ihrer Aktionen zu maximieren. Mit anderen Worten: Sie versucht, so unvorhersehbar wie möglich zu sein, gleichzeitig aber möglichst viele Belohnungen einzusammeln. Das unterstützt den Agenten darin, die Umgebung zu erforschen, was wiederum das Training beschleunigt und es weniger wahrscheinlich macht, die gleiche Aktion immer wieder auszuführen, wenn das DQN nicht perfekte Schätzungen erzeugt. Dieser Algorithmus hat eine erstaunliche Sample-Effizienz gezeigt (im Gegensatz zu allen zuvor vorgestellten Algorithmen, die sehr langsam lernen). SAC steht in TF-Agents zur Verfügung.

*Proximal Policy Optimization (<https://homl.info/ppo>) (PPO)*²⁴

Ein auf A2C basierender Algorithmus, der die Verlustfunktion beschneidet, um viel zu große Gewichtsaktualisierungen zu vermeiden (was oft zu Trainingsinstabilitäten führt). PPO ist eine Vereinfachung des älteren Trust-Region-Policy- Optimization-(TRPO-)Algorithmus (<https://homl.info/trpo>)²⁵, der ebenfalls von John Schulman und anderen

OpenAI-Forschern stammt. OpenAI schaffte es im April 2019 in die Nachrichten mit ihrer KI namens OpenAI Five, die auf dem PPO-Algorithmus basiert und den Weltmeister im Multiplayer-Spiel *Dota 2* schlug. PPO steht ebenfalls in TF-Agents bereit.

Curiosity-Based Exploration (<https://homl.info/curiosity>)²⁶

In RL gibt es immer wieder Probleme mit der Spärlichkeit der Belohnungen, wodurch das Lernen sehr langsam und ineffektiv wird. Deepak Pathak und andere Forscher der UC Berkeley haben einen interessanten Weg vorgeschlagen, um das Problem anzugehen: Warum ignorieren wir nicht die Belohnungen und sorgen nur dafür, dass der Agent ausgesprochen neugierig darauf ist, die Umgebung zu erforschen? Die Belohnungen werden damit für den Agenten intrinsisch, statt aus der Umgebung zu kommen. Genauso wird es bei einem Kind erfolgreicher sein, die Neugier zu wecken, als es einfach für gute Noten zu belohnen. Wie funktioniert das? Der Agent versucht fortlaufend, das Ergebnis seiner Aktionen vorherzusagen, und sucht Situationen, in denen das Ergebnis nicht seinen Vorhersagen entspricht. Mit anderen Worten – er will überrascht werden. Ist das Ergebnis vorhersagbar (langweilig), geht er woanders hin. Ist das Ergebnis hingegen unvorhergesehen, aber der Agent hat keine Kontrolle darüber, wird er nach einer Weile auch wieder gelangweilt sein. Nur mit Neugier haben es die Autoren geschafft, einen Agenten für viele Videospiele zu trainieren. Der Agent wird zwar nicht dafür bestraft, wenn er verliert, aber das Spiel beginnt dann von Neuem, und das ist langweilig – also lernt er, es zu vermeiden.

Wir haben in diesem Kapitel viele Themen behandelt: Policy-Gradienten, Markov-Ketten, Markov-Entscheidungsprozesse, Q-Learning, approximatives Q-Learning und Deep-Q-Learning in seinen diversen Varianten (feste Q-Wert-Ziele, Double DQN, Dueling DQN und priorisiertes Experience Replay). Wir haben besprochen, wie man TF-Agents nutzt, um Agenten im großen Maßstab zu trainieren, und schließlich haben wir uns noch kurz ein paar andere Algorithmen angeschaut. Reinforcement Learning ist ein riesiges und spannendes Feld, tagtäglich gibt es neue Ideen und Algorithmen. Daher hoffe ich, dass dieses Kapitel bei Ihnen Neugier geweckt hat: Es gibt eine ganze Welt zu entdecken!

Übungen

1. Wie würden Sie Reinforcement Learning definieren? Wie unterscheidet es sich von gewöhnlichem überwachtem oder unüberwachtem 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. Nutzen Sie TF-Agents, um einen Agenten dafür zu trainieren, bei »SpaceInvaders-v4« mit einem der verfügbaren Algorithmen übermenschlich gut zu werden.
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](#).

TensorFlow-Modelle skalierbar trainieren und deployen

Jetzt haben Sie ein wunderschönes Modell, das erstaunliche Vorhersagen liefert. Aber was machen Sie damit? Nun, Sie müssen es in eine Produktivumgebung bringen! Eventuell ist das ganz einfach – Sie lassen das Modell auf einem Satz Daten laufen und schreiben ein Skript, das dieses Modell jede Nacht ausführt. Aber oft ist viel mehr Aufwand nötig. Eventuell müssen verschiedenste Teile Ihrer Infrastruktur dieses Modell mit Live-Daten verwenden – dann wollen Sie Ihr Modell vermutlich in einem Webservice verpacken: So kann jeder Teil Ihrer Infrastruktur das Modell jederzeit über eine einfache REST-API (oder ein anderes Protokoll) abfragen, so wie wir es in [Kapitel 2](#) besprochen haben. Aber im Laufe der Zeit müssen Sie Ihr Modell regelmäßig mit frischen Daten neu trainieren und die aktualisierte Version wieder produktiv stellen. Sie müssen mit Versionen des Modells umgehen, den Übergang von einem zum nächsten sauber ermöglichen, eventuell zu einem früheren Modell zurückkehren, wenn es Probleme gibt, und vielleicht mehrere verschiedene Modelle parallel laufen lassen, um *A/B-Experimente* durchzuführen.¹ Wird Ihr Produkt ein Erfolg, empfängt Ihr Service eventuell immer mehr *Queries pro Sekunde* (QPS) und muss hochskaliert werden, um die Last abarbeiten zu können. Eine großartige Lösung zum Skalieren Ihres Service, wie wir in diesem Kapitel sehen werden, ist der Einsatz von TF Serving – entweder auf Ihrer eigenen Hardwareinfrastruktur oder über einen Cloud-Service wie Google Cloud AI Platform. TF Serving kümmert sich um das effiziente Ausführen Ihres Modells, den sauberen Wechsel von einem Modell zum nächsten und vieles mehr. Nutzen Sie die Cloud-Plattform, werden Sie noch mehr Features erhalten, wie zum Beispiel leistungsfähige Monitoring-Tools.

Haben Sie sehr viele Trainingsdaten und rechenintensive Modelle, kann die Trainingszeit zudem untragbar lang sein. Muss Ihr Produkt schnell auf Änderungen reagieren können, kann eine lange Trainingszeit ein Showstopper sein (stellen Sie sich beispielsweise ein Empfehlungssystem für Nachrichten vor, das Vorschläge für Artikel von letzter Woche macht ...). Wichtiger ist vielleicht noch, dass eine lange Trainingsdauer Sie davon abhält, mit neuen Ideen zu experimentieren. Beim Machine Learning ist es (wie in vielen anderen Bereichen) schwer, im Voraus zu wissen, welche Ideen funktionieren werden, daher sollten Sie so viele wie möglich ausprobieren – und zwar so schnell wie möglich. Um das Training zu beschleunigen, können Sie beispielsweise Hardwarebeschleuniger wie GPUs oder TPUs nutzen. Um noch schneller zu werden, können Sie ein Modell über mehrere Maschinen hinweg trainieren, von denen jede mit mehreren Hardwarebeschleunigern ausgestattet ist. Die einfache, aber trotzdem leistungsfähige Distribution Strategies API von TensorFlow ist da eine große Hilfe, wie wir sehen werden.

In diesem Kapitel werden wir uns anschauen, wie wir Modelle deployen – zuerst mit TF Serving,

dann auf der Google Cloud AI Platform. Wir werden uns auch kurz mit dem Deployen von Modellen auf mobilen Apps, Embedded Devices und Webanwendungen befassen. Und schließlich werden wir darauf eingehen, wie man die Berechnungen mit GPUs beschleunigen kann und wie man Modelle über mehrere Devices und Server hinweg mit der Distribution Strategies API trainiert. Das sind ganz schön viele Themen, also legen wir am besten gleich los.

Ein TensorFlow-Modell ausführen

Haben Sie ein TensorFlow-Modell einmal trainiert, können Sie es ganz einfach in beliebigem Python-Code verwenden: Handelt es sich um ein tf.keras-Modell, rufen Sie einfach seine Methode `predict()` auf. Aber mit wachsender Infrastruktur kommen Sie an einen Punkt, an dem es besser ist, Ihr Modell in einem kleinen Service zu verpacken, dessen einziger Zweck es ist, Vorhersagen zu treffen und sich vom Rest der Infrastruktur abfragen zu lassen (zum Beispiel über eine REST- oder gRPC-API).² Das entkoppelt Ihr Modell vom Rest der Infrastruktur und ermöglicht es, leicht die Modellversionen zu wechseln oder den Service bei Bedarf zu skalieren (unabhängig vom Rest Ihrer Infrastruktur), A/B-Experimente durchzuführen und sicherzustellen, dass all Ihre Softwarekomponenten auf den gleichen Modellversionen aufbauen. Zudem vereinfacht es das Testen, die Entwicklung und mehr. Sie könnten Ihren eigenen Microservice mit der von Ihnen gewählten Technologie erstellen (zum Beispiel mit der Flask-Bibliothek), aber warum das Rad neu erfinden, wenn Sie einfach TF Serving nutzen können?

TensorFlow Serving verwenden

TF Serving ist ein sehr effizienter, kampferprobter Modellserver, der in C++ geschrieben ist. Er kann eine hohe Last verarbeiten, mehrere Versionen Ihres Modells bedienen, ein Modell-Repository beobachten, um automatisch die neuesten Versionen zu deployen, und noch vieles mehr (siehe Abbildung 19-1).

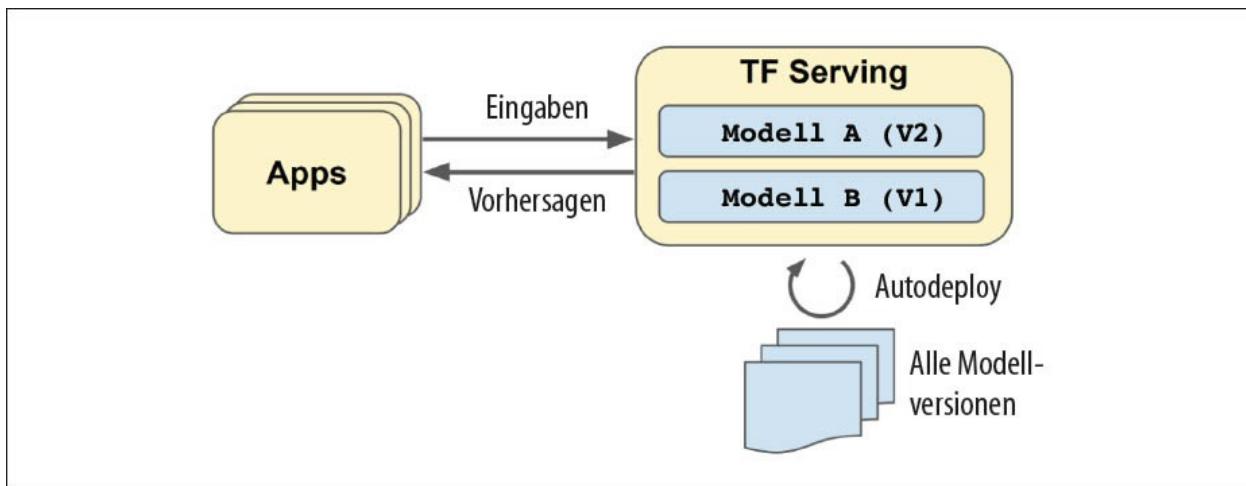


Abbildung 19-1: TF Serving kann mehrere Modelle betreiben und automatisch die neueste Version jedes Modells deployen.

Gehen wir einmal davon aus, dass Sie ein MNIST-Modell mit tf.keras trainiert haben und es

nach TF Serving deployen wollen. Als Erstes müssen Sie dieses Modell in TensorFlows *SavedModel-Format* exportieren.

SavedModels exportieren

TensorFlow stellt eine einfache Funktion `tf.saved_model.save()` bereit, um Modelle in das SavedModel-Format zu exportieren. Dazu müssen Sie nur das Modell angeben sowie Name und Versionsnummer festlegen, dann wird die Funktion den Rechengraphen und die Gewichte des Modells sichern:

```
model = keras.models.Sequential([...])  
  
model.compile([...])  
  
history = model.fit([...])  
  
model_version = "0001"  
  
model_name = "my_mnist_model"  
  
model_path = os.path.join(model_name, model_version)  
  
tf.saved_model.save(model, model_path)
```

Es ist im Allgemeinen eine gute Idee, alle Vorverarbeitungsschichten in das finale Modell mit aufzunehmen, das Sie exportieren, sodass es die Daten in ihrer natürlichen Form übernehmen kann, nachdem es in die Produktivumgebung deployt wurde. Damit vermeiden Sie, sich in der Anwendung, die das Modell nutzt, separat um die Vorverarbeitung kümmern zu müssen. Das Bündeln der Vorverarbeitungsschritte im Modell macht es auch leichter, sie später zu aktualisieren, und es begrenzt das Risiko von Abweichungen zwischen dem Modell und den Vorverarbeitungsschritten, die dafür notwendig sind.

Da ein SavedModel den Rechengraphen sichert, kann es nur für Modelle genutzt werden, die exklusiv auf TensorFlow-Operationen basieren und nicht die Operation `tf.py_function()` verwenden (die beliebigen Python-Code verpackt). Auch dynamische `tf.keras`-Modelle sind ausgeschlossen (siehe [Anhang G](#)), da diese Modelle nicht in Rechengraphen umgewandelt werden können. Dynamische Modelle müssen über andere Tools bereitgestellt werden (zum Beispiel Flask).

Ein SavedModel repräsentiert eine Version Ihres Modells. Es wird als Verzeichnis abgelegt, in dem eine Datei `saved_model.pb` enthalten ist. Diese definiert den Rechengraphen (repräsentiert als serialisierter Protocol Buffer). Dazu kommt ein Unterverzeichnis `variables` mit den Werten der Variablen. Bei Modellen mit einer großen Zahl von Gewichten sind diese Variablenwerte eventuell auf mehrere Dateien verteilt. Ein SavedModel enthält zudem ein Unterverzeichnis `assets`, in dem zusätzliche Daten zu finden sind, wie zum Beispiel Vokabulardateien, Klassennamen oder Beispieldaten für dieses Modell. Die Verzeichnisstruktur sieht wie folgt aus (hier kommen keine Assets zum Einsatz):

```
my_mnist_model
└── 0001
    ├── assets
    ├── saved_model.pb
    └── variables
        ├── variables.data-00000-of-00001
        └── variables.index
```

Wie Sie sich vielleicht schon gedacht haben, können Sie ein SavedModel mit der Funktion `tf.saved_model.load()` laden. Aber das zurückgelieferte Objekt ist kein Keras-Modell: Es repräsentiert das SavedModel mit seinem Rechengraphen und den Werten der Variablen. Sie können es wie eine Funktion verwenden, und es wird Vorhersagen treffen (achten Sie darauf, die Eingaben als Tensoren zu übergeben, zudem müssen Sie das Argument `training` setzen – üblicherweise auf `False`):

```
saved_model = tf.saved_model.load(model_path)
y_pred = saved_model(X_new, training=False)
```

Alternativ können Sie die Vorhersagefunktion dieses SavedModel in einem Keras-Modell verpacken:

```
inputs = keras.layers.Input(shape=...)
outputs = saved_model(inputs, training=False)
model = keras.models.Model(inputs=[inputs], outputs=[outputs])
y_pred = model.predict(X_new)
```

TensorFlow bringt auch ein kleines Befehlszeilentool `saved_model_cli` mit, um sich SavedModels anschauen zu können:

```
$ export ML_PATH="$HOME/ml" # auf dieses Projekt zeigen
$ cd $ML_PATH
$ saved_model_cli show --dir my_mnist_model/0001 --all
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:
signature_def['__saved_model_init_op']:
```

```
[...]
```

```
signature_def['serving_default']:
```

```
The given SavedModel SignatureDef contains the following input(s):
```

```
    inputs['flatten_input'] tensor_info:
```

```
        dtype: DT_FLOAT
```

```
        shape: (-1, 28, 28)
```

```
        name: serving_default_flatten_input:0
```

```
The given SavedModel SignatureDef contains the following output(s):
```

```
    outputs['dense_1'] tensor_info:
```

```
        dtype: DT_FLOAT
```

```
        shape: (-1, 10)
```

```
        name: StatefulPartitionedCall:0
```

```
Method name is: tensorflow/serving/predict
```

Alle SavedModels enthalten einen oder mehrere *Metagraphen*. Dabei handelt es sich um einen Rechengraphen plus die Definitionen von Funktionssignaturen (mit Namen, Typen und Formen von Ein- und Ausgaben). Jeder Metagraph wird durch einen Satz von Tags identifiziert. So kann es beispielsweise einen Metagraphen mit dem vollständigen Rechengraphen geben, der auch die Trainingsoperationen enthält (und der dann zum Beispiel mit "train" getaggt wird). Ein anderer Metagraph enthält einen beschnittenen Rechengraphen nur mit Operationen zum Vorhersagen, einschließlich einiger GPU-spezifischer Operationen (dieser Metagraph ist dann mit "serve" und "gpu" getaggt). Übergeben Sie ein tf.keras-Modell an die Funktion `tf.saved_model.save()`, wird standardmäßig ein viel einfacheres Saved-Model gesichert: Es enthält einen einzelnen Metagraphen mit dem Tag "serve", der zwei Signaturdefinitionen enthält – eine Initialisierungsfunktion (namens `__saved_model_init_op`, um die Sie sich keine Gedanken machen müssen) und eine Standard-Serving-Funktion (namens `serving_default`). Beim Speichern eines tf.keras-Modells entspricht die Standard-Serving-Funktion der Funktion `call()` des Modells, die natürlich Vorhersagen trifft.

Das Tool `saved_model_cli` kann auch genutzt werden, um Vorhersagen zu treffen (zum Testen, nicht für den produktiven Einsatz). Stellen Sie sich vor, Sie hätten ein NumPy-Array (`X_new`) mit drei Bildern handgeschriebener Ziffern, für die Sie Vorhersagen haben möchten. Zuerst müssen Sie sie in NumPys npy-Format exportieren:

```
np.save("my_mnist_tests.npy", X_new)
```

Dann nutzen Sie den Befehl `saved_model_cli` wie folgt:

```
$ saved_model_cli run --dir my_mnist_model/0001 --tag_set serve \
    --signature_def serving_default \
    --inputs flatten_input=my_mnist_tests.npy

[...] Result for output key dense_1:

[[1.1739199e-04 1.1239604e-07 6.0210604e-04 [...] 3.9471846e-04]
 [1.2294615e-03 2.9207937e-05 9.8599273e-01 [...] 1.1113169e-07]
 [6.4066830e-05 9.6359509e-01 9.0598064e-03 [...] 4.2495009e-04]]
```

Die Ausgabe des Tools enthält die zehn Kategorienwahrscheinlichkeiten für jede der drei Instanzen. Wunderbar! Nachdem Sie nun ein funktionierendes SavedModel haben, müssen Sie im nächsten Schritt TF Serving installieren.

TensorFlow Serving installieren

Es gibt viele Wege, TF Serving zu installieren: über ein Docker-Image³, mit dem Paketmanager des Systems, aus den Quelldateien und so weiter. Nutzen wir die Docker-Variante, die vom TensorFlow-Team wärmstens empfohlen wird, da sie sich leicht installieren lässt, Ihr System nicht durcheinanderbringt und eine hohe Performance bietet. Sie müssen zuerst Docker (<https://docker.com>) installieren. Dann laden Sie das offizielle Docker-Image von TF Serving herunter:

```
$ docker pull tensorflow/serving
```

Jetzt können Sie einen Docker-Container erstellen, um dieses Image auszuführen:

```
$ docker run -it --rm -p 8500:8500 -p 8501:8501 \
    -v "$ML_PATH/my_mnist_model:/models/my_mnist_model" \
    -e MODEL_NAME=my_mnist_model \
    tensorflow/serving

[...]
2019-06-01 [...] loaded servable version {name: my_mnist_model version: 1}
2019-06-01 [...] Running gRPC ModelServer at 0.0.0.0:8500 ...
2019-06-01 [...] Exporting HTTP/REST API at:localhost:8501 ...
[evhttp_server.cc : 237] RAW: Entering the event loop ...
```

Das ist alles, TF Server läuft schon! Es hat unser MNIST-Modell geladen (Version 1) und stellt

es sowohl über gRPC (an Port 8500) wie auch über REST (an Port 8501) bereit. Das bedeuten die Befehlszeilenoptionen:

-it

Macht den Container interaktiv (sodass Sie Strg-C drücken können, um ihn zu stoppen) und gibt die Ausgabe des Servers aus.

--rm

Löscht den Container, wenn Sie ihn stoppen (sodass sich Ihr Rechner nicht mit abgebrochenen Containern herumschlagen muss). Das Image wird dadurch aber nicht gelöscht.

-p 8500:8500

Lässt die Docker-Engine den TCP-Port 8500 des Hosts an den TCP-Port 8500 des Containers weiterleiten. Standardmäßig nutzt TF Serving diesen Port, um die gRPC-API zu bedienen.

-p 8501:8501

Leitet den TCP-Port 8501 des Hosts an den TCP-Port 8501 des Containers weiter. Standardmäßig nutzt TF Serving diesen Port, um die REST-API zu bedienen.

-v "\$ML_PATH/my_mnist_model:/models/my_mnist_model"

Macht das Verzeichnis \$ML_PATH/my_mnist_model des Hosts für den Container unter dem Pfad /models/mnist_model verfügbar. Unter Windows müssen Sie im Hostpfad die / eventuell durch \ ersetzen (aber nicht im Containerpfad).

-e MODEL_NAME=my_mnist_model

Setzt die Umgebungsvariable MODEL_NAME des Containers, sodass TF Serving weiß, welches Modell zu bedienen ist. Standardmäßig sucht es die Modelle im Verzeichnis /models und nutzt dort automatisch die neueste Version, die es findet.

tensorflow/serving

Der Name des auszuführenden Image.

Kehren wir jetzt zu Python zurück und fragen diesen Server ab, zuerst über die REST-API, dann über die gRPC-API.

TF Serving über die REST-API abfragen

Beginnen wir mit dem Erstellen der Abfrage. Sie muss den Namen der Funktionssignatur enthalten, die Sie aufrufen wollen, und natürlich die Eingabedaten:

```
import json

input_data_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

Beachten Sie, dass das JSON-Format vollständig textbasiert ist, daher muss das NumPy-Array `X_new` in eine Python-Liste konvertiert und dann als JSON formatiert werden:

```
>>> input_data_json  
'{"signature_name": "serving_default", "instances": [[[0.0, 0.0, 0.0, [...]  
0.3294117647058824, 0.725490196078431, [... sehr lang], 0.0, 0.0, 0.0, 0.0]]]}'
```

Jetzt senden wir die Eingabedaten an TF Serving, indem wir einen HTTP-POST-Request verschicken. Das lässt sich problemlos über die `requests`-Bibliothek erreichen (sie ist nicht Teil der Standardbibliothek von Python, daher müssen Sie sie zuerst installieren, zum Beispiel mit pip):

```
import requests  
  
SERVER_URL = 'http://localhost:8501/v1/models/my_mnist_model:predict'  
  
response = requests.post(SERVER_URL, data=input_data_json)  
  
response.raise_for_status() # bei einem Fehler eine Exception werfen  
  
response = response.json()
```

Der Response ist ein Dictionary mit einem einzelnen Schlüssel "predictions". Der entsprechende Wert ist die Liste mit Vorhersagen. Diese Liste ist eine Python-Liste, daher wollen wir sie in ein NumPy-Array umwandeln und die enthaltenen Gleitkommazahlen auf zwei Nachkommastellen runden:

```
>>> y_proba = np.array(response["predictions"])  
  
>>> y_proba.round(2)  
  
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],  
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],  
       [0. , 0.96, 0.01, 0. , 0. , 0. , 0. , 0.01, 0.01, 0. ]])
```

Hurra, wir haben die Vorhersagen! Das Modell ist sich fast zu 100% sicher, dass es sich beim ersten Bild um eine 7 handelt, zu 99%, dass das zweite Bild eine 2 zeigt, und zu 96%, dass das dritte Bild eine 1 enthält.

Die REST-API ist nett und einfach und funktioniert gut, wenn die Ein- und Ausgabedaten nicht zu groß sind. Und so gut wie jede Clientanwendung kann REST-Anfragen ohne zusätzliche Abhängigkeiten durchführen, während andere Protokolle nicht immer so direkt zur Verfügung stehen. Aber es basiert auf JSON, das textbasiert und ziemlich langatmig ist. Wir mussten zum Beispiel das NumPy-Array in eine Python-Liste umwandeln, und jede Gleitkommazahl wurde

als String dargestellt. Das ist sehr ineffizient, sowohl in Bezug auf die Serialisierungs-/Deserialisierungsdauer (um alle Gleitkommazahlen in Strings umzuwandeln und umgekehrt) als auch bezüglich der Payload-Größe: Viele Gleitkommazahlen werden mit mehr als 15 Zeichen dargestellt, was für 32-Bit-Floats über 120 Bits sind! Das führt zu einer hohen Latenz und entsprechendem Bandbreitenverbrauch beim Übertragen von großen NumPy-Arrays.⁴ Greifen wir daher lieber auf gRPC zurück.



Beim Übertragen großer Datenmengen ist es viel besser, die gRP-C API zu verwenden (wenn der Client sie unterstützt), da diese auf einem kompakten Binärformat und einem effizienten Kommunikationsprotokoll (auf Grundlage von HTTP/2 Framing) basiert.

TF Serving über die gRPC-API abfragen

Die gRPC-API erwartet einen serialisierten Protocol Buffer `PredictRequest` als Eingabe und gibt einen serialisierten Protocol Buffer `PredictResponse` als Ausgabe zurück. Diese Protobufs sind Teil der Bibliothek `tensorflow-serving-api`, die Sie installieren müssen (zum Beispiel mit pip). Erstellen wir zuerst den Request:

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()

request.model_spec.name = model_name

request.model_spec.signature_name = "serving_default"

input_name = model.input_names[0]

request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

Dieser Code erstellt einen Protocol Buffer `PredictRequest` und füllt ihn mit den erforderlichen Feldern, unter anderem mit dem Modellnamen (weiter oben definiert), dem Signaturnamen der aufzurufenden Funktion und schließlich mit den Eingabedaten in Form eines Tensor-Protobufs. Die Funktion `tf.make_tensor_proto()` erstellt einen Protocol Buffer Tensor, der auf dem angegebenen Tensor oder NumPy-Array basiert – in diesem Fall `X_new`.

Als Nächstes schicken wir den Request an den Server und erhalten seinen Response (dafür benötigen Sie die Bibliothek `grpcio`, die Sie über pip installieren können):

```
import grpc

from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')

predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)

response = predict_service.Predict(request, timeout=10.0)
```

Dieser Code ist recht klar: Nach den Importen erstellen wir einen gRPC-Kommunikationskanal nach *localhost* an TCP-Port 8500, dann erzeugen wir einen gRPC-Service über diesen Kanal und nutzen ihn, um einen Request mit einem Time-out von zehn Sekunden zu verschicken. (Beachten Sie, dass dieser Aufruf synchron geschieht: Er wird blockiert, bis er den Response erhält oder das Time-out abgelaufen ist.) In diesem Beispiel ist der Kanal unsicher (keine Verschlüsselung, keine Authentifizierung), aber gRPC und TensorFlow Serving unterstützen auch sichere Kanäle über SSL/TLS.

Nun konvertieren wir den Protocol Buffer `PredictResponse` in einen Tensor:

```
output_name = model.output_names[0]
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

Lassen Sie diesen Code laufen und geben `y_proba.numpy().round(2)` aus, werden Sie genau die gleichen geschätzten Kategorien wie zuvor erhalten. Und das ist auch schon alles: Mit ein paar Zeilen Code können Sie Ihr TensorFlow-Modell nun aus der Ferne abfragen – wahlweise mit REST oder gRPC.

Eine neue Modellversion deployen

Erstellen wir jetzt eine neue Modellversion und exportieren wir ein `SavedModel` wie zuvor, dieses Mal aber in das Verzeichnis `my_mnist_model/0002`:

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0002"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

In regelmäßigen Abständen (der Zeitraum ist konfigurierbar) prüft TensorFlow Serving auf neue Modellversionen. Findet es eine, geschieht der Übergang automatisch elegant: Standardmäßig wird es ausstehende Requests (sofern vorhanden) mit der vorherigen Modellversion beantworten, während es neue Requests mit der neuen Version verarbeitet.⁵ Sobald alle bestehenden Requests beantwortet sind, wird die vorherige Modellversion entladen. Sie können das in den Logs von TensorFlow Serving verfolgen:

```
[...]
```

```
reserved resources to load servable {name: my_mnist_model version: 2}

[...]

Reading SavedModel from: /models/my_mnist_model/0002

Reading meta graph with tags { serve }

Successfully loaded servable version {name: my_mnist_model version: 2}

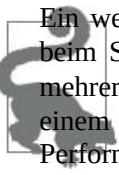
Quiescing servable version {name: my_mnist_model version: 1}

Done quiescing servable version {name: my_mnist_model version: 1}

Unloading servable version {name: my_mnist_model version: 1}
```

Dieser Ansatz ermöglicht einen sanften Übergang, verbraucht aber eventuell zu viel RAM (insbesondere GPU-RAM, was im Allgemeinen am stärksten beschränkt ist). In diesem Fall können Sie TF Serving so konfigurieren, dass es alle ausstehenden Requests mit der vorherigen Version verarbeitet und diese dann entlädt, bevor es die neue Modellversion lädt und einsetzt. Damit wird sichergestellt, dass zwei Modellversionen gleichzeitig geladen sind, aber der Service wird dann auch für eine kurze Zeit nicht zur Verfügung stehen.

Wie Sie sehen, macht es TF Serving ziemlich leicht, neue Modelle zu deployen. Und falls Sie feststellen, dass Version 2 nicht so gut funktioniert wie erwartet, ist es kein Problem, zu Version 1 zurückzukehren: Entfernen Sie einfach das Verzeichnis *my_mnist_model/0002*.



Ein weiteres tolles Feature von TF Serving ist seine automatische Batching-Fähigkeit, die Sie beim Starten mit der Option `--enable_batching` aktivieren können. Empfängt TF Serving mehrere Requests innerhalb einer kurzen Zeit (die konfigurierbar ist), wird es sie automatisch in einem Batch zusammenfassen, bevor es das Modell aufruft. Das ermöglicht eine deutliche Performanzsteigerung durch das Ausnutzen der GPU. Gibt das Modell die Vorhersagen zurück, nimmt TF Serving diese auseinander und verteilt sie an die richtigen Clients. Sie können also ein bisschen Latenz gegen einen höheren Durchsatz eintauschen, indem Sie die Batching-Verzögerung erhöhen (siehe die Option `--batching_parameters_file`).

Erwarten Sie viele Anfragen pro Sekunde, werden Sie TF Serving auf mehreren Servern deployen und die Anfragen per Load-Balancing verteilen wollen (siehe [Abbildung 19-2](#)). Das erfordert das Deployen und Managen vieler Container mit TF Serving auf diesen Servern. Eine Möglichkeit, das im Griff zu behalten, ist der Einsatz von Tools wie Kubernetes (<https://kubernetes.io>), bei dem es sich um ein Open-Source-System zum Vereinfachen der Container-Orchestrierung auf vielen Servern handelt. Wollen Sie die ganze Hardwareinfrastruktur nicht kaufen, warten und aktualisieren, können Sie virtuelle Maschinen auf einer Cloud-Plattform wie Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, Oracle Cloud oder einem anderen Platform-as-a-Service (PaaS) nutzen. Das Managen all der virtuellen Maschinen, die Container-Orchestrierung (selbst mit der Hilfe von Kubernetes), das Betreuen der Konfiguration von TF Serving, das Tunen und Monitoren – all das kann ein Vollzeitjob sein. Glücklicherweise gibt es Serviceprovider, die sich um all das

für Sie kümmern können. In diesem Kapitel werden wir die Google Cloud AI Platform verwenden, weil es aktuell die einzige Plattform mit TPUs ist, TensorFlow 2 unterstützt, einen netten Satz AI-Services anbietet (zum Beispiel AutoML, Vision API und Natural Language API) und diejenige ist, mit der ich am meisten Erfahrung habe. Aber es gibt auch noch viele andere Provider, wie zum Beispiel Amazon AWS SageMaker und die Microsoft AI Platform, die ebenso dazu in der Lage sind, TensorFlow-Modelle zu verarbeiten.

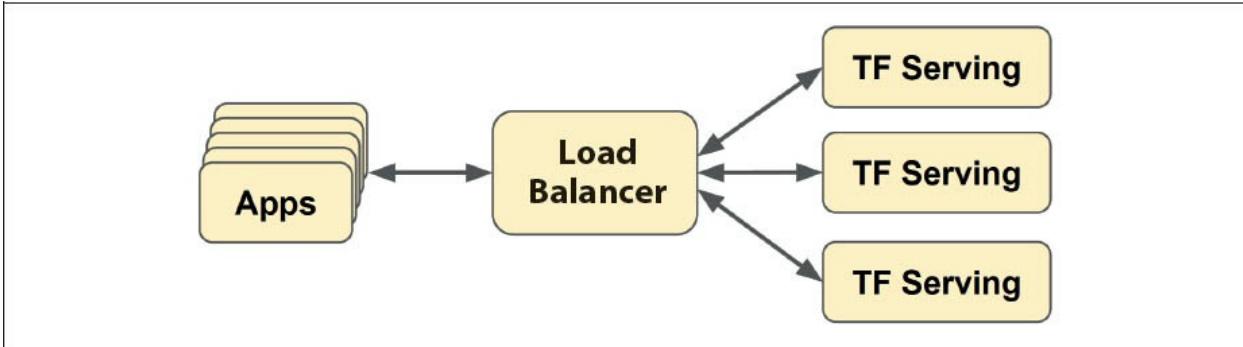


Abbildung 19-2: *TF Serving durch Load Balancing skalieren*

Schauen wir nun, wie wir unser wunderbares MNIST-Modell in die Cloud bringen können.

Einen Vorhersageservice auf der GCP AI Platform erstellen

Bevor Sie ein Modell deployen können, müssen wir uns ein bisschen um das Setup kümmern.

1. Melden Sie sich an Ihrem Google-Account an und gehen Sie dann zur Konsole der Google Cloud Platform (GCP) (<https://console.cloud.google.com/>) (siehe Abbildung 19-3). Haben Sie keinen Google-Account, erstellen Sie einen.
2. Nutzen Sie GCP das erste Mal, werden Sie die Bedingungen lesen und akzeptieren müssen. Wenn Sie möchten, können Sie auf *Tour Console* klicken. Aktuell wird neuen Benutzern eine Zeit zum freien Ausprobieren angeboten, darin ist ein GCP-Guthaben im Wert von 300 USD für die nächsten zwölf Monate enthalten. Sie brauchen nur sehr wenig davon, um für die Services zu bezahlen, die Sie in diesem Kapitel nutzen werden. Beim Anmelden für den Free Trial müssen Sie aber trotzdem ein Bezahlprofil anlegen und Ihre Kreditkartennummer angeben: Sie wird zur Verifikation genutzt (vermutlich um zu vermeiden, dass jemand den Free Trial mehrfach verwendet), aber nicht belastet. Aktivieren und upgraden Sie Ihren Account, wenn das erforderlich ist.

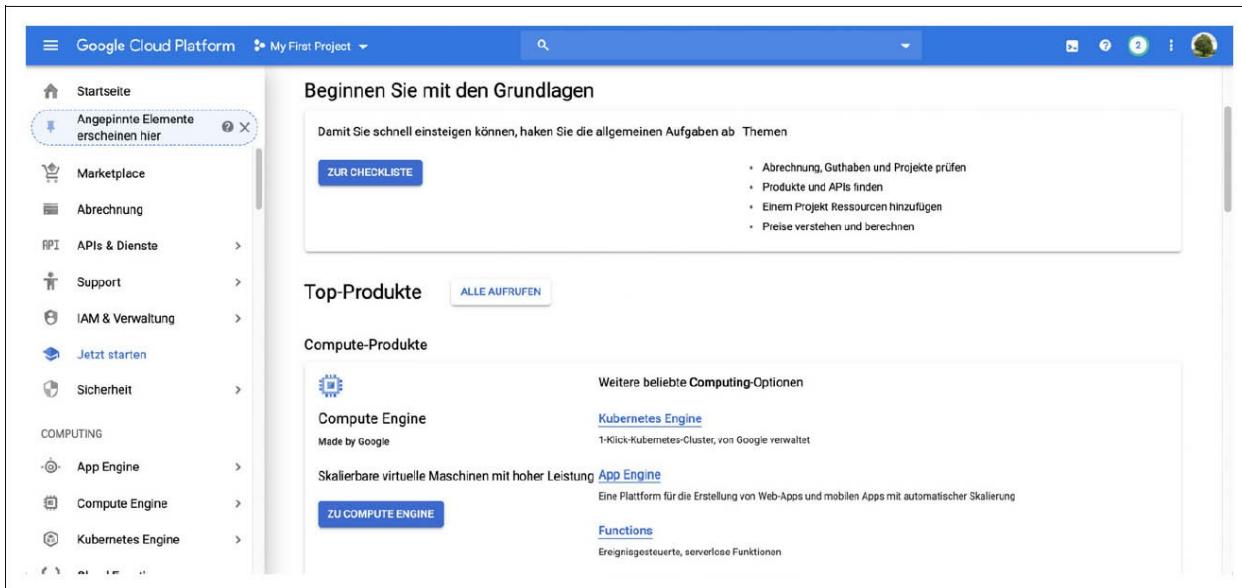


Abbildung 19-3: Konsole der Google Cloud Platform

3. Haben Sie die GCP schon zuvor verwendet und ist Ihre kostenlose Zeit abgelaufen, wird Sie der Service, den Sie in diesem Kapitel verwenden werden, etwas Geld kosten. Das sollte nicht allzu viel sein, insbesondere wenn Sie daran denken, ihn abzuschalten, wenn Sie ihn nicht mehr benötigen. Stellen Sie sicher, dass Sie die Bedingungen für die Kosten verstanden und ihnen zugestimmt haben, bevor Sie einen Service laufen lassen. Ich lehne hiermit jegliche Verantwortung ab, wenn Services nachher mehr kosten, als Sie erwartet haben! Sorgen Sie zudem dafür, dass Ihr Rechnungsaccount aktiv ist. Um das zu prüfen, öffnen Sie das Navigationsmenü auf der linken Seite, klicken auf »Abrechnung« und prüfen, ob Sie eine Bezahlmethode eingerichtet haben und ob der Account aktiv ist.
4. Jede Ressource in GCP gehört zu einem Projekt. Dazu gehören alle virtuellen Maschinen, die Sie vielleicht verwenden, die Dateien, die Sie speichern, und die Trainingsjobs, die Sie ausführen. Legen Sie einen Account an, erstellt GCP automatisch ein Projekt für Sie mit dem Namen »My First Project«. Wenn Sie wollen, können Sie den angezeigten Namen anpassen, indem Sie in die Projekteinstellungen wechseln: Im Navigationsmenü (links auf dem Bildschirm) wählen Sie »IAM & Verwaltung → Einstellungen«, ändern den Projektnamen und klicken auf »Speichern«. Beachten Sie, dass das Projekt auch eine eindeutige Projekt-ID und eine Nummer besitzt. Sie können die ID wählen, wenn Sie ein Projekt erstellen, sie später aber nicht mehr ändern. Wollen Sie ein neues Projekt erstellen, klicken Sie auf den Projektnamen oben auf der Seite, dann auf »Neues Projekt«, und dann geben Sie die Projekt-ID ein. Achten Sie darauf, dass die Abrechnung für dieses neue Projekt aktiv ist.



Sorgen Sie immer für eine Erinnerung, Services wieder abzuschalten, wenn Sie wissen, dass Sie sie nur für ein paar Stunden nutzen werden, denn ansonsten laufen sie eventuell Tage oder Monate, und das kann deutlich ins Geld gehen.

5. Nachdem Sie nun einen GCP-Account mit aktiver Abrechnung haben, können Sie die Services einsetzen. Der erste, den Sie brauchen, ist Google Cloud Storage (GCS): Hier werden Sie Ihre SavedModels, die Trainingsdaten und anderes ablegen. Scrollen Sie im Navigationsmenü zum Abschnitt »Storage« und klicken Sie dann auf »Storage → Browser«. Alle Ihre Dateien werden in einem oder mehreren *Buckets* untergebracht. Klicken Sie auf »Bucket erstellen« und wählen Sie den Namen des Buckets aus (eventuell müssen Sie die Storage-API erst aktivieren). GCS nutzt einen einzelnen, weltweiten Namensraum für Buckets, daher werden so einfache Namen wie »machine-learning« vermutlich nicht mehr verfügbar sein. Achten Sie darauf, dass der Bucketname den DNS-Namenskonventionen entspricht, da er eventuell in DNS Records zum Einsatz kommt. Zudem sind Bucketnamen öffentlich, daher sollten Sie hier nichts Privates erwähnen. Meist nutzt man seinen Domainnamen oder Firmennamen als Präfix, um eine Eindeutigkeit zu erreichen, oder Sie verwenden einfach eine Zufallszahl als Teil des Namens. Wählen Sie den Standort aus, an dem der Bucket gehostet werden soll – bei den restlichen Optionen sollten die Standardeinstellungen in Ordnung sein. Klicken Sie dann auf »Erstellen«.
6. Laden Sie den zuvor erstellten Ordner *my_mnist_model* (mit einer oder mehreren Versionen) in Ihren Bucket hoch. Dazu gehen Sie einfach zum GCS Browser, klicken auf den Bucket und ziehen dann den Ordner *my_mnist_model* aus Ihrem System in den Bucket (siehe Abbildung 19-4). Alternativ können Sie auf »Ordner hochladen« klicken und den Ordner *my_mnist_model* zum Hochladen auswählen. Standardmäßig beträgt die maximale Größe für ein SavedModel 250 MB, aber es ist möglich, eine höhere Quota anzufordern.

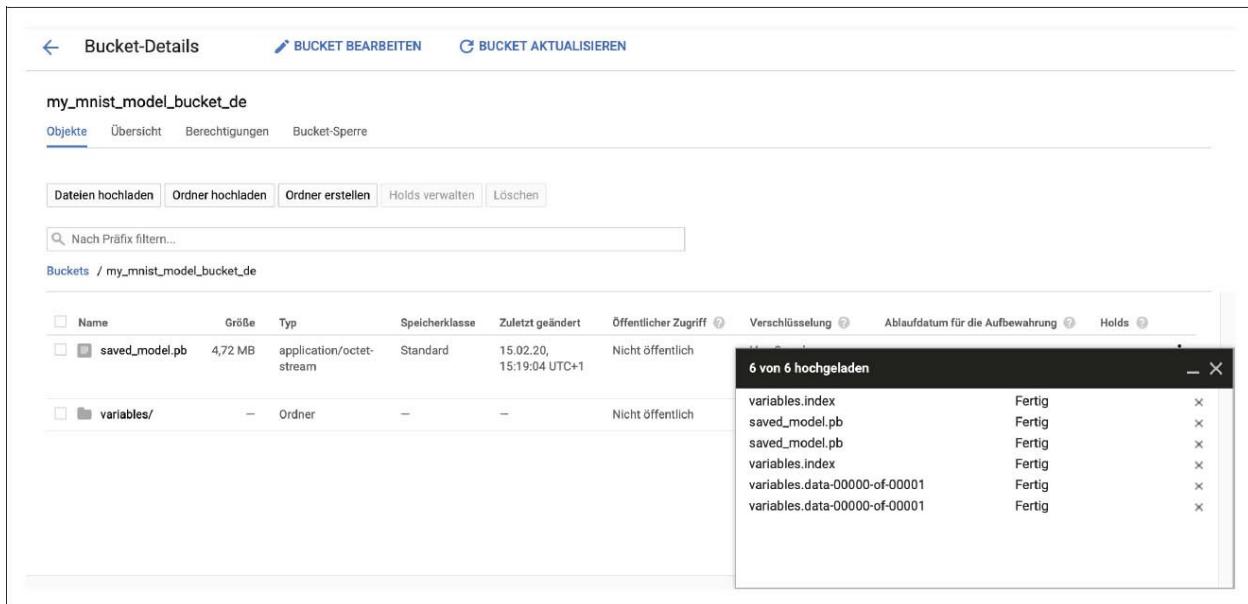


Abbildung 19-4: Ein SavedModel nach Google Cloud Storage hochladen

7. Jetzt müssen Sie AI Platform (ehemals ML Engine) so konfigurieren, dass sie weiß, welche Modelle und Versionen Sie verwenden wollen. Scrollen Sie im Navigationsmenü bis zum Abschnitt »Künstliche Intelligenz« und klicken Sie dort auf

»AI Platform → Modelle«. Klicken Sie auf »API aktivieren« (das dauert ein paar Minuten) und dann auf »Modell erstellen«. Tragen Sie die Details des Modells ein (siehe Abbildung 19-5) und klicken Sie auf »Erstellen«.

The screenshot shows the 'Modell erstellen' (Create Model) page in the Google Cloud AI Platform. On the left, there's a sidebar with icons for Dashboard, AI Hub, Daten-Labeling, Notebooks, Jobs, and Modelle. The 'Modelle' option is selected. The main area has a back arrow and the title 'Modell erstellen'. It contains fields for 'Modellname *' (set to 'my_mnist_model'), 'Region *' (set to 'europe-west1'), and 'Beschreibung' (set to 'My MNIST Model – Bilder mit handgeschriebenen Ziffern klassifizieren'). There are two checkboxes at the bottom: 'Logging für dieses Modell aktivieren' and 'Konsolen-Logging für dieses Modell aktivieren'. A note below the checkboxes states: 'Die Logging-Einstellungen gelten dauerhaft für dieses Modell. Für eine spätere Änderung Ihrer Logging-Einstellung müssen Sie ein neues Modell erstellen.' At the bottom are two buttons: 'ERSTELLEN' (highlighted in blue) and 'ABBRECHEN'.

Abbildung 19-5: Ein neues Modell auf der Google Cloud AI Platform erstellen

8. Nachdem Sie nun ein Modell auf der AI Platform haben, müssen Sie eine Modellversion erstellen. Klicken Sie in der Liste der Modelle auf das eben erstellte, dann auf »Neue Version« und geben Sie nun die Details dazu ein (siehe Abbildung 19-6): Name, Beschreibung, Python-Version (3.5 oder neuer), Framework (TensorFlow), Framework-Version (wenn möglich 2.0, ansonsten 1.13)⁶, ML-Laufzeitversion (wenn möglich 2.0, ansonsten 1.13), Machine Type (wählen Sie jetzt zunächst »Single-Core-CPU«), Modell-URI (dies ist der vollständige Pfad zum Ordner mit der Version, also zum Beispiel `gs://my_mnist_model_bucket_de/my_mnist_model/`), Skalierung (automatische Skalierung) und die Mindestanzahl von Knoten, die durchgehend laufen müssen (lassen Sie dieses Feld leer). Klicken Sie dann auf »Speichern«.

[!\[\]\(308522a65594114789db7890b9967c0a_img.jpg\) Version erstellen](#)

Wenn Sie eine neue Version Ihres Modells erstellen möchten, nehmen Sie vor dem Exportieren und Speichern des exportierten Modells in Cloud Storage die erforderlichen Anpassungen an der gespeicherten Modelldatei vor. [Weitere Informationen](#)

Name *
v0001

Namen können nicht geändert werden, müssen mit einem Buchstaben beginnen und dürfen nur Buchstaben, Ziffern und Unterstriche enthalten. Beachten Sie die Groß- und Kleinschreibung. 5/128

Beschreibung
Dense-Netz mit 2 Schichten (100, 10 Einheiten)

Python-Version *
3.5

Wählen Sie die Python-Version aus, mit der Sie das Modell trainiert haben

⚠️ Modellversionen mit Python 3.0 und höher können nicht für Batchvorhersagejobs verwendet werden. Die Onlinevorhersage funktioniert weiterhin.

Framework
TensorFlow

Abbildung 19-6: Eine neue Modellversion auf der Google Cloud AI Platform erstellen

Herzlichen Glückwunsch, Sie haben Ihr erstes Modell in die Cloud deployt! Weil Sie das automatische Skalieren gewählt haben, wird AI Platform mehr TF-Serving-Container starten, wenn die Anzahl der Queries pro Sekunde steigt, und die Abfragen zwischen ihnen per Load Balancing verteilen. Geht die QPS herunter, werden Container automatisch gestoppt. Die Kosten sind daher direkt mit der QPS verbunden (wie auch mit der gewählten Maschinenart und der Menge an Daten, die Sie auf GCS speichern). Dieses Preismodell ist insbesondere für gelegentliche Anwender und Services mit Lastspitzen nützlich, aber auch für Start-ups: Der Preis bleibt niedrig, bis das Start-up tatsächlich Gas gibt.



Nutzen Sie den Vorhersageservice nicht, wird AI Platform alle Container stoppen. Das bedeutet, Sie zahlen nur für den genutzten Storage (ein paar Cent pro Gigabyte im Monat). Fragen Sie den Service ab, muss AI Platform einen TF-Serving-Container starten, was ein paar Sekunden dauert. Ist diese Verzögerung inakzeptabel, werden Sie die minimale Anzahl an Containern beim Erstellen der Modellversion auf 1 setzen müssen. Das bedeutet natürlich, dass konstant immer mindestens eine Maschine läuft, was die monatlichen Kosten erhöht.

Fragen wir diesen Vorhersageservice nun ab!

Den Vorhersageservice verwenden

Unter der Motorhaube führt die AI Platform einfach TF Serving aus, daher könnten Sie im Prinzip den gleichen Code wie oben verwenden, wenn Sie wissen, welche URL Sie abfragen müssten. Es gibt nur ein Problem: GCP kümmert sich auch um die Verschlüsselung und die Authentifizierung. Die Verschlüsselung basiert auf SSL/TLS, die Authentifizierung ist tokenbasiert: Es muss mit jedem Request ein geheimes Authentifizierungstoken an den Server geschickt werden. Bevor Sie also den Vorhersageservice (oder einen anderen GCP-Service) verwenden können, brauchen Sie ein Token. Wir werden gleich sehen, wir wir das bekommen, aber zuerst müssen Sie die Authentifizierung konfigurieren und Ihrer Anwendung die passenden Zugriffsberechtigungen auf GCP geben. Sie haben für die Authentifizierung zwei Optionen:

- Ihre Anwendung (also der Clientcode, der den Vorhersageservice abfragen wird) kann sich über die Benutzer-Credentials Ihres eigenen Google-Log-ins und Passworts authentifizieren. Damit würde sich Ihre Anwendung die gleichen Rechte wie Sie selbst auf GCP verschaffen, was sicherlich viel mehr ist, als sie braucht. Zudem würden Sie Ihre Credentials in Ihrer Anwendung verteilen müssen, sodass jeder mit Zugriff Ihre Credentials stehlen könnte und damit vollständigen Zugriff auf Ihren GCP-Account erhalten würde. Kurz gesagt – wählen Sie nicht diese Option, sie ist nur in sehr seltenen Fällen notwendig (zum Beispiel wenn Ihre Anwendung direkt auf Ihren GCP-Account zugreifen muss).

Der Clientcode kann sich mit einem *Dienstkonto* authentifizieren. Dabei handelt es sich um einen Account, der keinen Benutzer, sondern eine Anwendung repräsentiert. Er erhält meist nur sehr eingeschränkte Zugriffsberechtigungen – nur das, was er braucht, aber nicht mehr. Das ist die empfohlene Option.

Erstellen wir also einen Serviceaccount für Ihre Anwendung. Gehen Sie dazu im Navigationsmenü zu »IAM & Verwaltung → Dienstkonten«, klicken Sie dann auf »Dienstkonto erstellen«, füllen Sie das Formular aus (Name des Dienstkontos, Dienstkonto-ID, Beschreibung) und klicken Sie auf »Erstellen« (siehe Abbildung 19-7). Als Nächstes müssen Sie Ihrem Dienstkonto Zugriffsberechtigungen erteilen. Wählen Sie die Rolle »ML Engine-Entwickler« – damit kann das Dienstkonto Vorhersagen erstellen, aber nicht viel mehr. Optional können Sie Benutzerzugriff auf das Dienstkonto erteilen (das ist nützlich, wenn Ihr GCP-Benutzer Teil einer Organisation ist und Sie möchten, dass andere Benutzer in der Organisation Anwendungen deployen dürfen, die auf diesem Dienstkonto basieren oder die das Konto selbst managen können sollen). Anschließend klicken Sie auf »Schlüssel erstellen«, um den privaten Schlüssel des Dienstkontos zu exportieren, wählen »JSON« und klicken auf »Erstellen«. Damit wird der private Schlüssel in Form einer JSON-Datei heruntergeladen. Sorgen Sie dafür, dass er auch privat bleibt!

Dienstkonto erstellen

- 1 Dienstkontodetails — 2 Diesem Dienstkonto Zugriff auf das Projekt erteilen (optional) —
- 3 Nutzern Zugriff auf dieses Dienstkonto erteilen (optional)

Dienstkontodetails

Name des Dienstkontos
my_software

Anzeigename für dieses Dienstkonto

Dienstkonto-ID
my-software @advance-airline-268313.iam.gserviceaccount.com X C

Beschreibung des Dienstkontos
Das ist meine Software, die auf dem Vorhersageservice basiert.

Beschreiben Sie den Zweck des Dienstkontos

ERSTELLEN ABBRECHEN

Abbildung 19-7: Ein neues Dienstkonto in Google IAM anlegen

Wunderbar. Jetzt wollen wir ein kleines Skript schreiben, das den Vorhersageservice abfragt. Google stellt eine Reihe von Bibliotheken bereit, mit denen der Zugriff auf seine Services vereinfacht wird:

Google API Client Library

Dies ist eine ziemlich dünne Schicht, die auf OAuth 2.0 (<https://oauth.net>) (zur Authentifizierung) und REST aufsetzt. Sie können sie mit allen GCP-Services verwenden, einschließlich AI Platform. Installieren Sie sie über pip: Die Bibliothek heißt `google-api-python-client`.

Google Cloud Client Libraries

Diese Bibliotheken arbeiten auf einem etwas höheren Level: Jede ist auf einen bestimmten Service ausgerichtet, wie zum Beispiel GCS, Google BigQuery, Google Cloud Natural Language oder Google Cloud Vision. Alle diese Bibliotheken können mithilfe von pip installiert werden (die GCS Client Library heißt zum Beispiel `google-cloud-storage`). Steht eine Clientbibliothek für einen Service zur Verfügung, ist es sinnvoll, diese auch statt der Google API Client Library zu verwenden, da sie die Best Practices implementiert und oft auf gRPC statt auf REST zurückgreift, was für eine bessere Performance sorgt.

Aktuell gibt es keine Clientbibliothek für AI Platform, daher werden wir die Google API Client Library verwenden. Sie muss den privaten Schlüssel des Dienstkontos verwenden – teilen Sie ihn mit, wo sie ihn findet, indem Sie die Umgebungsvariable

`GOOGLE_APPLICATION_CREDENTIALS` setzen. Entweder tun Sie das vor dem Starten des Skripts oder innerhalb des Skripts, zum Beispiel:

```
import os

os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "my_service_account_key.json"
```

 Deployen Sie Ihre Anwendung auf eine virtuelle Maschine auf der Google Cloud Engine (GCE), innerhalb eines Containers mit der Google Cloud Kubernetes Engine, als Webanwendung in der Google Cloud App Engine oder als Microservice auf Google Cloud Functions und ist die Umgebungsvariable `GOOGLE_APPLICATION_CREDENTIALS` nicht gesetzt, wird die Bibliothek das Standarddienstkontakt für den Hostservice nutzen (zum Beispiel das Standard-GCE-Dienstkontakt, wenn Ihre Anwendung auf GCE läuft).

Als Nächstes müssen Sie ein Ressourcenobjekt erstellen, das den Zugriff auf den Vorhersageservice verpackt:⁷

```
import googleapiclient.discovery

project_id = "onyx-smoke-242003" # nutzen Sie hier Ihre Projekt-ID

model_id = "my_mnist_model"

model_path = "projects/{}/models/{}".format(project_id, model_id)

ml_resource = googleapiclient.discovery.build("ml", "v1").projects()
```

Beachten Sie, dass Sie `/versions/0001` (oder eine andere Versionsnummer) an den `model_path` anhängen können, um die Version festzulegen, die Sie verwenden wollen. Das kann für A/B-Tests oder für das Testen neuer Versionen mit einer kleinen Gruppe von Benutzern nützlich sein, bevor eine Version ganz in die freie Wildbahn entlassen wird (das nennt sich *Canary Testing*). Als Nächstes schreiben wir eine kleine Funktion, die das Ressourcenobjekt nutzen wird, um den Vorhersageservice aufzurufen und die Vorhersagen zurückzuerhalten:

```
def predict(X):

    input_data_json = {"signature_name": "serving_default",
                      "instances": X.tolist()}

    request = ml_resource.predict(name=model_path, body=input_data_json)
    response = request.execute()

    if "error" in response:
        raise RuntimeError(response["error"])
```

```
return np.array([pred[output_name] for pred in response["predictions"]])
```

Die Funktion übernimmt ein NumPy-Array mit den Eingabebildern und bereitet ein Dictionary vor, das die Clientbibliothek in das JSON-Format konvertieren wird (wie wir das schon zuvor getan haben). Dann bereitet sie einen Vorhersage-Request vor und führt ihn aus – sie wirft eine Exception, wenn der Response einen Fehler enthält, ansonsten extrahiert sie die Vorhersagen für jede Instanz und fasst sie in einem NumPy-Array zusammen. Schauen wir, ob es funktioniert:

```
>>> Y_probas = predict(X_new)

>>> np.round(Y_probas, 2)

array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.96, 0.01, 0. , 0. , 0. , 0. , 0.01, 0.01, 0. ]])
```

Ja! Sie haben jetzt einen netten Vorhersageservice, der in der Cloud läuft und der automatisch als Reaktion auf beliebig viele QPS skaliert. Zudem kostet er Sie so gut wie nichts, wenn Sie ihn nicht verwenden – Sie zahlen nur ein paar Cent pro Monat pro Gigabyte, das Sie mit GCS nutzen. Und Sie können über Google Stackdriver (<https://cloud.google.com/stackdriver/>) detaillierte Logs und Metriken erhalten.

Aber wie sieht es aus, wenn Sie Ihr Modell in eine mobile App deployen wollen? Oder auf ein Embedded Device?

Ein Modell auf ein Mobile oder Embedded Device deployen

Müssen Sie Ihr Modell auf ein mobiles Gerät oder ein Embedded Device deployen, kann es sein, dass ein großes Modell einfach zu lange zum Herunterladen braucht und zu viel RAM und CPU benötigt, was Ihre App wenig responsiv sein lässt, das Gerät aufheizt und den Akku leer saugt. Um das zu vermeiden, müssen Sie ein mobiltaugliches, leichtgewichtiges und effizientes Modell erstellen, ohne zu viel von seiner Genauigkeit zu opfern. Die TFLite-Bibliothek (<https://tensorflow.org/lite>) enthält eine Reihe von Tools⁸, die Ihnen dabei helfen können, Ihre Modelle auf mobile Geräte und Embedded Devices zu deployen, und die drei Hauptziele verfolgen:

- Verringern der Modellgröße, um Ladezeiten und RAM-Einsatz zu reduzieren.
- Verringern des Rechenaufwands für jede Vorhersage, um Latenz, Akkuverbrauch und Erwärmung zu reduzieren.
- Anpassen des Modells an gerätespezifische Einschränkungen.

Um die Modellgröße zu verringern, kann der Modellkonverter von TFLite ein SavedModel nehmen und es in ein viel schlankeres Format komprimieren, das auf FlatBuffers (<https://google.github.io/flatbuffers/>) basiert. Dabei handelt es sich um eine effiziente und plattformübergreifende Serialisierungsbibliothek (ein bisschen wie Protocol Buffer), die

ursprünglich von Google für das Gaming entwickelt wurde. Sie ist so entworfen, dass Sie FlatBuffers direkt in den Speicher laden können, ohne sie noch vorverarbeiten zu müssen. Damit verringern sich die Ladedauer und der Speicherbedarf. Ist das Modell auf einem mobilen Gerät oder in ein Embedded Device geladen, führt der TFLite-Interpreter es aus, um die Vorhersagen zu treffen. So können Sie ein SavedModel in einen FlatBuffer umwandeln und es in eine `.tflite`-Datei abspeichern:

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_path)

tflite_model = converter.convert()

with open("converted_model.tflite", "wb") as f:

    f.write(tflite_model)
```



Sie können auch ein `tf.keras`-Modell mit `from_keras_model()` direkt in einen FlatBuffer speichern.

Der Konverter optimiert das Modell zudem, indem er es kleiner macht und seine Latenz verringert. Er schneidet alle Operationen weg, die für das Treffen von Vorhersagen nicht notwendig sind (wie zum Beispiel Trainingsoperationen), und optimiert wo immer möglich Berechnungen – beispielsweise wird $3a + 4a + 5a$ nach $(3 + 4 + 5)a$ konvertiert. Er versucht auch, Operationen möglichst zu verschmelzen. So werden zum Beispiel Batchnormalisierungsschichten mit den Additions- und Multiplikationsoperationen der vorherigen Schicht verbunden, wenn das machbar ist. Um eine Idee davon zu bekommen, wie weit TFLite ein Modell optimieren kann, laden Sie eines der vortrainierten TFLite-Modelle (<https://homl.info/litemodels>) herunter, entpacken das Archiv, öffnen das ausgezeichnete Netron Graph Visualization Tool (<https://lutzroeder.github.io/netron/>) und laden die `.pb`-Datei dort hoch, um das ursprüngliche Modell anzuzeigen. Das ist ein großer, komplexer Graph, oder? Nun öffnen Sie das optimierte `.tflite`-Modell und staunen über seine Schönheit!

Eine andere Möglichkeit, die Modellgröße zu reduzieren (neben dem Einsatz kleinerer Netzwerkarchitekturen) ist das Verwenden kleinerer Bitbreiten: Nutzen Sie beispielsweise Gleitkommazahlen mit 16 Bit Breite statt mit der normalen Breite von 32 Bit, verkleinert sich das Modell um einen Faktor von zwei, allerdings auf Kosten einer (im Allgemeinen nur wenig) sinkenden Genauigkeit. Zudem wird das Training schneller sein, und Sie werden nur ungefähr die Hälfte des GPU-RAM benötigen.

Der TFLite-Konverter kann sogar noch weitergehen und die Modellgewichte auf 8-Bit-Ganzzahlen herunterquantisieren! Das führt im Vergleich zu 32-Bit-Gleitkommazahlen zu einer Reduktion auf ein Viertel. Der einfachste Ansatz dafür nennt sich *Post-Training-Quantisierung*: Die Gewichte werden nach dem Training einfach quantisiert, wobei eine ziemlich einfache, aber effiziente symmetrische Quantisierungstechnik zum Einsatz kommt. Sie findet den maximalen absoluten Gewichtswert m und bildet dann den Gleitkommabereich $-m$ bis $+m$ auf den

ganzzahligen Bereich von –127 bis +127 ab. Liegen die Gewichte beispielsweise im Bereich von –1,5 bis +0,8 (siehe Abbildung 19-8), entsprechen die Bytes –127, 0 und +127 den Gleitkommazahlen –1,5, 0 und +1,5. Beachten Sie, dass 0,0 bei symmetrischer Quantisierung immer auf 0 abgebildet wird (und dass der Bytebereich +68 bis +127 in diesem Beispiel nicht verwendet wird, da er für die Gleitkommazahlen größer als +0,8 genutzt würde).

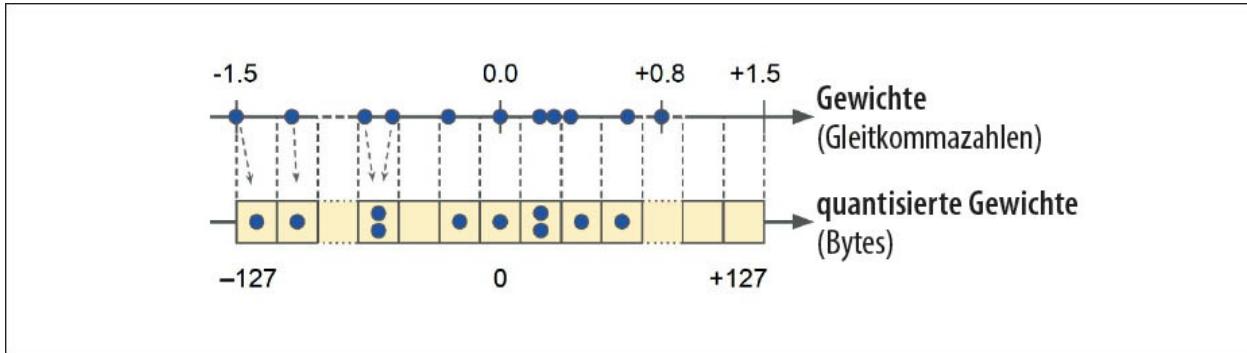


Abbildung 19-8: Von 32-Bit-Gleitkommazahlen zu 8-Bit-Ganzzahlen mit symmetrischer Quantisierung

Um diese Post-Training-Quantisierung durchzuführen, fügen Sie einfach `OPTIMIZE_FOR_SIZE` zur Liste der Konverteroptimierungen hinzu, bevor Sie die Methode `convert()` aufrufen:

```
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

Diese Technik verringert die Größe des Modells sehr deutlich, daher lässt es sich so viel schneller herunterladen und abspeichern. Aber zur Laufzeit werden diese quantisierten Gewichte wieder in Gleitkommazahlen umgewandelt, bevor sie verwendet werden (diese wiederhergestellten Gleitkommazahlen entsprechen nicht perfekt den ursprünglichen Werten, sind aber auch nicht zu weit weg von ihnen, daher ist der Genauigkeitsverlust im Allgemeinen durchaus akzeptabel). Um zu vermeiden, dass sie immer wieder neu errechnet werden müssen, werden die wiederhergestellten Gleitkommazahlen gecacht, daher ist der RAM-Bedarf gleich groß. Und auch die Rechenzeit ist nicht kürzer.

Am effektivsten reduzieren Sie Latenz und Strombedarf, indem Sie auch die Aktivierungen quantisieren, sodass die Berechnungen vollständig mit Ganzzahlen durchgeführt werden können und Gleitkommaoperationen nicht erforderlich sind. Selbst beim Einsatz der gleichen Bitbreite (zum Beispiel 32-Bit-Integer statt 32-Bit-Floats) werden für ganzzahlige Berechnungen weniger CPU-Aufrufe und weniger Energie gebraucht, und es wird weniger Wärme erzeugt. Verringern Sie dann noch die Bitbreite (zum Beispiel auf 8-Bit-Integer), läuft das Ganze deutlich schneller. Zudem können manche Devices zum Beschleunigen von neuronalen Netzen (beispielsweise die Edge TPU) nur Ganzzahlen verarbeiten, daher ist eine vollständige Quantisierung von Gewichten und Aktivierungen in diesem Fall zwingend erforderlich. Das kann nach dem Training geschehen – es erfordert einen Kalibrierungsschritt, um den maximalen absoluten Wert der Aktivierungen zu finden. Daher müssen Sie ein repräsentatives Sample mit Trainingsdaten an TFLite weitergeben (es muss nicht groß sein), damit dieses die Daten mit dem Modell verarbeiten und die Aktivierungsstatistiken zu Quantisierungszwecken messen kann (dieser

Schritt ist meist schnell).

Das Hauptproblem der Quantisierung ist, dass ein bisschen Genauigkeit verloren geht: Es entspricht dem Hinzufügen von Rauschen zu den Gewichten und Aktivierungen. Ist der Genauigkeitsverlust zu groß, müssen Sie eventuell ein *quantisierungsbewusstes Training* verwenden. Dabei werden künstliche Quantisierungsoperationen zum Modell hinzugefügt, sodass es lernen kann, das Quantisierungsrauschen während des Trainings zu ignorieren – die so entstandenen Gewichte sind dann robuster gegenüber der Quantisierung. Zudem kann der Kalibrierungsschritt schon automatisch während des Trainings geschehen, was den gesamten Prozess vereinfacht.

Ich habe die zentralen Konzepte von TFLite erläutert, aber der Weg zu einer mobilen Anwendung oder einem Embedded Program kann ein ganzes Buch füllen. Zum Glück gibt es so eins: Wollen Sie mehr darüber lernen, wie Sie TensorFlow-Anwendungen für Mobile und Embedded Devices bauen, schauen Sie sich das O'Reilly-Buch *TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers* (<https://homl.info/tinyml>) von Pete Warden (der das TFLite-Team leitet) und Daniel Situnayake an.

TensorFlow im Browser

Was sollten Sie tun, wenn Sie Ihr Modell auf einer Website laufen lassen wollen – direkt im Browser des Anwenders? Es gibt dafür einige sinnvolle Szenarien, unter anderem:

- Wenn Ihre Webanwendung häufig in Situationen genutzt wird, in denen der Internetzugang des Anwenders wackelig oder langsam ist (zum Beispiel eine Website für Wanderer), sodass das Ausführen des Modells direkt im Client die einzige Möglichkeit ist, Ihre Website zuverlässig laufen zu lassen.
- Wenn Sie die Antwort des Modells so schnell wie möglich brauchen (zum Beispiel in einem Onlinespiel). Indem Sie nicht erst den Server anfragen müssen, um eine Vorhersage zu erhalten, verringern Sie die Latenz definitiv, und die Website wird schneller reagieren.
- Erstellt Ihr Webservice Vorhersagen, die auf privaten Daten der Benutzer basieren, und wollen Sie die Privatsphäre der Anwender wahren, können Sie die Vorhersagen direkt auf Clientseite erstellen, sodass diese Daten niemals den Rechner des Anwenders verlassen müssen.⁹

Für all diese Szenarien können Sie Ihr Modell in einem speziellen Format exportieren, das von der TensorFlow.js-JavaScript-Bibliothek (<https://tensorflow.org/js>) geladen werden kann. Diese Bibliothek kann dann Ihr Modell nutzen, um Vorhersagen direkt im Browser des Anwenders zu machen. Das TensorFlow.js-Projekt enthält ein Tool `tensorflowjs_converter`, das ein TensorFlow-SavedModel oder eine Keras-Modelldatei in das Format *TensorFlow.js Layers* konvertieren kann: Dabei handelt es sich um ein Verzeichnis mit einem Satz gemeinsam genutzter Gewichtsdateien im Binärformat und einer Datei `model.json`, die die Architektur des Modells beschreibt und Verweise auf die Gewichtsdateien enthält. Dieses Format ist darauf optimiert, effizient im Web

heruntergeladen werden zu können. Anwender können dann mithilfe der TensorFlow.js-Bibliothek das Modell herunterladen und Vorhersagen im Browser erstellen lassen. Dieser Codeschnipsel vermittelt Ihnen eine Idee davon, wie die JavaScript-API aussieht:

```
import * as tf from '@tensorflow/tfjs';

const model = await
tf.loadLayersModel('https://example.com/tfjs/model.json');

const image = tf.fromPixels(webcamElement);

const prediction = model.predict(image);
```

Auch dieses Thema lässt sich nur mit einem ganzen Buch ausreichend umfassend behandeln. Wollen Sie mehr darüber wissen, schauen Sie sich das O'Reilly-Buch *Practical Deep Learning for Cloud, Mobile, and Edge* (<https://homl.info/tfjsbook>) von Anirudh Koul, Siddha Ganju und Meher Kasam an.

Als Nächstes werden wir uns damit beschäftigen, wie wir die Berechnungen mit GPUs beschleunigen können!

Mit GPUs die Berechnungen beschleunigen

In Kapitel 11 haben wir mehrere Techniken erwähnt, die das Trainieren erheblich beschleunigen können: bessere Initialisierung der Gewichte, Batchnormalisierung, 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 Abschnitt werden wir uns anschauen, wie wir unsere Modelle mithilfe von GPUs beschleunigen können. Auch werden Sie erfahren, wie Sie die Berechnungen auf mehrere Devices verteilen können, unter anderem auf die CPU und mehrere GPU-Devices (siehe Abbildung 19-9). Zunächst werden wir alles auf einer einzelnen Maschine laufen lassen, aber später werden wir noch besprechen, wie wir die Berechnungen auf mehrere Server verteilen können.

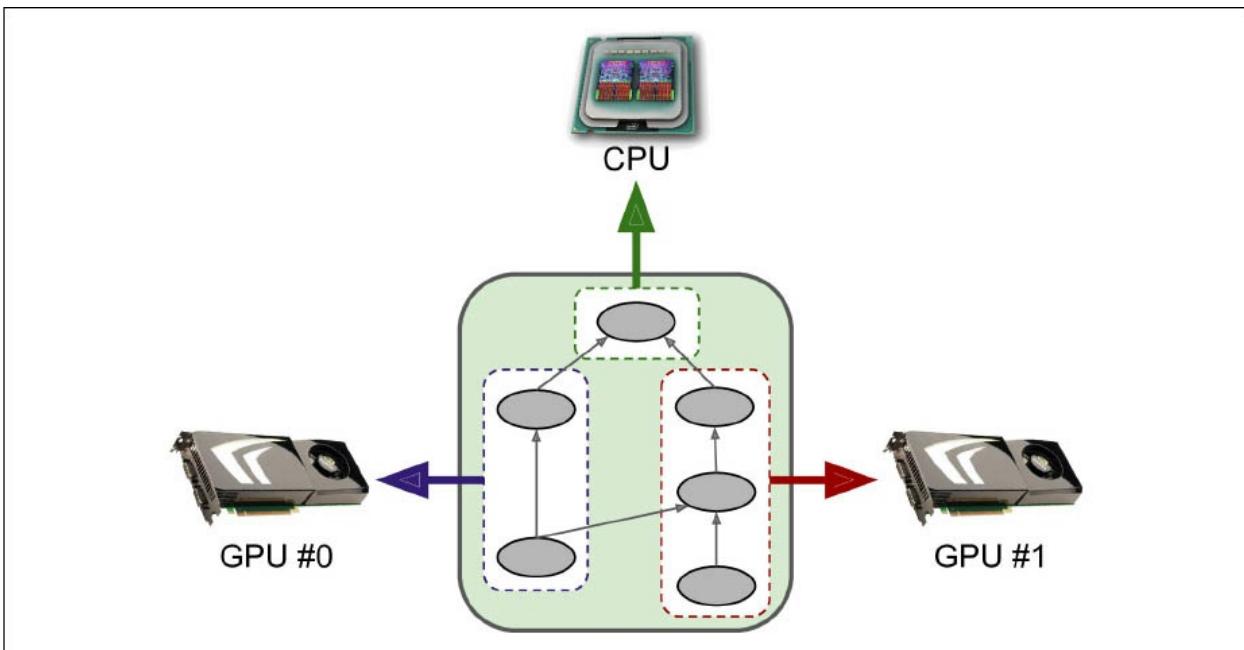


Abbildung 19-9: Paralleles Ausführen eines TensorFlow-Graphen auf mehreren Geräten

Dank der GPUs müssen wir nicht mehr Tage oder Wochen darauf warten, dass ein Trainingsalgorithmus fertig wird, sondern nur ein paar Minuten oder Stunden. Das spart nicht nur unglaublich viel Zeit, es bedeutet auch, dass Sie einfacher mit verschiedenen Modellen experimentieren und Ihre Modelle häufiger mit frischen Daten neu trainieren können.

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 vier GPUs auf demselben Computer etwa genauso schnell trainieren wie acht GPUs auf mehreren Computern (aufgrund der zusätzlichen Verzögerung durch das Netzwerk bei mehreren Computern). Genauso ist es oft sinnvoller, eine einzelne leistungsfähige GPU statt mehrerer langsamerer GPUs zu verwenden.

In einem ersten Schritt müssen Sie überhaupt an eine GPU kommen. Dafür gibt es zwei Möglichkeiten: Entweder kaufen Sie Ihre eigenen, oder Sie verwenden mit GPUs ausgestattete virtuelle Maschinen in der Cloud. Beginnen wir mit der ersten Option.

Sich eine eigene GPU zulegen

Haben Sie sich dazu entschieden, eine GPU-Karte zu kaufen, nehmen Sie sich etwas Zeit, um die richtige Wahl zu treffen. Tim Dettmers hat einen ausgezeichneten Blogartikel (<https://homl.info/66>) dazu geschrieben und aktualisiert ihn regelmäßig. Ich kann Ihnen nur raten, ihn sorgfältig zu lesen. Aktuell unterstützt TensorFlow nur Nvidia-Karten mit CUDA Compute Capability 3.5+ (<https://homl.info/cudagpus>) (und natürlich Googles TPUs), aber es erweitert seine Unterstützung eventuell auch auf andere Hersteller. Zudem sind TPUs aktuell nur auf GCP verfügbar, aber es ist sehr wahrscheinlich, dass TPU-ähnliche Karten in naher Zukunft auch so zu kaufen sein werden und dass TensorFlow sie unterstützt. Kurz gesagt: Schauen Sie in der Dokumentation von TensorFlow (<https://tensorflow.org/install>) nach, um herauszufinden, welche Devices dann unterstützt werden.

Entscheiden Sie sich für eine Nvidia-GPU-Karte, müssen Sie die passenden Nvidia-Treiber und diverse Nvidia-Bibliotheken installieren.¹⁰ Dazu gehören die CUDA-Bibliothek *Compute Unified Device Architecture*, mit der Entwickler CUDA-unterstützte GPUs für alle möglichen Berechnungen einsetzen können (nicht nur zur Grafikbeschleunigung), und die cuDNN-Bibliothek *CUDA Deep Neural Network*, bei der es sich um eine GPU-beschleunigte Bibliothek mit Primitiven für DNNs handelt. cuDNN stellt optimierte Implementierungen häufig genutzter DNN-Berechnungen bereit, wie zum Beispiel Aktivierungsschichten, Normalisierung, Forward- und Backward-Convolutions und Pooling (siehe [Kapitel 14](#)). Sie ist Teil von Nvidias Deep Learning SDK (beachten Sie, dass Sie einen Developer-Account bei Nvidia anlegen müssen, um sie herunterladen zu können). TensorFlow nutzt CUDA und cuDNN, um die GPU-Karten zu steuern und Berechnungen zu beschleunigen (siehe [Abbildung 19-10](#)).

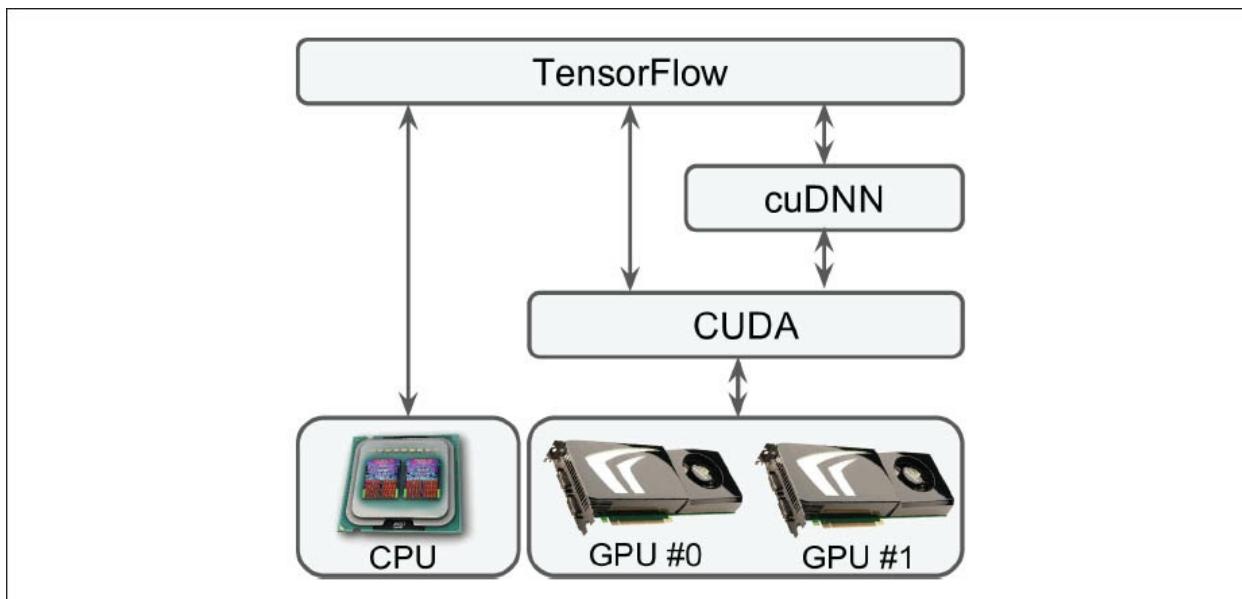


Abbildung 19-10: TensorFlows Verwendung von CUDA und cuDNN zum Ansteuern von GPUs und zum Beschleunigen von DNNs

Haben Sie die GPU-Karte(n) und die erforderlichen Treiber und Bibliotheken installiert, können Sie mit dem Befehl `nvidia-smi` prüfen, ob CUDA korrekt installiert ist. Der Befehl führt die verfügbaren GPU-Karten und die Prozesse auf jeder Karte auf:

```
$ nvidia-smi

Sun Jun  2 10:05:22 2019

+-----+
| NVIDIA-SMI 418.67      Driver Version: 410.79      CUDA Version: 10.0      |
|-----+
| GPU Name      Persistence-M| Bus-Id      Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|=====+=====+=====+=====+=====+=====+=====+=====|
| 0 Tesla T4        Off  | 00000000:00:04.0 Off |                  0 |
| N/A   61C     P8    17W /  70W |      0MiB / 15079MiB |      0%     Default |
|-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| Processes:                               GPU Memory |
| GPU  PID  Type  Process name          Usage  |
|=====+=====+=====+=====+=====+=====|
| No running processes found               |
+-----+
```

Aktuell müssen Sie auch noch die GPU-Version von TensorFlow installieren (also die Bibliothek `tensorflow-gpu`), aber es wird daran gearbeitet, eine einheitliche Installationsprozedur für Nur-CPU- und GPU-Rechner anzubieten. Lesen Sie also die Installationsdokumentation, um zu entscheiden, welche Bibliothek Sie installieren sollten. Auf jeden Fall ist das saubere Installieren jeder erforderlichen Bibliothek ein bisschen langwierig und knifflig (und es wird wirklich unerfreulich, wenn Sie die falschen Versionen installieren), daher bietet TensorFlow ein Docker-Image an, in dem alles enthalten ist, was Sie benötigen. Damit der Docker-Container allerdings Zugriff auf die GPU erhält, müssen Sie auf der Hostmaschine trotzdem die Nvidia-Treiber installieren.

Um zu prüfen, ob TensorFlow die GPUs wirklich sieht, führen Sie folgende Tests durch:

```
>>> import tensorflow as tf  
  
>>> tf.test.is_gpu_available()  
  
True  
  
>>> tf.test.gpu_device_name()  
  
'/device:GPU:0'  
  
>>> tf.config.experimental.list_physical_devices(device_type='GPU')  
  
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Die Funktion `is_gpu_available()` prüft, ob mindestens eine GPU verfügbar ist. Die Funktion `gpu_device_name()` liefert den Namen der ersten GPU – standardmäßig laufen die Operationen darauf. Die Funktion `list_physical_devices()` liefert die Liste aller verfügbaren GPU-Devices zurück (in diesem Beispiel nur eines).¹¹

Was können Sie aber tun, wenn Sie kein Geld und keine Zeit in das Beschaffen Ihrer eigenen GPU-Karte investieren wollen? Verwenden Sie einfach eine GPU-VM in der Cloud!

Eine mit GPU ausgestattete virtuelle Maschine einsetzen

Alle großen Cloud-Plattformen bietet mittlerweile GPU-VMs an – manche vorkonfiguriert mit allen erforderlichen Treibern und Bibliotheken (einschließlich TensorFlow). Google Cloud Platform setzt dabei verschiedene GPU-Kontingente um, sowohl weltweit als auch pro Region: Sie können nicht einfach Tausende GPU-VMs ohne vorherige Autorisierung durch Google erzeugen.¹² Standardmäßig ist das weltweite GPU-Kontingent gleich null – Sie können also keine GPU-VMs nutzen. Daher müssen Sie zuerst ein höheres weltweites Kontingent anfordern. Öffnen Sie in der GCP-Konsole das Navigationsmenü und gehen Sie zu »IAM & Verwaltung → Kontingente«. Klicken Sie auf »Messwerte«, dann auf »Keine«, um die Häkchen bei allen Einträgen zu entfernen, suchen Sie anschließend nach »GPU« und wählen Sie »GPUs (all regions)«, um das zugehörige Kontingent angezeigt zu bekommen. Ist dieser Wert null (oder für Ihre Bedürfnisse einfach nicht ausreichend), markieren Sie die Checkbox daneben (es sollte die einzige markierte sein) und klicken auf »Kontingente bearbeiten«. Geben Sie alle erforderlichen Informationen ein und klicken Sie dann auf »Anfrage senden«. Es kann ein paar Stunden (oder auch Tage) dauern, bis Ihre Anfrage bearbeitet und (im Allgemeinen) akzeptiert wird. Standardmäßig gibt es ebenfalls ein Kontingent von einer GPU pro Region und GPU-Typ. Sie können anfragen, auch diese Kontingente zu erhöhen: Klicken Sie auf »Messwerte«, wählen Sie »Keine«, suchen Sie nach »GPU« und markieren Sie den gewünschten GPU-Typ (zum Beispiel »NVIDIA P4 GPUs«). Dann klicken das Auswahlmenü »Zone« an und wählen den gewünschten Ort aus. Markieren Sie die Checkboxen neben den Kontingenten, die Sie ändern wollen, und klicken Sie dann auf »Kontingente bearbeiten«, um eine Anfrage zu stellen.

Wurde Ihr GPU-Kontingent genehmigt, können Sie jetzt in kurzer Zeit eine VM erstellen, die

mit einer oder mehreren GPUs ausgestattet ist, indem Sie die *Deep Learning VM Images* der Google Cloud AI Platform verwenden: Rufen Sie <https://homl.info/dlvm> auf, klicken Sie auf »Zur Konsole« und dann auf »Launch«. Hier können Sie das Konfigurationsformular für die VM ausfüllen. Beachten Sie, dass es in manchen Zonen nicht alle Arten von GPUs gibt und manche überhaupt keine GPUs anbieten (ändern Sie dann die Zone, um die verfügbaren GPUs angezeigt zu bekommen). Achten Sie darauf, dass TensorFlow 2.0 als Framework ausgewählt ist, und markieren Sie »Install NVIDIA GPU driver automatically on first startup«. Es ist auch nicht verkehrt, »Enable access to JupyterLab via URL instead of SSH« auszuwählen – damit wird es sehr einfach, ein Jupyter-Notebook zu starten, das auf dieser GPU-VM läuft und von JupyterLab betrieben wird (das ist eine alternative Weboberfläche zum Ausführen von Jupyter-Notebooks). Wurde die VM erstellt, scrollen Sie im Navigationsmenü zum Abschnitt »Künstliche Intelligenz« herunter und klicken dort auf »AI Platform → Notebooks«. Taucht die Notebook-Instanz in der Liste auf (das kann ein paar Minuten dauern, aktualisieren Sie dann gelegentlich die Seite), klicken Sie auf den Link »JupyterLab öffnen«. Damit wird JupyterLab in der VM ausgeführt und Ihr Browser damit verbunden. Sie können in der VM Notebooks erstellen, beliebigen Code darin ausführen und von ihren GPUs profitieren!

Aber wenn Sie nur ein paar schnelle Tests ausführen oder Notebooks mit Ihren Kollegen austauschen wollen, sollten Sie sich Colaboratory anschauen.

Colaboratory

Die einfachste und günstigste Möglichkeit, auf eine GPU-VM zuzugreifen, ist der Einsatz von *Colaboratory* (oder kurz *Colab*). Es ist kostenlos! Gehen Sie einfach zu <https://colab.research.google.com/> und erstellen Sie ein neues Python-3-Notebook. Damit wird auf Ihrem Google Drive ein Jupyter-Notebook erzeugt (alternativ können Sie auch eines auf GitHub oder Google Drive öffnen oder sogar Ihre eigenen Notebooks hochladen). Die Oberfläche von Colab ähnelt der von Jupyter, nur dass Sie die Notebooks wie normale Google Docs gemeinsam nutzen können. Zudem gibt es ein paar weitere kleine Unterschiede (zum Beispiel können Sie mit speziellen Kommentaren in Ihrem Code praktische Forms (<https://homl.info/colabforms>) erzeugen).

Öffnen Sie ein Colab-Notebook, läuft es auf einer kostenlosen Google-VM, die nur für Sie zur Verfügung steht und die als *Colab Runtime* bezeichnet wird. Standardmäßig nutzt die Runtime lediglich CPUs, aber das können Sie ändern, indem Sie zu »Runtime → Change runtime type« gehen, im Auswahlmenü »Hardware accelerator« und »GPU« auswählen und dann auf »Save« klicken. Tatsächlich könnten Sie sogar »TPU« wählen! (Ja, Sie können wirklich kostenlos eine TPU verwenden – wir sprechen darüber später noch, daher wählen Sie jetzt bitte nur GPU aus.)

Lassen Sie mehrere Colab-Notebooks mit dem gleichen Runtime-Typ laufen (siehe [Abbildung 19-11](#)), werden diese auch die gleiche Colab Runtime verwenden. Schreibt also eines in eine Datei, können die anderen diese Datei lesen. Es ist wichtig, die Auswirkungen auf die Sicherheit zu begreifen: Lassen Sie ein nicht vertrauenswürdiges Colab-Notebook laufen, das von einem fiesen Hacker geschrieben wurde, kann es eventuell private Daten lesen, die von anderen Notebooks erzeugt wurden, und diese dann wieder an den Hacker schicken. Wenn dazu private Zugriffsschlüssel für bestimmte Ressourcen gehören, erhält auch der Hacker Zugriff darauf. Und

installieren Sie eine Bibliothek in der Colab Runtime, haben auch die anderen Notebooks diese Bibliothek. Abhängig von dem, was Sie möchten, kann das großartig oder nervig sein (es bedeutet beispielsweise, dass Sie nicht einfach verschiedene Versionen der gleichen Bibliothek in unterschiedlichen Colab-Notebooks einsetzen können).

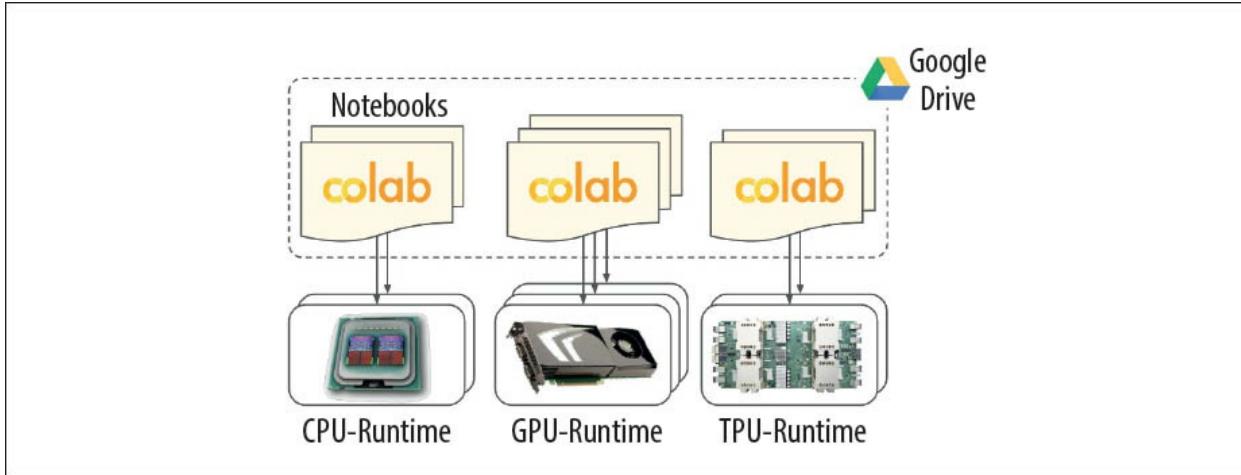


Abbildung 19-11: Colab Runtimes und Notebooks

Colab hat ein paar Einschränkungen: Wie die FAQ schreibt: »Colaboratory ist für eine interaktive Anwendung gedacht. Lang laufende Hintergrundberechnungen, insbesondere auf GPUs, werden eventuell gestoppt. Bitte nutzen Sie Colaboratory nicht zum Schürfen von Kryptowährung.« Die Weboberfläche trennt sich automatisch von der Colab Runtime, wenn Sie sie eine Weile ungenutzt lassen (ca. 30 Minuten). Verbinden Sie sich mit der Colab Runtime erneut, wurde diese eventuell zurückgesetzt, daher sollten Sie darauf achten, immer alle Daten herunterzuladen, die Ihnen wichtig sind. Auch wenn sich die Runtime nicht zurücksetzt, wird sie automatisch nach zwölf Stunden heruntergefahren, da sie nicht für lang laufende Berechnungen gedacht ist. Aber trotz dieser Beschränkungen ist es ein fantastisches Tool zum einfachen Ausführen von Tests, dem Einholen schneller Ergebnisse und der Zusammenarbeit mit Ihren Kollegen.

Das GPU-RAM verwalten

TensorFlow beansprucht automatisch das gesamte verfügbare RAM aller GPUs, wenn Sie einen Graphen das erste Mal ausführen. Das tut es, um die Fragmentierung des GPU-RAM zu begrenzen. Daher können Sie kein zweites TensorFlow-Programm (oder ein anderes, das die GPU erfordert) starten, während das erste noch läuft, weil dann zu wenig RAM verfügbar ist. Das geschieht nicht so oft, wie Sie vielleicht denken, da Sie meist nur ein TensorFlow-Programm auf einer Maschine laufen lassen – ein Trainingsskript, ein TF-Serving-Knoten oder ein Jupyter-Notebook. Wollen Sie aus irgendwelchen Gründen (zum Beispiel zum parallelen Trainieren zweier verschiedener Modelle auf demselben Rechner) mehrere Programme laufen lassen, müssen Sie das GPU-RAM zwischen diesen Prozessen besser aufteilen.

Haben Sie mehrere GPU-Karten in Ihrem Rechner, ist es am einfachsten, jede davon einem Prozess zuzuweisen. Dazu können Sie die Umgebungsvariable `CUDA_VISIBLE_DEVICES` so

setzen, dass jeder Prozess nur die passenden GPU-Karte(n) sieht. Setzen Sie zudem die Umgebungsvariable `CUDA_DEVICE_ORDER` auf `PCI_BUS_ID`, um sicherzustellen, dass sich jede ID immer auf dieselbe GPU-Karte bezieht. Haben Sie beispielsweise vier GPU-Karten, könnten Sie zwei Programme starten und jedem davon zwei GPUs zuweisen, indem Sie Befehle wie die folgenden in zwei getrennten Terminalfenstern ausführen:

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# und in einem anderen Terminal:
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Programm 1 wird dann nur die GPU-Karten 0 und 1 mit den Namen `/gpu:0` und `/gpu:1` sehen, während Programm 2 nur die GPU-Karten 2 und 3 mit den Namen `/gpu:1` und `/gpu:0` sieht (beachten Sie die Reihenfolge). Alles wird wunderbar funktionieren (siehe Abbildung 19-12). Natürlich können Sie diese Umgebungsvariablen auch in Python definieren, indem Sie `os.environ["CUDA_DEVICE_ORDER"]` und `os.environ["CUDA_VISIBLE_DEVICES"]` setzen, solange Sie das tun, bevor Sie TensorFlow verwenden.

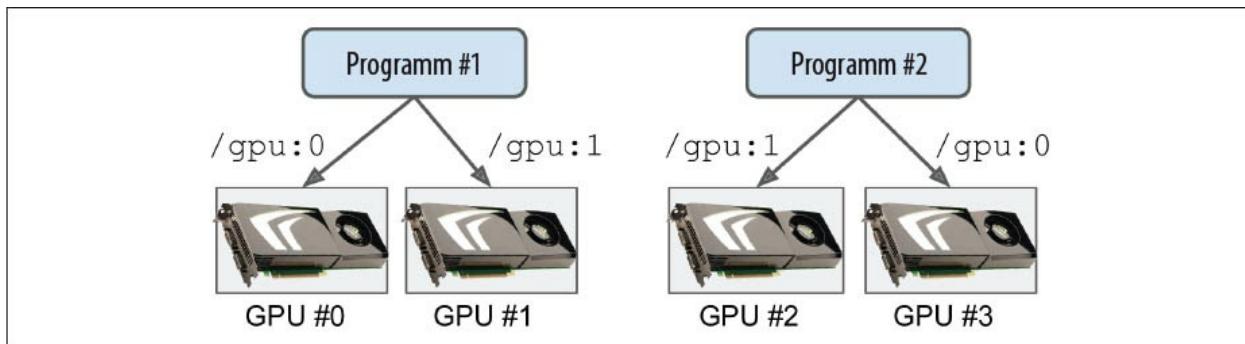


Abbildung 19-12: Jedes Programm erhält zwei eigene GPUs.

Eine andere Möglichkeit ist, TensorFlow nur eine bestimmte Menge GPU-RAM reservieren zu lassen. Das muss direkt nach dem Importieren von TensorFlow geschehen. Damit TensorFlow beispielsweise nur 2 GiB RAM auf jeder GPU reserviert, müssen Sie für jedes reale GPU-Device ein *virtuelles GPU-Device* erzeugen (auch als *logisches GPU-Device* bezeichnet) und dessen Speichergrenze auf 2 GiB setzen (also 2.048 MiB):

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_virtual_device_configuration(
        gpu,
        [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

Jetzt können (angenommen, Sie haben vier GPUs mit jeweils mindestens 4 GiB RAM) zwei Programme parallel laufen, von denen jedes alle vier GPU-Karten verwendet (siehe Abbildung 19-13).

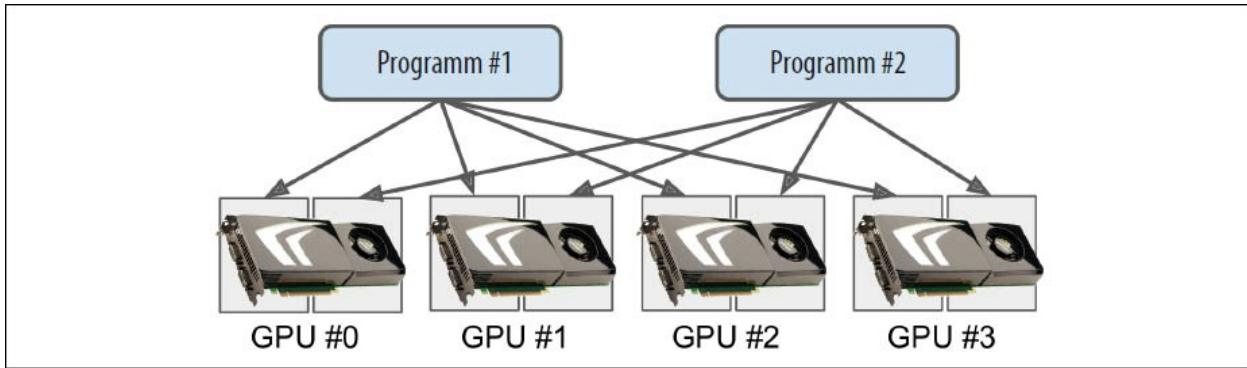


Abbildung 19-13: Jedes Programm verwendet alle vier GPUs, aber nur jeweils 2 GiB des RAM.

Wenn Sie den Befehl `nvidia-smi` ausführen, während beide Programme laufen, sollten Sie sehen, dass jeder Prozess 2 GiB des auf jeder Karte verfügbaren RAM verwendet:

```
$ nvidia-smi
[...]
+-----+
| Processes:                               GPU Memory |
| GPU     PID   Type  Process name        Usage      |
| ======|=====|=====|=====|=====|
| 0       2373    C   /usr/bin/python3    2241MiB  |
| 0       2533    C   /usr/bin/python3    2241MiB  |
| 1       2373    C   /usr/bin/python3    2241MiB  |
| 1       2533    C   /usr/bin/python3    2241MiB  |
[...]
```

Eine weitere Option ist, TensorFlow anzulegen, Speicher nur dann anzufordern, wenn es diesen benötigt (das muss ebenfalls direkt nach dem Importieren von TensorFlow geschehen):

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_memory_growth(gpu, True)
```

Sie können auch die Umgebungsvariable `TF_FORCE_GPU_ALLOW_GROWTH` auf `true` setzen. Dann gibt TensorFlow niemals Speicher frei, den es einmal angefordert hat (auch wieder, um eine Speicherfragmentierung zu vermeiden) – außer natürlich, wenn das Programm endet. Mit dieser Option kann es schwerer sein, ein deterministisches Verhalten zu garantieren (zum Beispiel stürzt ein Programm vielleicht ab, weil der Speicherbedarf eines anderen Programms

durch die Decke ging), daher werden Sie im Produktivumfeld eher bei den anderen Optionen bleiben. Aber in manchen Fällen ist sie sehr sinnvoll – wenn Sie beispielsweise auf einem Rechner mehrere Jupyter-Notebooks laufen lassen, von denen viele TensorFlow verwenden. Darum ist `TF_FORCE_GPU_ALLOW_GROWTH` in Colab Runtimes auf `true` gesetzt.

Und schließlich wollen Sie in manchen Fällen eine GPU in zwei oder mehr *virtuelle GPUs* aufteilen – etwa wenn Sie einen verteilten Algorithmus testen wollen (dies ist eine praktische Möglichkeit, die Codebeispiele aus diesem Kapitel auch dann testen zu können, wenn Sie nur eine einzelne GPU haben, wie beispielsweise in einer Colab Runtime). Der folgende Code teilt die erste GPU in zwei virtuelle Devices mit jeweils 2 GiB RAM auf (auch dies muss direkt nach dem Importieren von TensorFlow geschehen):

```
physical_gpus = tf.config.experimental.list_physical_devices("GPU")
tf.config.experimental.set_virtual_device_configuration(
    physical_gpus[0],
    [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048),
     tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

Diese zwei virtuellen Devices werden dann als `/gpu:0` und `/gpu:1` bezeichnet, und Sie können Operationen und Variablen auf beiden so einsetzen, als würde es sich um zwei echte, unabhängige GPUs handeln. Schauen wir nun, wie TensorFlow entscheidet, auf welchen Devices es Variablen platzieren und Operationen ausführen sollte.

Operationen und Variablen auf Devices verteilen

Das TensorFlow-Whitepaper (<https://homl.info/67>)¹³ stellt einen freundlichen *Dynamic-Placer*-Algorithmus vor, der automatisch Operationen auf alle verfügbaren Devices verteilt und dabei zum Beispiel berücksichtigt, wie die gemessene Rechenzeit in vorherigen Durchläufen durch den Graphen aussah, wie groß die Ein- und Ausgabetensoren für jede Operation ungefähr sein werden, wie viel RAM auf jedem Device zur Verfügung steht, wie groß die Verzögerung durch die Kommunikation beim Übermitteln von Daten zu den Devices und zurück ist und was für Hints und Constraints der Anwender mitgegeben hat. In der Praxis hat sich dieser Algorithmus als weniger effizient herausgestellt als ein kleiner Satz an Verteilungsregeln, die vom Benutzer vorgegeben werden, sodass das TensorFlow-Team den Dynamic Placer nicht mehr weiter verfolgt hat.

`tf.keras` und `tf.data` machen im Allgemeinen ihre Aufgabe beim Verteilen von Operationen und Variablen an die richtigen Stellen (zum Beispiel aufwendige Berechnungen auf der GPU, Datenvorverarbeitung auf der CPU) schon ziemlich gut. Aber Sie können Operationen und Variablen auch manuell verteilen, wenn Sie mehr Kontrolle haben wollen:

- Wie schon erwähnt, sollten die Operationen zur Datenvorverarbeitung im Allgemeinen auf der CPU ausgeführt werden, während die Operationen für das neuronale Netz auf die GPU gehören.

- GPUs haben meist nur eine eingeschränkte Bandbreite, daher ist es wichtig, unnötige Datentransfers auf die GPUs und von ihnen zurück zu vermeiden.
- Es ist einfach und billig, einen Rechner mit mehr CPU-RAM auszustatten, daher ist meist mehr als genug davon verfügbar. Das GPU-RAM hingegen ist fest mit der GPU verbunden – es handelt sich um eine begrenzte Ressource. Wird eine Variable nicht in den nächsten paar Trainingsschritten benötigt, sollte sie vermutlich auf der CPU platziert werden (zum Beispiel gehören Datasets allgemein auf die CPU).

Standardmäßig werden alle Variablen und Operationen auf der ersten GPU (mit dem Namen `/gpu:0`) untergebracht, mit Ausnahme derer, die keinen GPU-Kernel besitzen¹⁴ – diese landen auf der CPU (`/cpu:0`). Das Attribut `device` eines Tensors oder einer Variablen teilt Ihnen mit, auf welchem Device es platziert wurde:¹⁵

```
>>> a = tf.Variable(42.0)

>>> a.device

'/job:localhost/replica:0/task:0/device:GPU:0'

>>> b = tf.Variable(42)

>>> b.device

'/job:localhost/replica:0/task:0/device:CPU:0'
```

Sie können das Präfix `/job:localhost/replica:0/task:0` zunächst ignorieren (es ermöglicht Ihnen, Operationen auf anderen Maschinen zu platzieren, wenn Sie ein TensorFlow-Cluster nutzen – wir werden über Jobs, Repliken und Tasks später noch sprechen). Wie Sie sehen, wurde die erste Variable auf GPU 0 platziert, was das Standard-Device ist. Aber die zweite Variable wurde auf der CPU platziert: Das liegt daran, dass es keine GPU-Kernels für Integer-Variablen (oder für Operationen mit Integer-Tensoren) gibt, daher musste TensorFlow wieder auf die CPU zurückgreifen.

Wollen Sie eine Operation statt auf dem Standardgerät auf einem anderen Device unterbringen, nutzen Sie den Kontext `tf.device()`:

```
>>> with tf.device("/cpu:0"):

...     c = tf.Variable(42.0)

...

>>> c.device

'/job:localhost/replica:0/task:0/device:CPU:0'
```



Die CPU wird immer als einzelnes Device behandelt (`/cpu:0`), auch wenn Ihre Maschine mehrere CPU-Cores besitzt. Jede Operation, die auf der CPU landet, kann parallel auf

mehreren Cores laufen, wenn sie einen Multithreaded-Kernel besitzt.

Versuchen Sie explizit, eine Operation oder Variable auf ein Device zu bringen, das nicht existiert oder für das kein Kernel vorhanden ist, werden Sie eine Exception erhalten. Aber in manchen Fällen möchten Sie dann lieber auf die CPU zurückgreifen – läuft Ihr Programm beispielsweise sowohl auf Nur-CPU- wie auch auf GPU-Maschinen, möchten Sie vielleicht, dass TensorFlow auf der Nur-CPU-Maschine Ihr `tf.device("/gpu: *")` ignoriert. Dazu können Sie direkt nach dem Importieren von TensorFlow `tf.config.set_soft_device_placement(True)` aufrufen. Schlägt dann eine Platzierungsanforderung fehl, greift TensorFlow auf seine Standardplatzierungsregeln zurück (also standardmäßig GPU 0, wenn diese vorhanden ist und ein GPU-Kernel vorhanden ist, ansonsten CPU 0).

Wie führt nun TensorFlow alle diese Operationen über mehrere Devices hinweg genau aus?

Paralleles Ausführen auf mehreren Devices

Wie wir in [Kapitel 12](#) gesehen haben, ist einer der Vorteile des Einsatzes von TF Functions die parallele Verarbeitung. Schauen wir uns diese ein bisschen genauer an. Führt TensorFlow eine TF Function aus, analysiert es zunächst ihren Graphen, um die Liste der Operationen zu erhalten, die ausgewertet werden müssen, anschließend zählt es, wie viele Abhängigkeiten jede davon besitzt. TensorFlow fügt dann jede Operation ohne Abhängigkeiten (also jede Source-Operation) zur Auswertungsqueue des Device dieser Operation hinzu (siehe [Abbildung 19-14](#)). Wurde eine Operation ausgewertet, wird der Abhängigkeitszähler jeder Operation, die davon abhängt, heruntergezählt. Erreicht der Abhängigkeitszähler einer Operation die Null, wird sie in die Auswertungsqueue ihres Device gesteckt. Und sind dann alle Knoten, die TensorFlow benötigt, ausgewertet worden, gibt es deren Ausgabe zurück.

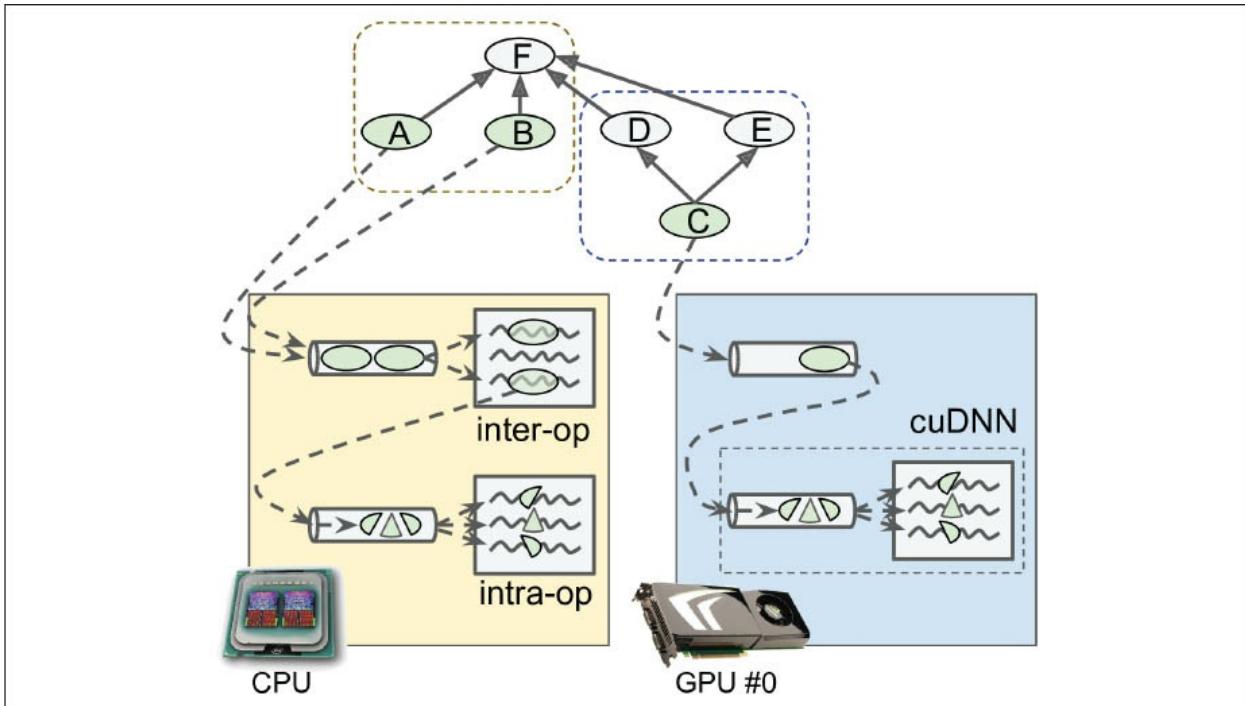


Abbildung 19-14: Parallelisierte Ausführung eines TensorFlow-Graphen

Operationen in der Auswertungsqueue der CPU werden in einen Thread-Pool dispatcht, der als *Inter-Op Thread Pool* bezeichnet wird. Besitzt die CPU mehrere Cores, werden diese Operationen effektiv parallel ausgeführt. Manche Operationen haben Multithreaded-CPU-Kernels: Diese Kernels teilen ihre Tasks in mehrere Suboperationen auf, die in einer anderen Auswertungsqueue landen und an einen zweiten Thread Pool namens *Intra-Op Thread Pool* dispatcht werden (der von allen Multithreaded-Kernels gemeinsam genutzt wird). Kurz gesagt, können mehrere Operationen und Suboperationen auf unterschiedlichen CPU-Cores parallel ausgewertet werden.

Bei der GPU ist es ein bisschen einfacher. Operationen in der Auswertungsqueue einer GPU werden sequenziell ausgewertet. Aber die meisten Operationen besitzen Multithreaded-GPU-Kernels, die meist durch Bibliotheken implementiert sind, auf denen TensorFlow aufbaut, wie zum Beispiel CUDA und cuDNN. Diese Implementierungen besitzen ihre eigenen Thread Pools und setzen typischerweise so viele GPU-Threads wie möglich ein (was der Grund dafür ist, dass es keinen Bedarf für einen Inter-Op Thread Pool auf GPUs gibt, jede Operationen verwendet schon viele GPU-Threads). So handelt es sich beispielsweise in Abbildung 19-14 bei den Operationen A, B und C um Source-Ops, und sie können direkt ausgewertet werden. Die Operationen A und B werden auf die CPU platziert, sodass sie an die Auswertungsqueue der CPU geschickt, dann an den Inter-Op Thread Pool dispatcht und direkt parallel ausgewertet werden. Operation A besitzt einen Multithreaded-Kernel – ihre Berechnungen werden in drei Teile aufgeteilt, die parallel vom Intra-Op Thread Pool ausgeführt werden. Operation C wandert in die Auswertungsqueue von GPU 0, und in diesem Beispiel setzt deren GPU-Kernel cuDNN ein, die ihren eigenen Intra-Op Thread Pool mitbringt und Operationen auf viele GPU-Threads parallel verteilt. Nehmen wir an, C sei zuerst fertig. Die Abhängigkeitszähler von D und E

werden heruntergezählt, und sie erreichen null, sodass beide Operationen in die Auswertungsqueue von GPU 0 geschoben und sequenziell ausgeführt werden. Beachten Sie, dass C nur einmal ausgewertet wird, auch wenn sowohl D wie auch E davon abhängen. Gehen wir nun davon aus, dass B als Nächstes fertig wird. Dann wird der Abhängigkeitszähler von F von 4 auf 3 reduziert, aber da er noch nicht bei 0 steht, wird F noch nicht ausgeführt. Sind A, D und E fertig, erreicht auch der Abhängigkeitszähler von F den Wert 0, und er wird in die Auswertungsqueue der CPU übergeben und ausgewertet. Schließlich gibt TensorFlow die angefragten Ergebnisse zurück.

Ein bisschen zusätzliche Magie geschieht durch TensorFlow, wenn die TF Function eine zustandsbehaftete Ressource (wie zum Beispiel eine Variable) verändert – es stellt sicher, dass die Reihenfolge der Ausführung der Reihenfolge im Code entspricht, auch wenn es keine explizite Abhängigkeit zwischen den Anweisungen gibt. Enthält beispielsweise Ihre TF Function `v.assign_add(1)`, gefolgt von `v.assign(v * 2)`, stellt TensorFlow sicher, dass diese Operationen in dieser Reihenfolge ausgeführt werden.



Sie können die Anzahl an Threads im Inter-Op Thread Pool steuern, indem Sie `tf.config.threading.set_inter_op_parallelism_threads()` aufrufen. Für die Menge an Intra-Op Threads nutzen Sie `tf.config.threading.set_intra_op_parallelism_threads()`. Das ist nützlich, wenn Sie nicht wollen, dass TensorFlow alle CPU-Cores in Beschlag nimmt oder wenn es nur einen Thread geben soll.¹⁶

Jetzt haben Sie alles zusammen, um beliebige Operationen auf beliebigen Devices laufen lassen und Ihre GPUs richtig ausreizen zu können! Hier ein paar Dinge, die damit möglich sind:

- Sie könnten mehrere Modelle parallel trainieren – jedes auf seiner eigenen GPU. Schreiben Sie einfach ein Trainingsskript für jedes Modell und führen Sie diese parallel aus, wobei Sie `CUDA_DEVICE_ORDER` und `CUDA_VISIBLE_DEVICES` so setzen, dass jedes Skript nur ein einzelnes GPU-Device sieht. Das ist toll, wenn Sie Hyperparameter anpassen wollen, weil Sie so mehrere Modelle mit unterschiedlichen Hyperparametern parallel trainieren können. Haben Sie eine einzelne Maschine mit zwei GPUs und dauert es eine Stunde, ein Modell auf einer GPU zu trainieren, dauert das parallele Trainieren von zwei Modellen – jedes auf seiner eigenen GPU – auch nur eine Stunde!
- Sie könnten ein Modell auf einer einzelnen GPU trainieren und die gesamte Vorverarbeitung parallel dazu auf der CPU mithilfe der Dataset-Methode `pre_fetch()` durchführen¹⁷, um die nächsten paar Batches im Voraus vorzubereiten, sodass sie zur Verfügung stehen, wenn die GPU sie benötigt (siehe [Kapitel 13](#)).
- Erhält Ihr Modell zwei Bilder als Eingabe und verarbeitet sie mit zwei CNNs, bevor deren Ergebnisse zusammengebracht werden, läuft es vermutlich viel schneller, wenn Sie jedes CNN auf einer eigenen GPU unterbringen.
- Sie können ein effizientes Ensemble erstellen: Platzieren Sie einfach auf jeder GPU ein anders trainiertes Modell, sodass Sie alle Vorhersagen viel schneller bekommen, um die abschließende Vorhersage des Ensembles zu erhalten.

Wie sieht es nun aber aus, wenn Sie ein einzelnes Modell auf mehreren GPUs *trainieren* wollen?

Modelle auf mehreren Devices trainieren

Es gibt zwei zentrale Vorgehensweisen, um ein einzelnes Modell auf mehreren Devices zu trainieren: *parallelisierte Modelle*, bei denen das Modell auf die Devices verteilt wird, und *parallelisierte Daten*, bei denen das Modell auf jedes Device repliziert wird und jede Replik mit einer Untermenge der Daten trainiert wird. Schauen wir uns diese beiden Optionen näher an, bevor wir ein Modell auf mehreren GPUs trainieren.

Parallelisierte Modelle

Bisher haben wir jedes neuronale Netz auf einem einzelnen Device ausgeführt. Wie können wir ein einzelnes neuronales Netz auf mehrere Devices verteilen? Dazu müssen Sie Ihr Modell in einzelne Stücke teilen und jedes Teilstück auf einem anderen Device ausführen. 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 19-15). Intuitiv mag es so erscheinen, dass man einfach jede Schicht auf einem anderen Device platzieren kann. Dies funktioniert aber nicht, weil jede Schicht auf die Ausgabe der vorherigen Schicht warten muss, bevor sie überhaupt etwas tun kann. Vielleicht lässt sich das Netz stattdessen vertikal zerschneiden – die linke Hälfte jeder Schicht auf ein Device, die rechte Hälfte auf ein anderes? 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 vorherigen benötigt. Es kommt also zu einer Menge Kommunikation zwischen den Devices (durch die gestrichelten Pfeile dargestellt). Dies macht mit hoher Wahrscheinlichkeit die Vorteile des parallelen Rechnens wieder zunichte, da die Kommunikation zwischen Devices langsam ist (besonders zwischen separaten Rechnern).

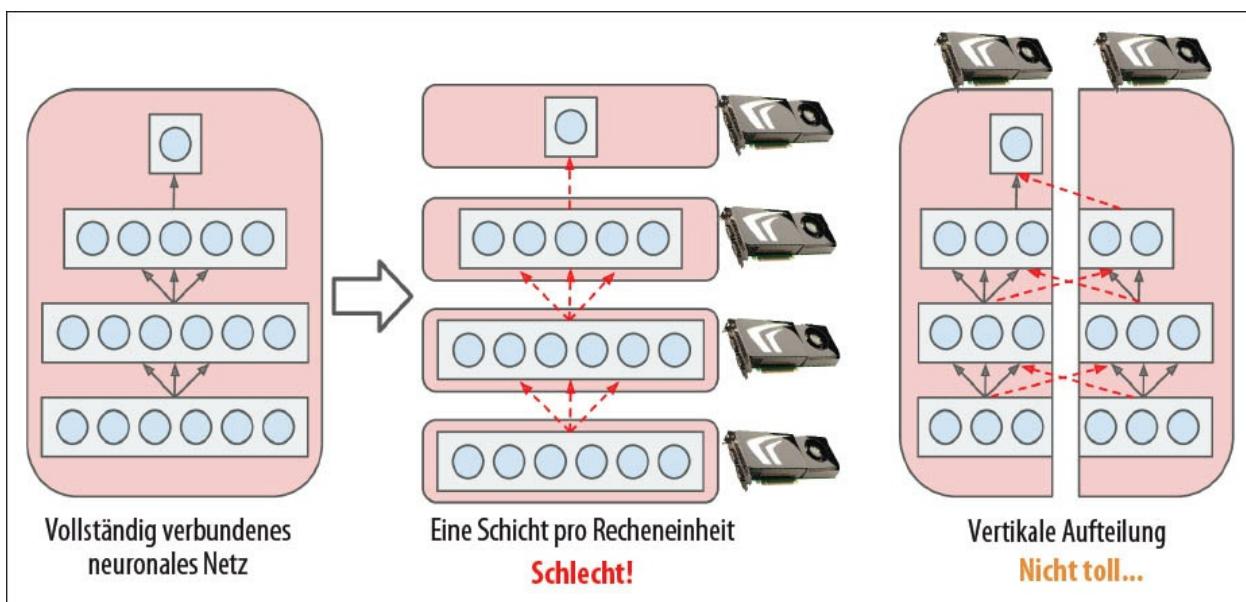


Abbildung 19-15: Aufteilen eines vollständig verbundenen neuronalen Netzes

Manche Architekturen neuronaler Netze, wie zum Beispiel Convolutional Neural Networks (siehe Kapitel 14), haben Schichten, die nur teilweise mit den weiter vorn gelegenen Schichten

verbunden sind. Dort ist das effiziente Aufteilen auf mehrere Geräte viel einfacher (siehe Abbildung 19-16).

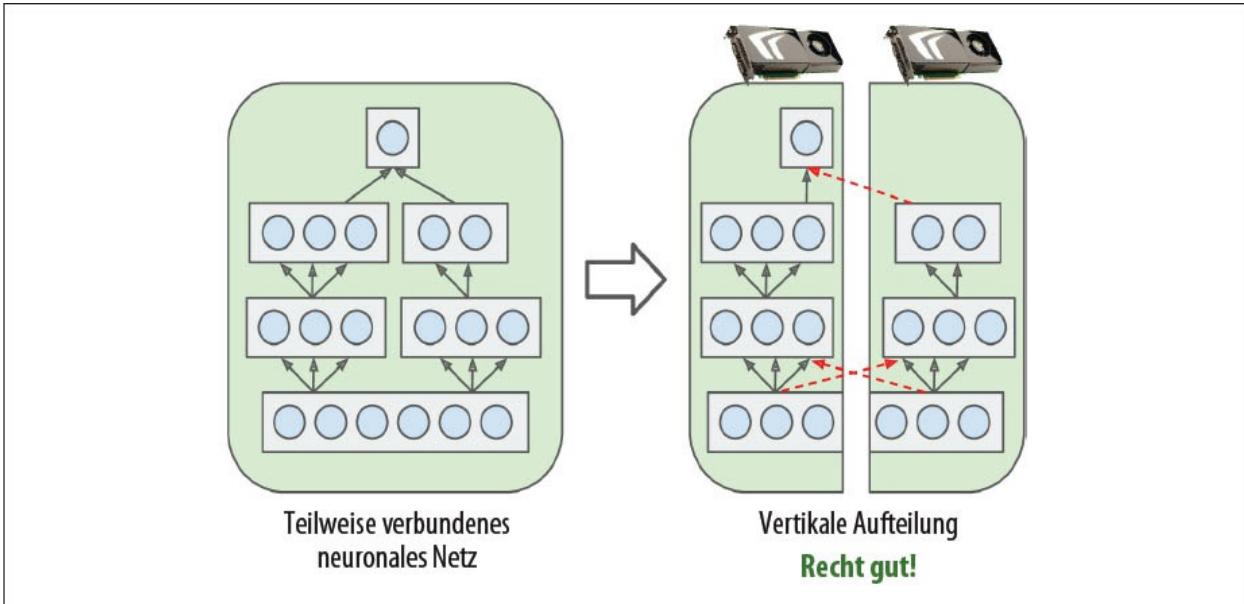


Abbildung 19-16: Aufteilen eines teilweise verbundenen neuronalen Netzes

Rekurrente Deep-Learning-Netze (siehe Kapitel 15) können ein bisschen effizienter auf mehrere GPUs verteilt werden. Wenn Sie ein derartiges Netz horizontal aufteilen und jede Schicht auf einem anderen Device landet und wenn Sie dann das Netz mit einer Eingabesequenz füttern, ist beim ersten Schritt nur ein Device aktiv (es verarbeitet den ersten Wert der Sequenz), beim zweiten Schritt sind es zwei (die zweite Schicht verarbeitet die Ausgabe der ersten Schicht, während die erste Schicht den zweiten Wert verarbeitet), und beim dritten Schritt hat sich das Signal bis zur Ausgabeschicht fortgepflanzt, sodass alle Devices gleichzeitig aktiv sind (siehe Abbildung 19-17).

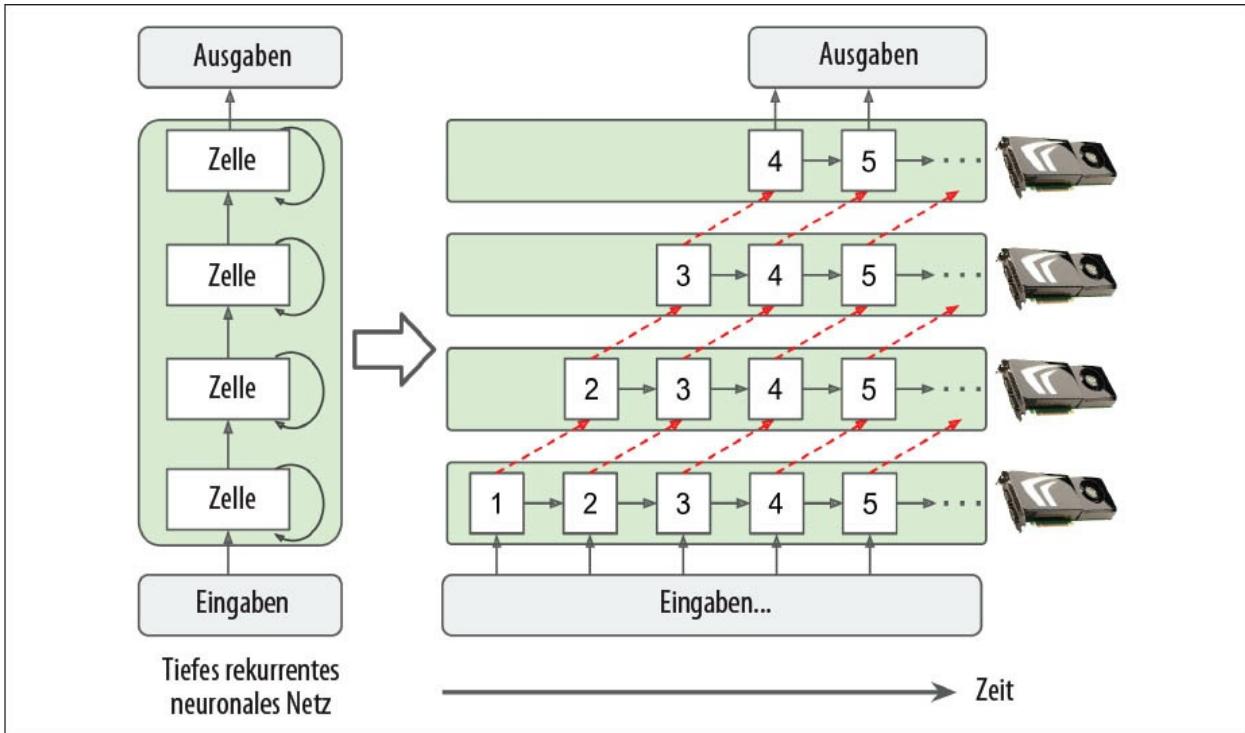


Abbildung 19-17: Aufteilen eines rekurrenten Deep-Learning-Netzes

Auch hier gibt es noch eine Menge Kommunikation zwischen den Devices, aber da jede Zelle sehr komplex sein kann, mag der Nutzen des parallelen Ausführens mehrerer Zellen (in der Theorie) größer sein als der Kommunikationsaufwand. In der Praxis läuft allerdings ein normaler Stack mit LSTM-Schichten auf einer einzelnen GPU viel schneller.

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 Devices mit dem höchsten Kommunikationsaufwand auf demselben Rechner zu platzieren.¹⁸ Sehen wir uns eine viel einfachere und im Allgemeinen deutlich effizientere Option an – parallelisierte Daten.

Parallelisierte Daten

Eine weitere Möglichkeit, das Trainieren eines neuronalen Netzes zu parallelisieren, ist, es auf jedem Device zu replizieren, auf 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*. Es gibt viele Varianten dieser Idee, daher wollen wir uns die wichtigsten anschauen.

Parallelisierte Daten mit der Spiegel-Strategie

Der einfachste Ansatz ist natürlich, alle Modellparameter auf allen GPUs vollständig zu spiegeln und immer genau die gleichen Parameteraktualisierungen auf jeder GPU anzuwenden. So bleiben alle Repliken immer genau gleich. Das nennt sich *Spiegel-Strategie*, und sie stellt sich als ziemlich effizient heraus, insbesondere beim Einsatz einer einzelnen Maschine (siehe [Abbildung](#)

19-18).

Knifflig wird es, wenn es um das effiziente Berechnen des Mittelwerts aller Gradienten von allen GPUs und das Verteilen der Ergebnisse geht. Das kann mit einem *All-Reduce*-Algorithmus geschehen, einer Klasse von Algorithmen, bei denen mehrere Knoten zusammenarbeiten, um eine Reduce-Operation durchzuführen (wie das Berechnen von Mittelwert, Summe oder Maximum), während gleichzeitig sichergestellt wird, dass alle Knoten das gleiche Endergebnis erhalten. Zum Glück gibt es fertige Implementierungen für solche Algorithmen, wie wir noch sehen werden.

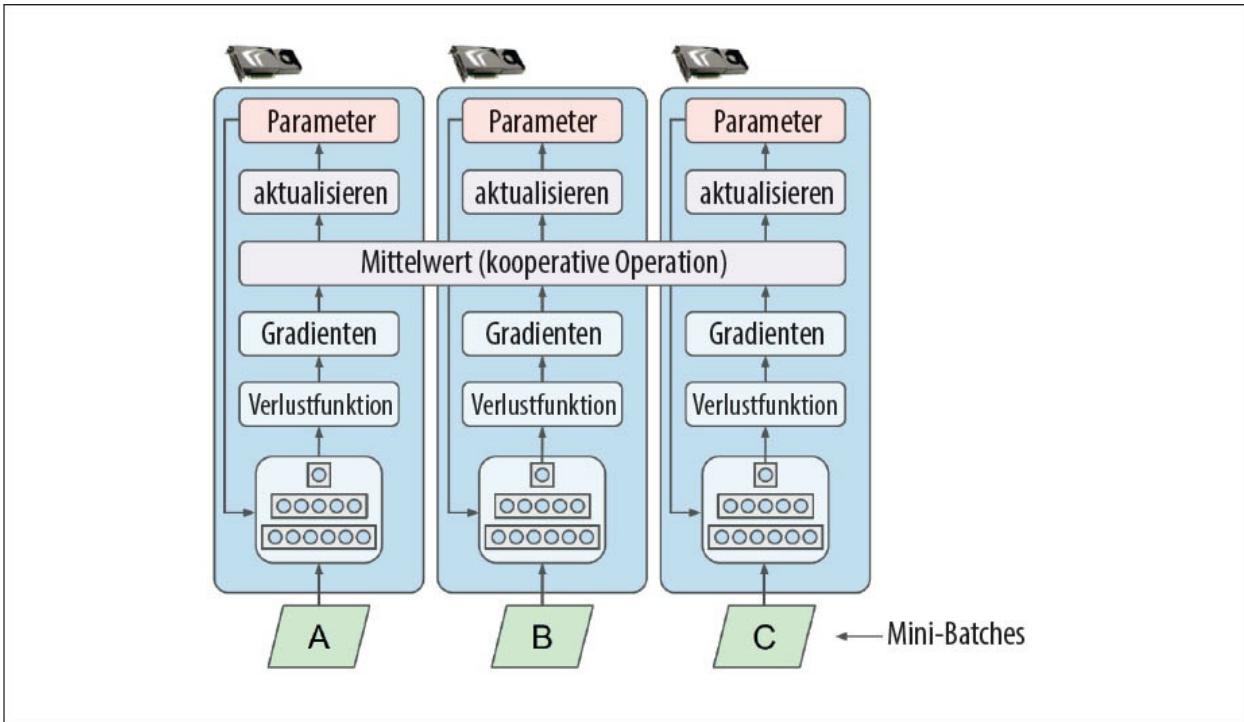


Abbildung 19-18: Parallelisierte Daten mit der Spiegel-Strategie

Parallelisierte Daten mit zentralisierten Parametern

Bei einem anderen Ansatz werden die Modellparameter außerhalb der GPU-Devices gespeichert, die die Berechnungen durchführen (die *Worker*), zum Beispiel auf der CPU (siehe [Abbildung 19-19](#)).

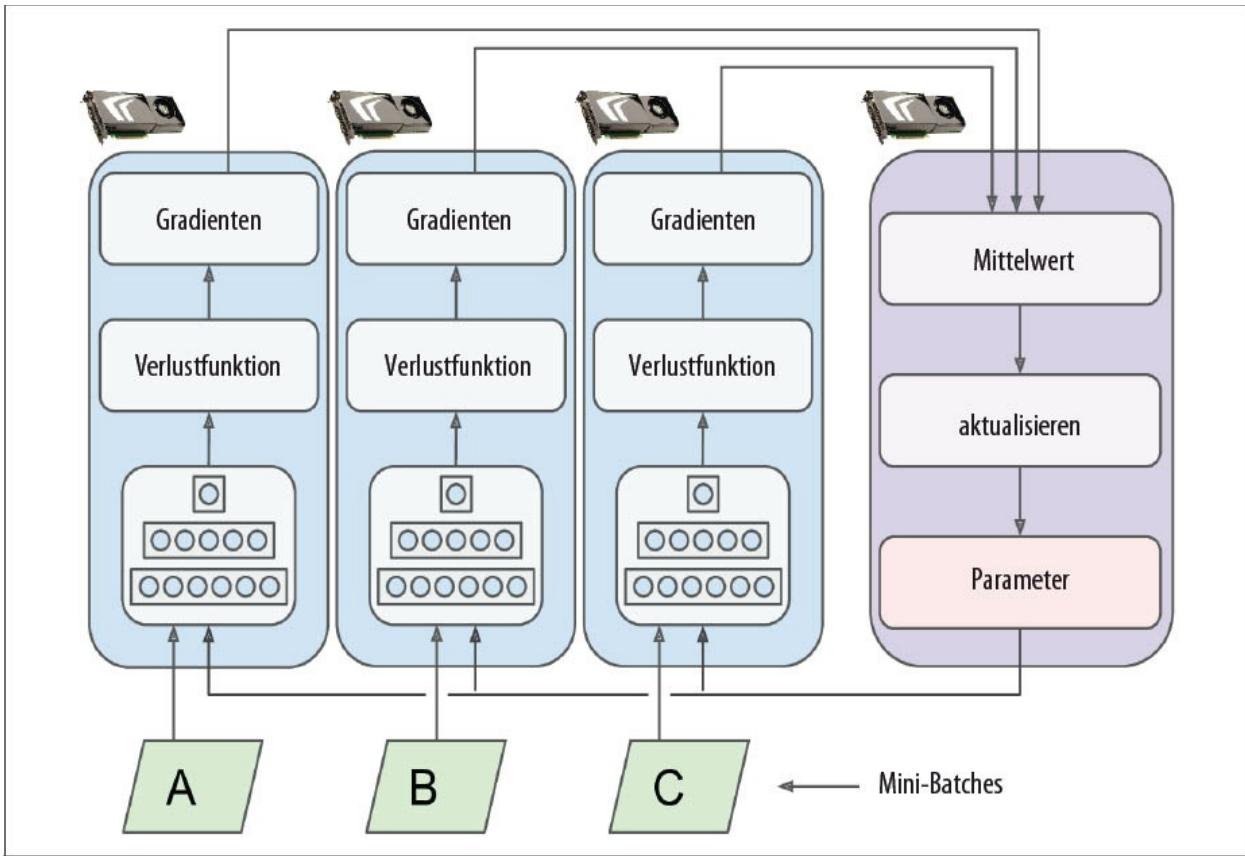


Abbildung 19-19: Parallelisierte Daten mit zentralisierten Parametern

In einem verteilten Setup können Sie alle Parameter auf einem oder mehreren Servern speichern, die nur eine CPU besitzen (die *Parameterserver*) und deren einzige Aufgabe das Bereitstellen und Aktualisieren der Parameter ist.

Während bei der Spiegel-Strategie die Gewichte synchron auf allen GPUs aktualisiert werden, erlaubt dieser zentralisierte Ansatz entweder synchrone oder asynchrone Updates. Schauen wir uns die Vor- und Nachteile beider Optionen an.

Synchrone Updates. Bei *synchronen 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 MiniBatch 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 Netzwerkantrittsrate 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 sind, können die ersten 18 Repliken sofort weiterarbeiten, ohne auf die 2 langsamen Repliken zu warten. Diese Konfiguration lässt sich als 18 Repliken plus 2 *Ersatzrepliken* umschreiben.¹⁹

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 19-19](#) 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. Aber 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 19-20](#)). 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.

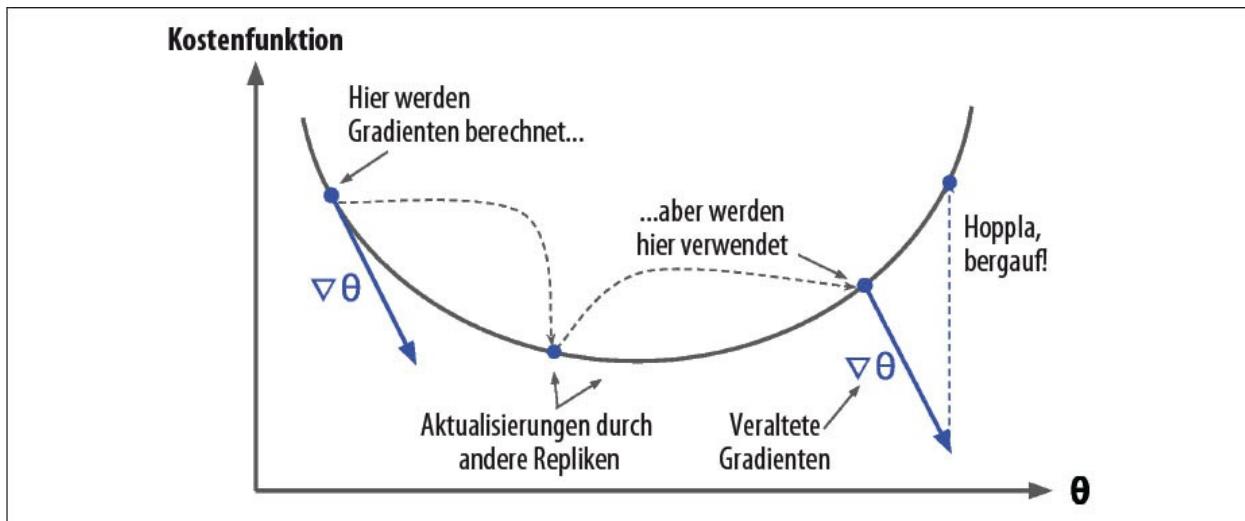


Abbildung 19-20: Stale Gradients beim Verwenden asynchroner Updates

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 *Warmup-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 (<https://homl.info/68>)²⁰ wertete unterschiedliche Ansätze aus und kam zu dem Schluss, dass parallelisierte Daten mit synchronen Updates und einigen Ersatzrepliken das effizienteste Verfahren sei. Es konvergiert nicht nur schneller, sondern gelangt auch zu einem besseren Modell. Allerdings ist dies noch immer ein aktives Forschungsgebiet, Sie sollten daher die asynchronen Updates noch nicht abtun.

Erschöpfen der Netzwerkbandbreite

Sowohl bei synchronen wie auch bei 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. Genauso müssen beim Einsatz der Spiegel-Strategie die von jeder GPU erzeugten Gradienten mit allen anderen GPUs geteilt 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 aufwiegert. 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 mit großer Speicherbandbreite 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 (<https://homl.info/69>), 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-fache Beschleunigung auf 8 GPUs
- Inception/ImageNet: 32-fache Beschleunigung auf 50 GPUs
- RankBrain: 300-fache Beschleunigung auf 500 GPUs

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 bekämpfen, indem Sie eher wenige leistungsfähige statt viele schwachbrüstige GPUs nutzen und Ihre GPUs auf wenigen, dafür aber gut miteinander vernetzten Servern gruppieren. Sie können auch die Genauigkeit der Modellparameter von Gleitkommazahlen mit 32 Bit (`tf.float32`) auf 16 Bit (`tf.float16`) reduzieren. Damit halbieren Sie die zu übertragende Datenmenge, ohne die Konvergenzrate oder die Leistung des Modells häufig deutlich zu schmälern. Und wenn Sie zentralisierte Parameter verwenden, können Sie diese auf mehrere Parameterserver aufteilen: Mit mehr Parameterservern sinkt die Netzwerklast auf jedem Server, und das Risiko der Bandbreitensättigung wird begrenzt.

Okay, trainieren wir jetzt ein Modell auf mehreren GPUs!

Mit der **Distribution Strategies API** auf mehreren Devices trainieren

Viele Modelle können ziemlich gut auf einer einzelnen GPU oder sogar auf einer CPU trainiert werden. Aber wenn das Training zu lange dauert, können Sie versuchen, es auf mehrere GPUs auf derselben Maschine zu verteilen. Wenn das immer noch zu langsam ist, versuchen Sie es mit leistungsfähigeren GPUs oder mit mehr GPUs in der Maschine. Muss das Modell sehr stark rechnen (wie zum Beispiel mit großen Matrixmultiplikationen), wird es viel schneller auf leistungsfähigen GPUs laufen, und Sie können sogar versuchen, TPUs der Google Cloud AI Platform zu verwenden, die für solche Modelle meist noch schneller sind. Aber wenn Sie keine weiteren GPUs mehr in die Maschine bekommen und wenn TPUs für Sie nicht infrage kommen (zum Beispiel weil Ihr Modell vielleicht nicht so sehr von TPUs profitiert oder weil Sie Ihre eigene Hardwareinfrastruktur verwenden wollen), können Sie versuchen, das Modell über mehrere Server verteilt zu trainieren, von denen jeder mehrere GPUs besitzt (wenn das immer noch nicht reicht, können Sie als letzten Ausweg versuchen, das Modell mehr oder weniger zu parallelisieren, aber dafür ist sehr viel mehr Aufwand nötig). In diesem Abschnitt werden wir sehen, wie wir Modelle in einem größeren Rahmen trainieren – beginnend mit mehreren GPUs auf derselben Maschine (oder TPUs), um dann zu mehreren GPUs auf mehreren Rechnern zu wechseln.

Zum Glück bringt TensorFlow eine sehr einfache API mit, die sich um die ganze Komplexität kümmert – die *Distribution Strategies API*. Um ein Keras-Modell auf allen verfügbaren GPUs (zunächst auf einer einzelnen Maschine) zu trainieren, verwenden Sie parallelisierte Daten mit der Spiegel-Strategie, erstellen ein Objekt vom Typ `MirroredStrategy`, rufen dessen Methode `scope()` auf, um einen Verteilungskontext zu erhalten, und verpacken das Erstellen und Kompilieren Ihres Modells in diesem Kontext. Dann rufen Sie ganz normal die Methode `fit()` des Modells auf:

```
distribution = tf.distribute.MirroredStrategy()
```

```
with distribution.scope():
```

```

mirrored_model = tf.keras.Sequential([...])

mirrored_model.compile(...)

batch_size = 100 # muss durch die Anzahl an Repliken teilbar sein

history = mirrored_model.fit(X_train, y_train, epochs=10)

```

Hinten den Kulissen ist sich `tf.keras` der Verteilungssituation bewusst, daher weiß es in diesem `MirroredStrategy`-Kontext, dass es alle Variablen und Operationen auf alle verfügbaren GPU-Devices verteilen muss. Beachten Sie, dass die Methode `fit()` automatisch jeden Trainingsbatch unter allen Repliken aufteilen wird, daher ist es wichtig, dass die Batchgröße ein Vielfaches der Anzahl an Repliken ist. Und das ist alles! Das Training wird im Allgemeinen deutlich schneller ablaufen als beim Einsatz eines einzelnen Device, und die Codeänderung war wirklich minimal.

Sind Sie mit dem Trainieren Ihres Modells fertig, können Sie es verwenden, um effizient Vorhersagen zu treffen: Rufen Sie die Methode `predict()` auf, wird der Batch automatisch zwischen allen Repliken aufgeteilt, und die Vorhersagen werden parallel abgearbeitet (auch hier muss die Batchgröße wieder durch die Anzahl an Repliken teilbar sein). Rufen Sie die Methode `save()` des Modells auf, wird es als normales Modell gespeichert, *nicht* als Spiegel-Modell mit mehreren Repliken. Wenn Sie es dann laden, wird es wie ein normales Modell auf einem einzelnen Device laufen (standardmäßig GPU 0 oder auf der CPU, wenn es keine GPU gibt). Wollen Sie ein Modell laden und auf allen verfügbaren Devices laufen lassen, müssen Sie `keras.models.load_module()` in einem Verteilungskontext aufrufen:

```

with distribution.scope():

    mirrored_model = keras.models.load_model("my_mnist_model.h5")

```

Wollen Sie nur einen Teil der verfügbaren GPU-Devices nutzen, können Sie die Liste an den Konstruktor von `MirroredStrategy` übergeben:

```
distribution = tf.distribute.MirroredStrategy(["/gpu:0", "/gpu:1"])
```

Standardmäßig nutzt die Klasse `MirroredStrategy` die *NVIDIA Collective Communications Library* (NCCL) für die AllReduce-Durchschnittsoperation, aber Sie können das ändern, indem Sie das Argument `cross_device_ops` auf eine Instanz der Klasse `tf.distribute.HierarchicalCopyAllReduce` oder `tf.distribute.ReduceToOneDevice` setzen. Die Standard-NCCL-Variante basiert auf der Klasse `tf.distribute.NcclAllReduce`, die im Allgemeinen schneller ist. Das hängt von der Anzahl und Art der GPUs ab, aber Sie können es durchaus einmal versuchen.²¹

Wollen Sie parallelisierte Daten mit zentralisierten Parametern ausprobieren, ersetzen Sie die

MirroredStrategy durch die CentralStorageStrategy:

```
distribution = tf.distribute.experimental.CentralStorageStrategy()
```

Sie können optional das Argument `compute_devices` setzen, um die Liste der Devices festzulegen, die Sie als Worker einsetzen wollen (standardmäßig werden alle verfügbaren GPUs genommen), und Sie können auch das Argument `parameter_devices` nutzen, um das Device zu definieren, auf dem Sie die Parameter ablegen möchten (standardmäßig wird die CPU genommen oder die GPU, falls es nur eine gibt).

Schauen wir jetzt, wie wir ein Modell in einem Cluster aus TensorFlow-Servern trainieren können!

Ein Modell in einem TensorFlow-Cluster trainieren

Ein *TensorFlow-Cluster* ist eine Gruppe von TensorFlow-Prozessen, die parallel laufen, meist auf verschiedenen Maschinen, und die miteinander kommunizieren, um Aufgaben zu erledigen – zum Beispiel ein Training oder das Ausführen eines neuronalen Netzes. Jeder TF-Prozess im Cluster wird als *Task* oder *TF-Server* bezeichnet. Er hat eine IP-Adresse, einen Port und einen Typ (auch als seine *Rolle* oder sein *Job* bezeichnet). Der Typ kann entweder "worker", "chief", "ps" (Parameterserver) oder "evaluator" sein:

- Jeder *Worker* führt Berechnungen durch – meist auf einer Maschine mit einer oder mehreren GPUs.
- Der *Chief* führt ebenfalls Berechnungen durch (es ist ein Worker), aber er erledigt noch zusätzliche Aufgaben, wie das Schreiben von TensorBoard-Logs oder das Sichern von Checkpoints. Es gibt nur einen Chief in einem Cluster. Ist kein Chief angegeben, wird der erste Worker zum Chief.
- Ein *Parameterserver* kümmert sich lediglich um die Variablenwerte – meist ist es ein reiner CPU-Rechner ohne GPUs. Diese Art von Tasks kommt nur zusammen mit der `ParameterServerStrategy` zum Einsatz.
- Ein *Evaluator* kümmert sich offensichtlich um die Auswertung.

Um ein TensorFlow-Cluster zu starten, müssen Sie es erst spezifizieren. Sie müssen also die IP-Adressen, TCP-Ports und Typen jedes Tasks definieren. So definiert beispielsweise die folgende *Cluster-Spezifikation* ein Cluster mit drei Tasks (zwei Worker und ein Parameterserver; siehe [Abbildung 19-21](#)). Die Cluster-Spec ist ein Dictionary mit einem Schlüssel pro Job, während die Werte Listen mit Task-Adressen sind (*IP: Port*):

```
cluster_spec = {
    "worker": [
        "machine-a.example.com:2222",  # /job:worker/task:0
        "machine-b.example.com:2222"   # /job:worker/task:1
    ]
}
```

```

    ],
    "ps": ["machine-a.example.com:2221"] # /job:ps/task:0
}

```

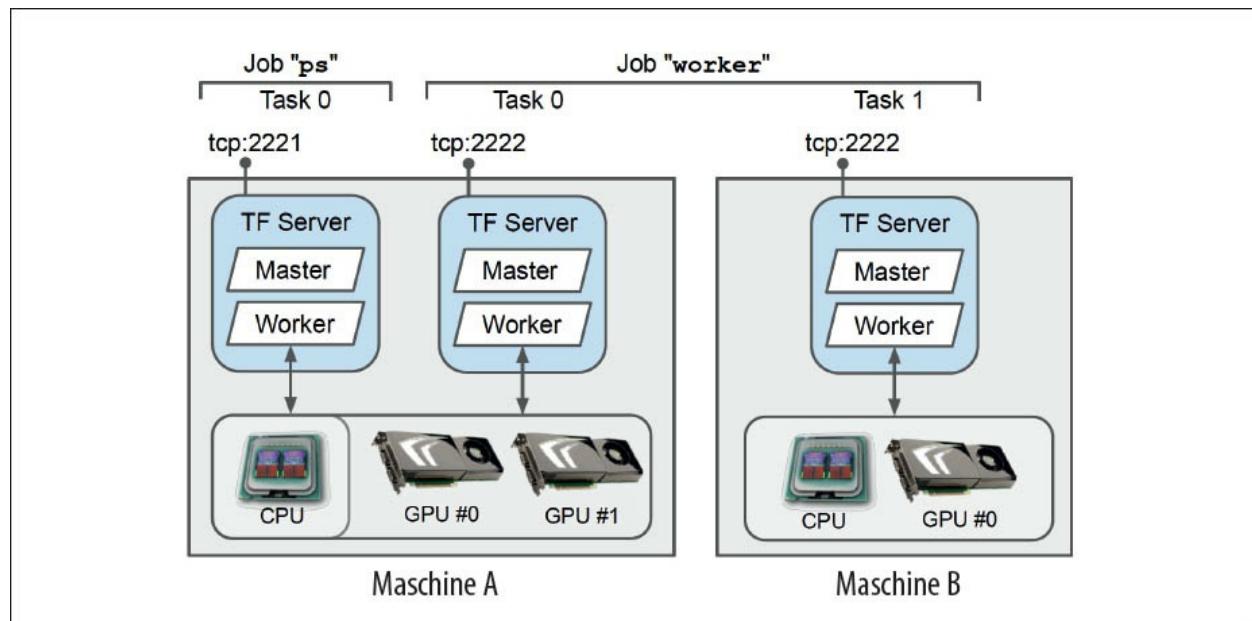


Abbildung 19-21: TensorFlow-Cluster

Im Allgemeinen gibt es einen einzelnen Task pro Maschine, aber wie dieses Beispiel zeigt, können Sie auch mehrere Tasks auf einer Maschine konfigurieren, wenn Sie das möchten (teilen sie sich die gleichen GPUs, achten Sie darauf, das RAM passend aufzuteilen, wie wir das weiter oben besprochen haben).

- ☞ Standardmäßig kann jeder Task im Cluster mit jedem anderen Task kommunizieren, daher sollten Sie dafür sorgen, dass Ihre Firewall so konfiguriert ist, dass die Kommunikation zwischen den Maschinen über die entsprechenden Ports möglich ist (es ist normalerweise einfacher, auf jeder Maschine den gleichen Port zu verwenden).

Starten Sie einen Task, müssen Sie ihm die Cluster-Spec übergeben und ihm auch noch sagen, was für ein Typ er ist und welchen Index er hat (zum Beispiel Worker 0). Am einfachsten spezifizieren Sie alles auf einmal (sowohl die Cluster-Spec als auch Typ und Index des aktuellen Tasks), indem Sie die Umgebungsvariable `TF_CONFIG` vor dem Starten von TensorFlow setzen. Sie muss ein als JSON geschriebenes Dictionary mit einer Cluster-Spezifikation (unter dem Schlüssel "cluster") sowie Typ und Index des aktuellen Tasks (unter dem Schlüssel "task") enthalten. Die folgende Umgebungsvariable `TF_CONFIG` nutzt beispielsweise das oben definierte Cluster und legt fest, dass der zu startende Task der erste Worker ist:

```

import os
import json

```

```

os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})

```



Normalerweise werden Sie die Umgebungsvariable `TF_CONFIG` außerhalb von Python definieren wollen, sodass der Code nicht Typ und Index des aktuellen Tasks enthalten muss (so lässt sich der gleiche Code auf allen Workern verwenden).

Trainieren wir jetzt ein Modell auf einem Cluster! Wir beginnen mit der Spiegel-Strategie – sie ist erstaunlich einfach umzusetzen! Als Erstes müssen Sie die Umgebungsvariable `TF_CONFIG` passend für jeden Task setzen. Es sollte keinen Parameterserver geben (entfernen Sie den Schlüssel "ps" aus der Cluster-Spec), und im Allgemeinen wollen Sie einen einzelnen Worker pro Maschine nutzen. Achten Sie ganz besonders darauf, unterschiedliche Task-Indizes für die Tasks zu setzen. Schließlich starten Sie den folgenden Trainingscode auf jedem Worker:

```

distribution = tf.distribute.experimental.MultiWorkerMirroredStrategy()

with distribution.scope():
    mirrored_model = tf.keras.Sequential([...])
    mirrored_model.compile(...)

batch_size = 100 # muss durch die Anzahl an Repliken teilbar sein
history = mirrored_model.fit(X_train, y_train, epochs=10)

```

Ja, das ist genau der Code, den wir schon zuvor genutzt haben, nur dass wir dieses Mal die `MultiWorkerMirroredStrategy` verwenden (in zukünftigen Versionen wird die `MirroredStrategy` vermutlich sowohl mit einzelnen Maschinen als auch mit Clustern aus mehreren Maschinen umgehen können). Starten Sie dieses Skript auf den ersten Workern, werden sie im AllReduce-Schritt aufgehalten werden, aber sobald der letzte Worker gestartet wurde, geht das Training los, und Sie werden sehen, dass sie alle mit der gleichen Rate voranschreiten (da sie sich bei jedem Schritt synchronisieren).

Sie können für diese Verteilungsstrategie unter zwei AllReduce-Implementierungen wählen: ein Ring-AllReduce-Algorithmus, der auf gRPC für die Netzwerkkommunikation basiert, und die NCCL-Implementierung. Es hängt von der Anzahl der Worker, der Menge und Art von GPUs und dem Netzwerk ab, welcher Algorithmus der beste ist. Standardmäßig wendet TensorFlow ein paar Heuristiken an, um den richtigen Algorithmus auszuwählen. Wollen Sie einen Algorithmus erzwingen, übergeben Sie an den Konstruktor der Strategie wahlweise

`CollectiveCommunication.RING` oder `CollectiveCommunication.NCCL` (aus `tf.distribute.experimental`).

Möchten Sie asynchrone parallelisierte Daten mit Parameterservern implementieren, wechseln Sie die Strategie zu `ParameterServerStrategy`, ergänzen einen oder mehrere Parameterserver und konfigurieren `TF_CONFIG` entsprechend für jeden Task. Beachten Sie, dass zwar die Worker asynchron, die Repliken auf jedem Worker aber synchron arbeiten werden.

Haben Sie Zugriff auf TPUs auf Google Cloud (<https://cloud.google.com/tpu/>), können Sie eine `TPUStrategy` erstellen (und sie dann wie die anderen Strategien verwenden):

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()  
  
tf.tpu.experimental.initialize_tpu_system(resolver)  
  
tpu_strategy = tf.distribute.experimental.TPUStrategy(resolver)
```



Sind Sie Forscher, kann es sein, dass Sie kostenlos TPUs nutzen dürfen – unter <https://tensorflow.org/tfrc> finden Sie weitere Informationen.

Sie können jetzt Modelle über mehrere GPUs und Server hinweg trainieren – dafür dürfen Sie sich gerne selbst loben! Wollen Sie ein großes Modell trainieren, werden Sie viele GPUs auf vielen Servern benötigen, wofür Sie entweder viel Hardware kaufen oder viele Cloud-VMs managen müssen. Häufig wird es weniger aufwendig und teuer sein, einen Cloud-Service einzusetzen, der sich für Sie um das Provisionieren und Managen all dieser Infrastruktur kümmert, wenn Sie sie brauchen. Schauen wir, wie wir das mit GCP machen können.

Große Trainingsjobs auf der Google Cloud AI Platform ausführen

Wenn Sie sich dazu entschließen, Google AI Platform zu verwenden, können Sie einen Trainingsjob mit dem gleichen Trainingscode deployen, der auch in Ihrem TF-Cluster laufen würde – die Plattform kümmert sich um das Provisionieren und Konfigurieren von so vielen GPUs, wie Sie möchten (und Ihre Quota hergibt).

Um den Job zu starten, benötigen Sie das Befehlszeilentool `gcloud`, das Teil des Google Cloud SDK (<https://cloud.google.com/sdk/>) ist. Sie können das SDK entweder auf Ihrem eigenen Rechner installieren oder einfach die Google Cloud Shell auf GCP verwenden. Dabei handelt es sich um ein Terminal, das Sie direkt in Ihrem Webbrowser einsetzen können – es läuft auf einer freien Linux-VM (Debian), auf der das SDK schon installiert und für Sie konfiguriert wurde. Die Cloud Shell steht überall in GCP zur Verfügung: Klicken Sie einfach oben rechts auf der Seite auf das Symbol für »Cloud Shell aktivieren« (siehe [Abbildung 19-22](#)).



Abbildung 19-22: Die Google Cloud Shell aktivieren

Ist es Ihnen lieber, das SDK auf Ihrem Rechner zu installieren, müssen Sie es nach der Installation initialisieren, indem Sie `gcloud init` aufrufen: Sie melden sich am GCP an und erlauben den Zugriff auf Ihre GCP-Ressourcen, danach wählen Sie das GCP-Projekt aus, das Sie verwenden wollen (wenn Sie mehr als eines haben), dazu die Region, in der der Job laufen soll. Der Befehl `gcloud` ermöglicht Ihnen Zugriff auf jedes GCP-Feature, einschließlich derer, die wir schon verwendet haben. Sie müssen nicht jedes Mal über die Weboberfläche gehen, stattdessen können Sie Skripte schreiben, die VMs für Sie starten oder stoppen, Modelle deployen oder beliebige andere GCP-Aktivitäten ausführen.

Bevor Sie den Trainingsjob ausführen können, müssen Sie den Trainingscode schreiben – genau so, wie Sie es zuvor schon für ein verteiltes Setup getan haben (also zum Beispiel die `ParameterServerStrategy` verwenden). AI Platform kümmert sich darum, die `TF_CONFIG` für Sie in jeder VM zu setzen. Ist das geschehen, können Sie das Programm deployen und auf einem TF-Cluster mit einem Befehl wie dem folgenden starten:

```
$ gcloud ai-platform jobs submit training my_job_20190531_164700 \
  --region asia-southeast1 \
  --scale-tier PREMIUM_1 \
  --runtime-version 2.0 \
  --python-version 3.5 \
  --package-path /my_project/src/trainer \
  --module-name trainer.task \
  --staging-bucket gs://my-staging-bucket \
  --job-dir gs://my-mnist-model-bucket/trained_model \
  --
  --my-extra-argument1 foo --my-extra-argument2 bar
```

Gehen wir die Optionen durch. Der Befehl startet einen Trainingsjob namens `my_job_20190531_164700` in der Region `asia-southeast1` mit einem `PREMIUM_1 Scale`

Tier: Das entspricht 20 Workern (einschließlich einem Chief) und 11 Parameterservern (schauen Sie sich auch die anderen verfügbaren Scale Tier (https://homl.info/scale_tiers) an). All diese VMs basieren auf der AI Platform 2.0 Runtime (einer VM-Konfiguration, die TensorFlow 2.0 und diverse andere Pakete enthält)²² und Python 3.5. Der Trainingscode befindet sich im Verzeichnis `/my_project/src/trainer`, und der Befehl `gcloud` wird ihn automatisch in einem pip-Paket verpacken und nach GCS unter `gs://my-staging-bucket` hochladen. Als Nächstes startet AI Platform eine Reihe von VMs, deployt das Paket dorthin und führt das Modul `trainer.task` aus. Schließlich werden das Argument `--job-dir` und alle zusätzlichen Argumente (also alle nach dem Trenner `--`) an das Trainingsprogramm weitergereicht: Der Chief-Task nutzt im Allgemeinen das Argument `--job-dir`, um herauszufinden, wo das fertige Modell in GCS gespeichert werden soll – in diesem Fall unter `gs://mymnist-model-bucket/trained_model`. Und das ist alles! In der GCP-Konsole können Sie dann das Navigationsmenü öffnen, zum Bereich »Künstliche Intelligenz« scrollen und »AI Platform → Jobs« anklicken. Sie sollten den laufenden Job sehen, und wenn Sie ihn anklicken, erhalten Sie Diagramme mit der CPU-, GPU- und RAM-Auslastung für jeden Task. Klicken Sie auf »Logs ansehen«, um mithilfe von Stackdriver Zugriff auf die detaillierten Logs zu erhalten.



Legen Sie die Trainingsdaten auf GCS ab, können Sie ein `tf.data.TextLineDataset` oder ein `tf.data.TFRecordDataset` erstellen, um darauf zuzugreifen: Nutzen Sie einfach die GCS-Pfade als Dateinamen (zum Beispiel `gs://my-data-bucket/my_data_001.csv`). Diese Datasets verwenden das Paket `tf.io.gfile`, um auf Dateien zuzugreifen: Es unterstützt sowohl lokale Dateien wie auch GCS-Dateien (achten Sie aber darauf, dass das verwendete Dienstkonto auch Zugriff auf GCS hat).

Wollen Sie die Werte von ein paar Hyperparametern austesten, können Sie einfach mehrere Jobs starten und die Hyperparameter als zusätzliche Argumente für Ihre Tasks mitgeben. Wollen Sie aber effizient viele Hyperparameter durchprobieren, ist es sinnvoll, den Hyperparameter-Tuning-Service der AI Platform einzusetzen.

Black Box Hyperparameter Tuning auf der AI Platform

Die AI Platform stellt einen leistungsfähigen Hyperparameter-Tuning-Service bereit, der auf bayesscher Optimierung beruht. Er nennt sich Google Vizier (<https://homl.info/vizier>).²³ Um ihn zu verwenden, müssen Sie beim Erstellen des Jobs eine YAML-Konfigurationsdatei mitgeben (`--config tuning.yaml`). Die kann zum Beispiel so aussehen:

```
trainingInput:  
  
hyperparameters:  
  
goal: MAXIMIZE  
  
hyperparameterMetricTag: accuracy  
  
maxTrials: 10  
  
maxParallelTrials: 2
```

```

params:
  - parameterName: n_layers
    type: INTEGER
    minValue: 10
    maxValue: 100
    scaleType: UNIT_LINEAR_SCALE
  - parameterName: momentum
    type: DOUBLE
    minValue: 0.1
    maxValue: 1.0
    scaleType: UNIT_LOG_SCALE

```

Damit wird AI Platform mitgeteilt, dass wir die Metrik namens "accuracy" maximieren wollen, der Job maximal 10 Trials ausführen soll (jeder Trial führt unseren Trainingscode zum Trainieren des Modells von vorne aus) und dass maximal zwei Trials parallel laufen. Wir wollen zwei Hyperparameter tunen: den Hyperparameter `n_layers` (eine Ganzzahl zwischen 10 und 100) sowie den Hyperparameter `momentum` (eine Gleitkommazahl zwischen 0,1 und 1,0). Das Argument `scaleType` legt den Prior für den Hyperparameter-Wert fest: `UNIT_LINEAR_SCALE` steht für einen flachen Prior (also keine A-Priori-Präferenz), `UNIT_LOG_SCALE` besagt, dass wir davon ausgehen, dass der optimale Wert näher am maximalen Wert liegt (der andere mögliche Prior ist `UNIT_REVERSE_LOG_SCALE`, wenn wir glauben, dass der optimale Wert nahe am minimalen Wert liegt).

Die Argumente `n_layers` und `momentum` werden als Befehlszeilenargumente an den Trainingscode übergeben, und natürlich wird erwartet, dass er sie auch nutzt. Die Frage ist, wie der Trainingscode die Metrik zurück an die AI Platform überträgt, sodass diese entscheiden kann, welche Hyperparameter-Werte während des nächsten Trials verwendet werden soll. Nun, AI Platform überwacht einfach das Ausgabeverzeichnis (definiert über `--job-dir`) auf eine Eventdatei (siehe [Kapitel 10](#)) mit der Zusammenfassung für eine Metrik namens "accuracy" (oder welcher Metrikname ansonsten als `hyperparameterMetricTag` spezifiziert wurde) und liest deren Wert aus. Ihr Trainingscode muss also einfach nur den Callback `TensorBoard()` verwenden (was Sie zum Monitoring sowieso machen werden).

Ist der Job erledigt, stehen alle Hyperparameter-Werte, die in jedem Trial verwendet wurden, und die sich daraus ergebende Genauigkeit in der Ausgabe des Jobs zur Verfügung (die Sie über die Seite »AI Platform → Jobs« erreichen).



AI Platform Jobs können auch genutzt werden, um Ihre Modell effizient mit großen Datenmengen auszuführen: Jeder Worker kann einen Teil der Daten von GCS lesen,

Vorhersagen treffen und diese wieder in GCS speichern.

Jetzt haben Sie alle Tools und Voraussetzungen zusammen, um neuronale Netzarchitekturen zu erstellen, die State of the Art sind, und diese im großen Umfang zu trainieren, wobei diverse Verteilungsstrategien zum Einsatz kommen – auf Ihrer eigenen Infrastruktur oder in der Cloud. Und Sie können sogar eine leistungsfähige bayessche Optimierung nutzen, um die Hyperparameter anzupassen!

Übungen

1. Was enthält ein SavedModel? Wie untersuchen Sie dessen Inhalte?
2. Wann sollten Sie TF Serving verwenden? Was sind seine wichtigsten Features? Welche Tools können Sie zum Deployen einsetzen?
3. Wie deployen Sie ein Modell auf mehrere TF-Serving-Instanzen?
4. Wann sollten Sie lieber die gRPC-API statt der REST-API nutzen, um ein Modell abzufragen, das durch TF Serving bereitgestellt wird?
5. Auf welchen Wegen kann TFLite die Größe eines Modells reduzieren, damit es auf einem mobilen Gerät oder einem Embedded Device laufen kann?
6. Was ist quantisierungsbewusstes Training, und warum könnten Sie es benötigen?
7. Was sind parallelisierte Modelle und was parallelisierte Daten? Warum wird Letzteres eher empfohlen?
8. Was für Verteilungsstrategien können Sie einsetzen, wenn Sie ein Modell auf mehreren Servern trainieren? Wie entscheiden Sie, welche Sie verwenden?
9. Trainieren Sie ein Modell (suchen Sie sich eines aus) und deployen Sie es auf TF Serving oder die Google Cloud AI Platform. Schreiben Sie den Clientcode, um es über die REST-API oder die gRPC-API abzufragen. Aktualisieren Sie das Modell und deployen Sie die neue Version. Ihr Clientcode wird nun die neue Version verwenden. Rollen Sie zurück zur ersten Version.
10. Trainieren Sie ein Modell auf mehreren GPUs auf dem gleichen Rechner mithilfe der MirroredStrategy (wenn Sie nicht an GPUs herankommen, können Sie Colaboratory mit einer GPU Runtime nutzen und zwei virtuelle GPUs erstellen). Trainieren Sie das Modell erneut mit der CentralStorageStrategy und vergleichen Sie die Trainingsdauer.
11. Trainieren Sie ein kleines Modell auf der Google Cloud AI Platform und nutzen Sie dabei Black Box Hyperparameter Tuning.

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 Buchs hatten wie ich beim Schreiben 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, das GitHub-Projekt [ageron/handson-ml2](https://github.com/ageron/handson-ml2) oder auf Twitter über @aureliengeron 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](https://www.kaggle.com) oder einer anderen ML-Community bei. Schauen Sie sich ML-Kurse an, lesen Sie Fachartikel, besuchen Sie Konferenzen und treffen Sie Experten. Es hilft auch ungemein, ein konkretes Projekt zu haben, an dem Sie arbeiten – egal ob es für die Arbeit oder aus Spaß ist (idealerweise beides). Gibt es also etwas, das Sie schon immer einmal bauen wollten, versuchen Sie es! Arbeiten Sie inkrementell – versuchen Sie nicht gleich, den Himmel zu erreichen, sondern konzentrieren Sie sich auf Ihr Projekt und bauen Sie es Schritt für Schritt aus. Sie brauchen dafür Geduld und Durchhaltevermögen, aber wenn Sie einen laufenden Roboter, einen funktionierenden Chatbot oder etwas anderes Cooles haben, hat es sich gelohnt!

Meine Hoffnung ist, dass dieses Buch Sie zum Entwickeln einer ML-Anwendung befähigt, die uns allen nützt! Was wird es wohl für eine sein?

Aurélien Géron

Lösungen zu den Übungsaufgaben



Lösungen zu allen Programmierübungen sind online in den Jupyter-Notebooks auf <https://github.com/ageron/hanson-ml2> 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 geeignet zum Lösen komplexer Aufgaben, 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 wahrscheinlich am besten, wenn ein Roboter lernen soll, in unbekanntem Gelände zu laufen, da dies typischerweise die Art von Problem ist, die das Reinforcement Learning angeht. 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. Kennen Sie die gewünschten Gruppen dagegen bereits, können Sie einem Klassifikationsalgorithmus viele Beispiele aus jeder Gruppe zeigen (überwachtes Lernen) und alle Kunden in diese Gruppen einordnen lassen.
8. Spamerkennung ist eine typische überwachte Lernaufgabe: Dem Algorithmus werden viele E-Mails und deren Labels (Spam oder Nicht-Spam) bereitgestellt.
9. Ein Onlinelernsystem kann im Gegensatz zu einem Batchlernsystem 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 mehrere 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 verallgemeinern kann. Ein Hyperparameter ist ein Parameter des Lernalgorithmus selbst und 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 Fall 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 ein Overfitting der Trainingsdaten vor (oder wir hatten bei den Trainingsdaten eine Menge Glück). Gegenmaßnahmen zum Overfitting sind das Beschaffen zusätzlicher Daten, das Vereinfachen des Modells (Auswählen eines einfacheren Algorithmus, 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. Das Train-Dev-Set wird eingesetzt, wenn das Risiko besteht, dass es eine Diskrepanz zwischen den Trainingsdaten und den in den Validierungs- und Testdatensätzen verwendeten Daten gibt (die immer so nahe wie möglich an den Daten liegen sollten, die das Modell produktiv nutzen). Es ist Teil des Trainingsdatensatzes, der zurückgehalten wird (das Modell wird nicht damit trainiert). Stattdessen wird das Modell mit dem Rest des Trainingsdatensatzes trainiert und sowohl mit dem Train-Dev-Set wie auch mit dem

Validierungsdatensatz evaluiert. Funktioniert das Modell mit dem Trainingsdatensatz gut, nicht aber mit dem Train-Dev-Set, ist es vermutlich für den Trainingsdatensatz overfittet. Funktioniert es gut dem Trainingsdatensatz und dem Train-Dev-Set, aber nicht mit dem Validierungsdatensatz, gibt es vermutlich einen signifikanten Unterschied zwischen Trainingsdaten einerseits und Validierungs- und Testdaten andererseits, und Sie sollten versuchen, die Trainingsdaten zu verbessern, damit diese mehr wie die Validierungs- und Testdaten aussehen.

19. 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).

Kapitel 2: Ein Machine-Learning-Projekt von A bis Z

Die Lösungen finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 3: Klassifikation

Die Lösungen finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 4: Trainieren von Modellen

1. Haben Sie einen Trainingsdatensatz mit Millionen Merkmalen, 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 und auch der SVD-Ansatz funktionieren 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 und der SVD-Ansatz funktionieren 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. Sollte der Validierungsfehler nach jeder Epoche immer wieder steigen, 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 über eine längere Zeit nicht verbessert (es also vermutlich den eigenen Rekord nicht knacken wird).
7. Die Trainingsiterationen sind beim stochastischen Gradientenverfahren am schnellsten, da dieses 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 Freiheitsgraden 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 ebenfalls 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 sind, liegt vermutlich ein 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.
 - 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. Sind Sie sich nicht sicher, 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.
- Möchten Sie Bilder als außen/innen und Tag/Nacht klassifizieren, schließen sich die Kategorien nicht gegenseitig aus (d.h., alle vier Kombinationen sind möglich). Sie sollten daher zwei Klassifikatoren mit logistischer Regression trainieren.
 - Die Lösungen finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 5: Support Vector Machines

- Der Grundgedanke bei Support Vector Machines ist, die breitestmögliche »Straße« zwischen den Kategorien zu fitten. Anders ausgedrückt, soll zwischen der Entscheidungsgrenze zwischen den beiden Kategorien und den Trainingsdatenpunkten eine möglichst große Lücke sein. Bei der Soft-Margin-Klassifikation sucht die 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.
- Nach dem Trainieren einer SVM ist jeder Datenpunkt an der »Straße« (siehe vorherige 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.
- SVMs versuchen, die breitestmögliche »Straße« zwischen den Kategorien einzufügen (siehe erste Antwort). Wenn also die Trainingsdaten nicht skaliert sind, neigt die SVM dazu, kleine Merkmale zu ignorieren (siehe Abbildung 5-2).
- Ein SVM-Klassifikator kann den Abstand zwischen einem Testdatenpunkt und der Entscheidungsgrenze ausgeben, und Sie können diesen als Konfidenzmaß interpretieren. Allerdings lässt sich dieser Score nicht direkt in eine Schätzung der Wahrscheinlichkeit einer Kategorie umrechnen. Wenn Sie beim Erstellen einer SVM in Scikit-Learn `probability=True` einstellen, werden nach dem Training die Wahrscheinlichkeiten mithilfe einer logistischen Regression auf die Scores der SVM kalibriert (trainiert durch eine zusätzliche fünffache Kreuzvalidierung der Trainingsdaten). Damit erhalten Sie auch für eine SVM die Methoden `predict_proba()` und `predict_log_proba()`.
- Diese Frage betrifft lediglich lineare SVMs, da Kernel-SVMs nur die duale Form verwenden können. Die Komplexität der Berechnung der primalen 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 primale 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 »Quadratische Programme«). 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: $\mathbf{H} = \begin{pmatrix} \mathbf{H}' & \mathbf{0} & \cdots \\ \mathbf{0} & \mathbf{0} & \\ \vdots & & \ddots \end{pmatrix}$
 - \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-ml2>.

Kapitel 6: Entscheidungsbäume

1. Die Tiefe eines ausbalancierten Binärbaums mit m Blättern ist $\log_2(m)^2$, aufgerundet. Ein binärer Entscheidungsbau (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-Trainingsalgorithmus 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 einem aus Kategorie B. Dessen Gini-Unreinheit beträgt $1 - \left(\frac{1}{5}\right)^2 - \left(\frac{4}{5}\right)^2 = 0,32$.

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 Kindes beträgt $1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^2 = 0,5$, was höher als die des Elternknotens ist. Dies wird dadurch kompensiert, dass 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 das 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 100.000 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-ml2>.

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 sich die Modelle 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 wird auch ohne effektiv sein, 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 vorherigen 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 vorherige 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-ml2>.

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 höher dimensionalen Räumen viele Schwierigkeiten auftreten, die es bei weniger Dimensionen nicht gibt. Beim Machine Learning ist eine häufige Erscheinungsform, dass zufällig ausgewählte höher dimensionale 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 verwerfen 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 bestünde 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 1.000 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 Onlineaufgaben 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; sofern 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-ml2>.

Kapitel 9: Techniken des unüberwachten Lernens

1. Im Machine Learning ist Clustering die unüberwachte Aufgabe, ähnliche Instanzen zu gruppieren. Der Begriff der Ähnlichkeit hängt von der aktuellen Aufgabe ab: In manchen Fällen werden zwei nahe beieinanderliegende Instanzen als ähnlich angesehen, während in anderen ähnliche Instanzen weit voneinander entfernt sein können, solange sie zur gleichen dicht gepackten Gruppe gehören. Zu den verbreiteten Clustering-Algorithmen gehören K-Means, DBSCAN, agglomeratives Clustern, BIRCH, Mean-Shift, Affinity Propagation und spektrales Clustern.
2. Zu den Hauptanwendungsgebieten von Clustering-Algorithmen gehören Datenanalyse, Kundensegmentierung, Empfehlungssysteme, Suchmaschinen, Bildsegmentierung, teilüberwachtes Lernen, Dimensionsreduktion, Anomalieerkennung und Novelty Detection.
3. Die Ellenbogenregel ist eine einfache Technik, um beim Einsatz von K-Means die Anzahl an Clustern festzulegen: Tragen Sie einfach die Trägheit (den mittleren quadratischen Abstand jeder Instanz zu ihrem nächstgelegenen Schwerpunkt) gegen die Anzahl der Cluster auf und finden Sie den Punkt der Kurve, bei dem die Trägheit nicht mehr schnell fällt (der »Ellenbogen«). Das liegt im Allgemeinen nahe an der optimalen Anzahl an Clustern. Eine andere Möglichkeit ist, den Silhouettenkoeffizienten als Funktion der Anzahl von Clustern auszugeben. Es gibt oft ein Maximum, und die optimale Anzahl an Clustern liegt meist dort in der Nähe. Der Silhouettenkoeffizient ist der Mittelwert der Silhouetten

aller Instanzen. Dieser Wert liegt zwischen +1 für Instanzen, die gut in ihren Clustern und fern anderer Cluster liegen, und -1 für Instanzen, die sich sehr nahe an anderen Clustern befinden. Sie können auch die Silhouettendiagramme ausgeben und eine umfassendere Analyse durchführen.

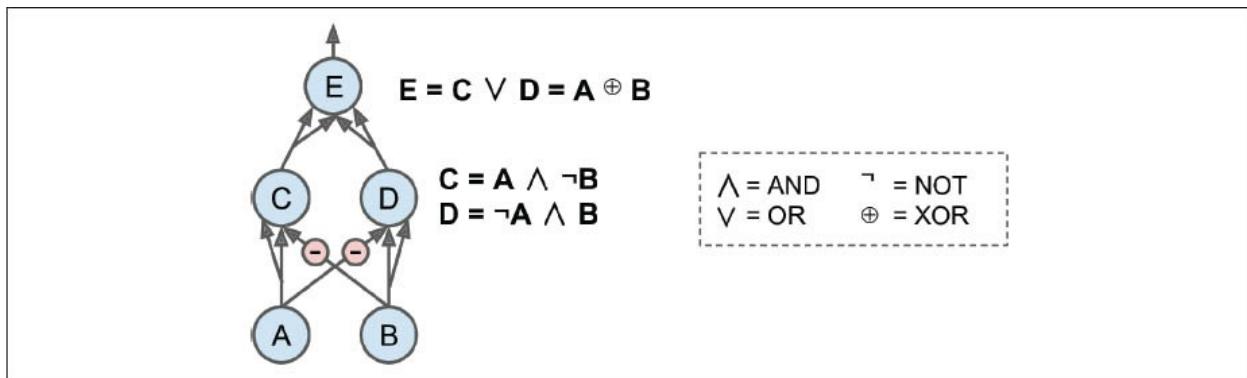
4. Es ist teuer und zeitaufwendig, einen Datensatz mit Labels auszustatten. Daher hat man häufig viele ungelabelte Instanzen und nur wenige mit Labels. Label Propagation ist eine Technik, bei der manche (oder alle) Labels von den gelabelten auf ähnliche ungelabelte Instanzen übertragen werden. Das kann die Menge an gelabelten Instanzen deutlich vergrößern und es damit einem überwachten Algorithmus ermöglichen, eine bessere Performance zu liefern (das ist eine Form des teilüberwachten Lernens). Ein Ansatz ist die Verwendung eines Clustering-Algorithmus wie K-Means für alle Instanzen. Danach wird für jedes Cluster das verbreitetste Label oder das der repräsentativsten Instanz genutzt (zum Beispiel der Instanz, die am nächsten am Schwerpunkt liegt) und auf die ungelabelten Instanzen des gleichen Clusters übertragen.
5. K-Means und BIRCH skalieren sehr gut bei großen Datensätzen. DBSCAN und Mean-Shift suchen nach Regionen mit hoher Dichte.
6. Aktives Lernen ist immer dann nützlich, wenn Sie viele ungelabelte Instanzen haben, das Labeln aber teuer ist. In diesem (recht häufig vorkommenden) Fall ist es oft besser, nicht zufällig ausgewählte Instanzen mit Labels zu versehen, sondern ein aktives Lernen durchzuführen, bei dem menschliche Experten mit dem Lernalgorithmus interagieren und Labels für spezifische Instanzen liefern, wenn der Algorithmus diese anfordert. Häufig wird dazu Uncertainty Sampling eingesetzt (siehe die Beschreibung in »»).
7. Die Begriffe *Anomalieerkennung* und *Novelty Detection* werden oft für das Gleiche verwendet, auch wenn sie nicht ganz genau das Gleiche beschreiben. Bei der Anomalieerkennung wird der Algorithmus mit einem Datensatz trainiert, der Ausreißer enthalten kann. Das Ziel ist normalerweise, diese Ausreißer (im Trainingsdatensatz) und solche in neuen Instanzen zu identifizieren. Bei der Novelty Detection wird der Algorithmus mit einem Datensatz trainiert, der als »sauber« angenommen wird. Ziel ist hier, Ausreißer (Novelties) nur in neuen Instanzen zu finden. Manche Algorithmen funktionieren am besten bei der Anomalieerkennung (zum Beispiel Isolation Forest), andere sind besser zur Novelty Detection geeignet (zum Beispiel One-Class-SVMs).
8. Ein gaußsches Mischverteilungsmodell (GMM) ist ein Wahrscheinlichkeitsmodell, das annimmt, dass die Instanzen aus einer Mischung mehrerer Gaußverteilungen erzeugt wurden, deren Parameter unbekannt sind. Oder anders gesagt: Es wird davon ausgegangen, dass die Daten in einer endlichen Zahl von Clustern gruppiert sind, von denen jedes eine ellipsoide Form hat (das aber je nach Cluster eine andere Größe, Ausrichtung und Dichte haben kann), und wir wissen nicht, zu welchem Cluster jede Instanz gehört. Dieses Modell ist für die Dichteabschätzung, das Clustering und die Anomalieerkennung nützlich.
9. Sie können die richtige Zahl an Clustern mit einem gaußschen Mischverteilungsmodell unter anderem herausfinden, indem Sie das bayessche Informationskriterium (BIC) oder das Akaike-Informationskriterium (AIC) als Funktion der Anzahl von Clustern auftragen und dann die Zahl wählen, die das BIC oder AIC minimiert. Eine andere Technik ist der Einsatz eines bayesschen gaußschen Mischverteilungsmodells, das automatisch die Zahl der Cluster

bestimmt.

Die Lösungen zu den Übungen 10 bis 13 finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 10: Einführung in künstliche neuronale Netze mit Keras

1. Suchen Sie den TensorFlow Playground (<https://playground.tensorflow.org/>) auf und spielen Sie mit ihm herum (wie in der Übung beschrieben).
2. 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.



3. Ein klassisches Perzeptron konvergiert nur, wenn der Datensatz linear separierbar ist, und es ist nicht in der Lage, die Wahrscheinlichkeiten für Kategorien 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 Perzeptrons 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.
4. Die logistische Aktivierungsfunktion war ein Hauptbestandteil beim Trainieren der ersten MLPs, da ihre Ableitung immer 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.
5. Häufig eingesetzte Aktivierungsfunktionen sind die Stufenfunktion, die logistische (Sigmoid-)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.
6. 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 Trainingsbatchs 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*.

7. 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.³
8. Backpropagation ist ein Verfahren zum Trainieren künstlicher neuronaler Netze. Es berechnet zunächst die Gradienten der Kostenfunktion nach jedem Modellparameter (alle Gewichte und Biase) 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 Trainingsbatch 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.
9. 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.⁴ 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.

10. Weitere Lösungen finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

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 SELU-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.
 - Wenn die Bedingungen stimmen (das Modell zum Beispiel sequenziell ist, die Gewichte mit der LeCun-Initialisierung aufgesetzt wurden, die Eingaben standardisiert sind und es keine inkompatiblen Schichten oder Regularisierungen gibt, wie zum Beispiel Drop-out oder ℓ_1 -Regularisierung), stellt die SELU-Aktivierungsfunktion sicher, dass das Modell selbstnormalisierend ist, was die Probleme der explodierenden/verschwindenden Gradienten löst.
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 SELU 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 17](#)). Zudem kann es manchmal von optimierten Implementierungen und von Hardwarebeschleunigung profitieren. 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 (außer in rekurrenten Netzen). 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 Autoencoder, siehe [Kapitel 17](#)). 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 SGD-Optimierer den Hyperparameter `momentum` zu nah an 1 setzen (z.B. $0,99999$), wird der Algorithmus voraussichtlich stark an Geschwindigkeit aufnehmen, sich hoffentlich in etwa auf das globale Minimum zubewegen und dann daran vorbeizischen. Dann bremst er ab, kehrt zurück, beschleunigt wieder, fliegt erneut 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, das TensorFlow Optimization Toolkit einzusetzen.
7. Ja, das Trainieren wird durch Drop-out langsamer, meist etwa um den Faktor zwei. Es hat allerdings keinen Einfluss auf die Inferenzgeschwindigkeit, da Drop-out nur beim Trainieren angeschaltet ist. MC-Drop-out verhält sich beim Training genauso, aber es ist auch während der Inferenz aktiv, daher wird jede Inferenz ein bisschen verlangsamt. Entscheidender ist, dass Sie beim Einsatz von MC-Drop-out im Allgemeinen die Inferenz zehn Mal laufen lassen wollen, um bessere Vorhersagen zu erhalten. Das heißt, dass sich das Treffen von Vorhersagen um den Faktor 10 oder mehr verlangsamt.

Die Lösung zur Übungsaufgabe 8 finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 12: Eigene Modelle und Training mit TensorFlow

1. TensorFlow ist eine Open-Source-Bibliothek für numerisches Rechnen, die besonders auf Machine Learning im großen Maßstab ausgelegt ist. Ihr Kern ähnelt NumPy, aber sie bietet zudem GPU-Unterstützung, sie ermöglicht verteiltes Rechnen, die Analyse von Rechengraphen und Optimierungsmöglichkeiten (mit einem portierbaren Graphenformat, das Ihnen erlaubt, ein TensorFlow-Modell in einer Umgebung zu trainieren und in einer anderen auszuführen), eine Optimierungs-API, die auf Autodiff im Reverse-Mode basiert, und eine Reihe leistungsfähiger APIs wie `tf.keras`, `tf.data`, `tf.image` oder `tf.signal`. Andere

beliebte Bibliotheken zum Deep Learning sind PyTorch, MXNet, Microsoft Cognitive Toolkit, Theano, Caffe2 und Chainer.

2. Auch wenn TensorFlow einen Großteil der Funktionalität von NumPy bietet, können Sie es nicht einfach austauschen: Die Namen der Funktionen sind nicht immer die gleichen (z.B. `tf.reduce_sum()` vs. `np.sum()`), manche Funktionen verhalten sich nicht genau gleich (so erzeugt `tf.transpose()` eine transponierte Kopie eines Tensors, während das Attribut `T` bei NumPy eine transponierte Sicht bietet, ohne tatsächlich Daten zu kopieren), und schließlich sind Arrays bei NumPy veränderbar, während das für die Tensoren bei TensorFlow nicht gilt (Sie können aber eine `tf.Variable` nutzen, wenn Sie ein veränderbares Objekt benötigen).
3. Sowohl `tf.range(10)` wie auch `tf.constant(np.arange(10))` geben einen eindimensionalen Tensor mit den Integer-Werten 0 bis 9 zurück. Aber beim ersten kommen 32-Bit-Ganzzahlen zum Einsatz, während im zweiten Fall 64-Bit-Werte genutzt werden. TensorFlow verwendet standardmäßig 32 Bits, während NumPy mit 64 Bits arbeitet.
4. Neben normalen Tensoren bietet TensorFlow noch viele andere Datenstrukturen, unter anderem Sparse-Tensoren, Tensor-Arrays, Ragged-Tensoren, Queues, String-Tensoren und Sets. Die letzten beiden werden sogar als normale Tensoren repräsentiert, aber TensorFlow bietet zusätzliche Funktionen zu ihrer Bearbeitung an (in `tf.strings` und `tf.sets`).
5. Wollen Sie eine eigene Verlustfunktion definieren, können Sie sie im Allgemeinen als normale Python-Funktion implementieren. Aber wenn sie Hyperparameter (oder einen anderen Status) unterstützen muss, sollten Sie eine Unterklasse von `keras.losses.Loss` erstellen und die Methoden `__init__()` und `call()` implementieren. Sollen die Hyperparameter der Verlustfunktion zusammen mit dem Modell gesichert werden, müssen Sie zudem die Methode `get_config()` implementieren.
6. Wie eigene Verlustfunktionen können die meisten Metriken als normale Python-Funktionen definiert werden. Soll Ihre Metrik auch Hyperparameter (oder einen anderen Status) unterstützen, erstellen Sie eine Unterklasse von `keras.metrics.Metric`. Ist das Berechnen der Metrik über eine ganze Epoche zudem nicht das Gleiche wie das Berechnen der mittleren Metrik über alle Batches in dieser Epoche (zum Beispiel bei Relevanz und Sensitivität), sollten Sie eine Unterklasse von `keras.metrics.Metric` bilden und die Methoden `__init__()`, `update_state()` und `result()` implementieren, um eine gleitende Metrik für jede Epoche zu erschaffen. Außerdem sollten Sie die Methode `reset_states()` implementieren, sofern nicht nur einfach alle Variablen auf 0,0 zu setzen sind. Wollen Sie den Status zusammen mit dem Modell abspeichern, sollten Sie auch noch die Methode `get_config()` implementieren.
7. Sie sollten die internen Komponenten Ihres Modells (also die Schichten oder wiederverwendbaren Blöcke mit Schichten) vom Modell selbst trennen (also das zu trainierende Objekt). Erstere sollten Unterklassen von `keras.layers.Layer` sein, Letzteres eine Unterklasse von `keras.models.Model`.
8. Es ist schon recht fortgeschritten, eine eigene Trainingsschleife zu schreiben, daher sollten Sie das nur in Angriff nehmen, wenn Sie es wirklich brauchen. Keras stellt diverse Werkzeuge zum Anpassen des Trainings bereit, ohne dass Sie eine eigene Trainingsschleife

schreiben müssen: Callbacks, eigene Regularisierer, Constraints, Verluste und so weiter. Sie sollten diese wann immer möglich verwenden: Durch das Schreiben einer eigenen Trainingsschleife fangen Sie sich eher Fehler ein, außerdem wird sich der Code schlechter wiederverwenden lassen. Aber manchmal ist das Schreiben einer eigenen Trainingsschleife notwendig – wenn Sie zum Beispiel verschiedene Optimierer für unterschiedliche Teile Ihres neuronalen Netzes verwenden wollen (wie im Wide-&-Deep-Artikel (<https://homl.info/widedeep>)). Sie kann auch beim Debuggen nützlich sein oder wenn Sie verstehen wollen, wie das Training genau abläuft.

9. Eigene Keras-Komponenten sollten in TF Functions konvertierbar sein – sie sollten sich also so weit wie möglich an TF-Operationen und alle in »Regeln für TF Functions« aufgeführten Regeln halten. Müssen Sie unbedingt anderen Python-Code in einer eigenen Komponente verwenden, können Sie ihn entweder in einer `tf.py_function()`-Operation verpacken (das wird aber die Performance verringern und die Portierbarkeit Ihres Modells einschränken) oder beim Erstellen der eigenen Schicht oder des Modells `dynamic=True` setzen (oder die Modellmethode `compile()` mit `run_eagerly=True` aufrufen).
10. In »Regeln für TF Functions« finden Sie die Liste mit den Regeln für das Erstellen einer TF Function.
11. Es kann zum Debuggen nützlich sein, ein dynamisches Keras-Modell zu erstellen, da dann keine eigenen Komponenten in eine TF Function umgewandelt werden und Sie Ihren Code mit einem beliebigen Python-Debugger untersuchen können. Außerdem ist es praktisch, wenn Sie beliebigen Python-Code oder auch Aufrufe externer Bibliotheken in Ihrem Modell (oder im Trainingscode) einsetzen wollen. Um ein Modell dynamisch zu machen, müssen Sie beim Erstellen `dynamic=True` setzen. Alternativ können Sie `run_eagerly=True` setzen, wenn Sie die Methode `compile()` des Modells aufrufen. Wenn Sie ein Modell dynamisch machen, halten Sie Keras davon ab, TensorFlow-Graphen zu verwenden. Damit werden Training und Inferenz langsamer, und Sie haben nicht die Möglichkeit, den Rechengraphen zu exportieren, wodurch Sie die Portierbarkeit Ihres Modells einschränken.

Die Lösungen zu den Übungsaufgaben 12 und 13 finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 13: Daten mit TensorFlow laden und vorverarbeiten

1. Das Einlesen eines großen Datensatzes und seine effiziente Vorverarbeitung können für einen Entwickler eine komplexe Aufgabe sein. Die Data-API erleichtert das sehr. Sie bietet viele Features, unter anderem das Laden der Daten aus unterschiedlichen Quellen (wie zum Beispiel Text- oder Binärdateien), das parallele Lesen aus mehreren Quellen, das Umwandeln und Verweben der Datensätze, ein Durchmischen, in Batches einteilen und das Prefetchen von Daten.
2. Durch das Aufteilen eines großen Datensatzes auf mehrere Dateien können Sie die Daten grober vormischen, bevor Sie sie dann mit einem Shuffling Buffer feiner durchmischen. Zudem können Sie so mit riesigen Datenmengen umgehen, die nicht in einen einzelnen Rechner passen. Es ist auch einfacher, Tausende kleiner Dateien zu verarbeiten als eine riesige Datei – die Daten lassen sich so beispielsweise in mehrere Untermengen aufteilen.

Und wenn die Daten über mehrere Dateien auf mehreren Servern verteilt sind, ist es zudem noch möglich, sie simultan herunterzuladen, was die verfügbare Bandbreite besser auslastet.

3. Sie können mit TensorBoard Profiling-Daten visualisieren. Wird die GPU nicht vollständig ausgelastet, wird vermutlich Ihre Eingabepipeline der Flaschenhals sein. Das können Sie beheben, indem Sie die Daten parallel in mehreren Threads lesen, dann vorverarbeiten und sicherstellen, dass ein paar Batches prefetcht werden. Wenn das nicht ausreicht, damit Ihre GPU während des Trainings 100% Last hat, achten Sie darauf, dass der Code zum Vorverarbeiten optimiert ist. Sie können auch versuchen, den Datensatz in mehreren TFRecord-Dateien zu sichern und eventuell einen Teil der Vorverarbeitung schon im Voraus zu erledigen, sodass es nicht während des Trainings geschehen muss (hier kann TF Transform helfen). Verwenden Sie nötigenfalls eine Maschine mit mehr CPU und RAM und stellen Sie sicher, dass die GPU-Bandbreite ausreicht.
4. Eine TFRecord-Datei besteht aus einer Folge beliebiger Binärdatensätze: Sie können wirklich jegliche Binärdaten in jedem Datensatz ablegen. Aber in der Praxis enthalten die meisten TFRecord-Dateien Sequenzen serialisierter Protocol Buffer. Damit können Sie die Vorteile dieser Protocol Buffer nutzen, wie zum Beispiel, dass sie sich leicht auf mehreren Plattformen und mit vielen Programmiersprachen lesen lassen, zudem können ihre Definitionen auch im Nachhinein noch abwärtskompatibel geändert werden.
5. Das Protobuf-Format Example bietet den Vorteil, dass TensorFlow Operationen zum Parsen anbietet (die `tf.io.parse*example()`-Funktionen), ohne dass Sie Ihr eigenes Format definieren müssen. Es ist für die meisten Datensatzinstanzen ausreichend flexibel. Aber wenn Ihr Anwendungsfall nicht abgedeckt ist, können Sie Ihren eigenen Protocol Buffer definieren, ihn mit protoc kompilieren (die Argumente `--descriptor_set_out` und `--include_imports` sorgen für einen Export des Protobuf-Deskriptors) und mit der Funktion `tf.io.decode_proto()` die serialisierten Protobufs parsen (im Abschnitt »Custom protobuf« des Notebooks finden Sie ein Beispiel). Das Ganze ist komplizierter und erfordert das Deployen des Deskriptors zusammen mit dem Modell, aber es ist machbar.
6. Beim Einsatz von TFRecords werden Sie normalerweise die Komprimierung aktivieren wollen, wenn die TFRecord-Dateien vom Trainingsskript heruntergeladen werden müssen, da die Kompression die Dateien verkleinert und damit die Zeit zum Herunterladen verringert. Befinden sich die Dateien aber auf der gleichen Maschine wie das Trainingsskript, ist es im Allgemeinen besser, die Kompression abzuschalten, um die CPU nicht unnötig zu belasten.
7. Schauen wir uns die Vor- und Nachteile jeder Vorverarbeitungsmöglichkeit an:
 - Verarbeiten Sie die Daten beim Erstellen der Datendateien vor, wird das Trainingsskript schneller laufen, da dann nicht beim Training vorverarbeitet werden muss. In manchen Fällen werden die vorverarbeiteten Daten auch viel kleiner als die ursprünglichen Daten sein, dann können Sie so Platz sparen und ein Herunterladen beschleunigen. Es kann auch hilfreich sein, die vorverarbeiteten Daten greifbar zu haben, um sie beispielsweise zu untersuchen oder zu archivieren. Aber es gibt auch Nachteile. Es ist zum Beispiel nicht so einfach, mit mehreren Vorverarbeitungslogiken zu experimentieren, wenn Sie

einen vorverarbeiteten Datensatz für jede Variante benötigen. Und wenn Sie eine Data Augmentation durchführen wollen, kann es sein, dass Sie viele Varianten Ihres Datensatzes vorhalten müssen, die viel Plattenplatz benötigen und viel Zeit zum Generieren brauchen. Schließlich wird das trainierte Modell auch vorverarbeitete Daten erwarten, und Sie werden Ihre Anwendung um Code zum Vorverarbeiten ergänzen müssen, bevor diese das Modell aufruft.

- Werden die Daten mit der tf.data-Pipeline vorverarbeitet, ist es viel einfacher, die Vorverarbeitungslogik anzupassen und Data Augmentation umzusetzen. Zudem macht es tf.data einfach, sehr effiziente Vorverarbeitungspipelines umzusetzen (zum Beispiel mit Multithreading und Prefetching). Aber das Vorverarbeiten der Daten wird das Training verlangsamen. Und jede Trainingsinstanz wird in jeder Epoche vorverarbeitet statt nur einmal beim Erstellen der Datendateien. Schließlich wird das trainierte Modell immer noch vorverarbeitete Daten erwarten.
- Ergänzen Sie Ihr Modell um Schichten zur Vorverarbeitung, müssen Sie den Code zum Vorverarbeiten für Training und Inferenz nur einmal schreiben. Muss Ihr Modell auf vielen verschiedenen Plattformen deployt werden, muss der Code zur Vorverarbeitung nicht mehrfach geschrieben werden. Zudem laufen Sie nicht Gefahr, die falsche Vorverarbeitungslogik für Ihr Modell zu nutzen, da sie Teil des Modells ist. Andererseits wird ein Vorverarbeiten der Daten das Training verlangsamen, und jede Trainingsinstanz wird einmal pro Epoche neu vorverarbeitet. Zudem laufen die Vorverarbeitungsoperationen standardmäßig auf der GPU für den aktuellen Batch (Sie profitieren dann nicht von einem parallelen Vorverarbeiten auf der CPU und Prefetching). Glücklicherweise sollten die kommenden Keras-Schichten zur Vorverarbeitung dazu in der Lage sein, die entsprechenden Operationen als Teil der tf.data-Pipeline laufen zu lassen, womit Sie dann von Multithreading auf der CPU und Prefetching profitieren.
- Der Einsatz von TF Transform zum Vorverarbeiten bietet Ihnen schließlich viele Vorteile der vorherigen Optionen: Die vorverarbeiteten Daten sind greifbar, jede Instanz wird nur einmal vorverarbeitet (was das Training beschleunigt), und die Vorverarbeitungsschichten werden automatisch erzeugt, sodass Sie den Code dazu nur einmal schreiben müssen. Der größte Nachteil ist, dass Sie den Umgang mit diesem Tool lernen müssen.

8. Schauen wir uns an, wie man kategorische Merkmale und Text codieren kann:

- Um ein kategorisches Merkmal mit einer natürlichen Reihenfolge zu codieren, wie zum Beispiel eine Filmbewertung (»schlecht«, »mittelmäßig«, »gut«), ist die einfachste Option eine Ordinalcodierung: Sortieren Sie die Kategorien nach ihrer natürlichen Reihenfolge und bilden Sie jede auf ihren Rang ab (»bad« wird zu 0, »mittelmäßig« zu 1 und »gut« zu 2). Aber die meisten kategorischen Merkmale haben keine solche natürliche Reihenfolge. Es gibt zum Beispiel keine für Berufe oder Länder. In diesem Fall können Sie eine One-Hot-Codierung oder bei vielen Kategorien Embeddings einsetzen.
- Bei Text gibt es die Möglichkeit der Bag-of-Words-Repräsentierung: Ein Satz

wird als Vektor dargestellt, der das Auftreten jedes möglichen Worts zählt. Da häufige Wörter meist nicht so wichtig sind, werden Sie TF-IDF einsetzen wollen, um deren Gewichte zu reduzieren. Statt Wörter zu zählen, ist es auch üblich, n -Gramme zu zählen, bei denen es sich um Folgen von n aufeinanderfolgenden Wörtern handelt – nett und einfach. Alternativ können Sie jedes Wort durch Word Embeddings codieren, die eventuell vorgenutzt sind. Statt Wörter können Sie auch jeden Buchstaben oder Subwort-Token codieren (zum Beispiel »smartest« in »smart« und »est« aufteilen). Diese zwei letzten Optionen werden in [Kapitel 16](#) behandelt.

Die Lösungen zu den Übungsaufgaben 9 und 10 finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 14: Deep Computer Vision mit 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 Bildern 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 2.800 Parameter. Der zweite Convolutional Layer enthält 3×3 Kernels, und die Eingabe sind die 100 Feature Maps aus der vorherigen 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 $900 \times 200 = 180.200$ Parameter. Der dritte und letzte Convolutional

Layer enthält ebenfalls 3×3 Kernels, und seine Eingabe sind die 200 Feature Maps aus den vorherigen Schichten. Also enthält jede Feature Map $3 \times 3 \times 200 = 1.800$ Gewichte sowie einen Bias-Term. Da es 400 Feature Maps gibt, enthält diese Schicht insgesamt $1.801 \times 400 = 720.400$ Parameter. Rechnet man alles zusammen, hat dieses CNN $2.800 + 180.200 + 720.400 = 903.400$ Parameter.

Rechnen 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, werden die horizontale und die vertikale Dimension 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 (6 MB). Die zweite Schicht benötigt $4 \times 50 \times 75 \times 200 = 3$ Millionen Bytes (3 MB). Schließlich benötigt die dritte Schicht $4 \times 25 \times 38 \times 400 = 1.520.000$ Bytes (1,5 MB). Sobald eine Schicht berechnet wurde, kann der von der vorherigen Schicht verwendete Speicher freigegeben werden, bei guter Optimierung sind also im RAM nur $6 + 3 = 9$ Millionen Bytes (9 MB) nötig (wenn die zweite Schicht soeben fertig berechnet ist, 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 903.400 Parameter ausgerechnet, von denen jeder 4 Bytes belegt. Dadurch kommen noch einmal 3.613.600 Bytes hinzu (etwa 3,6 MB). Das gesamte nötige RAM beträgt (mindestens) 12.613.600 Bytes (etwa 12,6 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 Vorwärtsdurchlauf 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 statt in Bytes weiter. Wir hatten oben berechnet, dass die drei Schichten jeweils 6, 3 und 1,5 MB pro Datenpunkt benötigen. Das sind insgesamt 10,5 MB pro Datenpunkt. Bei 50 Datenpunkten benötigen wir also 525 MB an RAM. Dazu kommt das für die Eingabebilder benötigte RAM, also $50 \times 4 \times 200 \times 300 \times 3 = 36$ Millionen Bytes (36 MB) sowie das für die Modellparameter benötigte RAM, die oben berechneten 3,6 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 $525 + 36 + 3,6 = 564,6$ MB. Und das ist wirklich nur eine optimistisch errechnete 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 vorherige Fragen).
 5. Ein Local Response Normalization Layer sorgt dafür, dass die am stärksten aktivierten Neuronen die Neuronen an der gleichen Position in benachbarten Feature Maps hemmen, 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 es wesentlich größer und tiefer ist und 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 seine Konsistenz können als innovativ gelten. Der wichtigste Fortschritt von SENet war die Idee, einen SE-Block (ein Dense-Netz aus zwei Schichten) nach jedem Inception-Modul in einem Inception-Netz oder jeder Residual Unit in einem ResNet zu verwenden, um die relative Wichtigkeit von Feature Maps neu zu kalibrieren. Die wichtigste Innovation von Xception war schließlich der Einsatz der Depthwise Separable Convolutional Layers, die sich räumliche Muster und Muster entlang der Tiefe getrennt anschauen.
 7. Fully Convolutional Networks sind neuronale Netze, die nur aus Convolutional und Pooling Layers bestehen. FCNs können Bilder beliebiger Breite und Höhe (zumindest oberhalb einer Minimalgröße) effizient verarbeiten. Sie sind für die Objekterkennung und semantische Segmentierung am nützlichsten, weil sie sich das Bild nur einmal anschauen müssen (statt ein CNN mehrfach über die verschiedenen Teile des Bilds laufen zu lassen). Haben Sie ein CNN mit Dense-Schichten obendrauf, können Sie diese Dense-Schichten in Convolutional Layers umwandeln, um ein FCN zu schaffen: Ersetzen Sie einfach die unterste Dense-Schicht durch einen Convolutional Layer mit einer Kernelgröße gleich der Eingabegröße der Schicht, mit einem Filter pro Neuron in der Dense-Schicht und dem Einsatz des "valid"-Padding. Die Schrittweite sollte im Allgemeinen 1 sein, aber Sie können sie auf einen höheren Wert setzen, wenn Sie möchten. Die Aktivierungsfunktion sollte die gleiche wie die der Dense-Schicht sein. Die anderen Dense-Schichten sollten genauso konvertiert werden, allerdings mit 1×1 -Filtern. Es ist sogar möglich, ein trainiertes CNN auf diese Art und Weise umzuwandeln, indem die Gewichtsmatrizen der Dense-Schichten passend umgeformt werden.
 8. Die größte technische Schwierigkeit einer semantischen Segmentierung liegt darin, dass in einem CNN viel räumliche Auflösung verloren geht, wenn das Signal die Schichten durchläuft, insbesondere in Pooling Layers und in Schichten mit einer Schrittweite größer 1.

Diese räumliche Information muss irgendwie wiederhergestellt werden, um die Kategorie jedes Pixels genau vorhersagen zu können.

Die Lösungen zu den Übungsaufgaben 9 bis 12 finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 15: Verarbeiten von Sequenzen mit RNNs und CNNs

1. Folgendes sind einige Anwendungsgebiete von RNNs:
 - Bei einem Sequence-to-Sequence-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 Sequence-to-Vector-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 von vielen möglichen Implementierungen von *kollaborativen Filtern* für ein Empfehlungssystem).
 - Bei einem Vector-to-Sequence-RNN: Bilder mit Untertiteln versehen, eine Playlist von Musikstücken aus einem Embedding des aktuellen Interpreten erstellen, eine Melodie anhand einer Parameterliste erzeugen, Fußgänger auf einem Bild erkennen (z.B. einer Momentaufnahme der Kamera eines selbstfahrenden Autos).
2. Eine RNN-Schicht muss dreidimensionale Eingaben haben: Die erste Dimension ist die Batchdimension (dessen Größe die Batchgröße ist), die zweite repräsentiert die Zeit (deren Größe die Anzahl an Zeitschritten ist), und die dritte enthält die Eingaben für jeden Zeitschritt (deren Größe die Anzahl an Eingabemerkmale pro Zeitschritt ist). Wollen Sie zum Beispiel einen Batch mit fünf Zeitserien mit jeweils zehn Zeitschritten und zwei Werten pro Zeitschritt verarbeiten (etwa die Temperatur und die Windgeschwindigkeit), ist die Form [5, 10, 2]. Die Ausgaben sind ebenfalls dreidimensional: Die ersten beiden Dimensionen sind gleich, die dritte entspricht der Zahl von Neuronen. Verarbeitet beispielsweise eine RNN-Schicht mit 32 Neuronen den eben besprochenen Batch, wird die Ausgabe die Form [5, 10, 32] haben.
3. Um ein tiefes Sequence-to-Sequence-RNN mit Keras zu bauen, müssen Sie für alle RNN-Schichten `return_sequences=True` setzen. Bei einem Sequence-to-Vector-RNN setzen Sie `return_sequences=True` für alle RNN-Schichten mit Ausnahme der obersten, bei der `return_sequences=False` stehen muss (oder Sie setzen gar nichts, da `False` der Standardwert ist).
4. Haben Sie eine tägliche univariate Zeitserie und wollen Sie die nächsten sieben Tage vorhersagen, ist die einfachste RNN-Architektur dafür ein Stapel RNN-Schichten (alle mit `return_sequences=True` außer für die oberste RNN-Schicht) und sieben Neuronen in der Ausgabe-RNN-Schicht. Dann können Sie dieses Modell mit zufälligen Fenstern aus der

Zeitserie trainieren (zum Beispiel Sequenzen mit 30 aufeinanderfolgenden Tagen als Eingabe und einem Vektor mit den Werten der nächsten 7 Tage als Ziel). Das ist ein Sequence-to-Vector-RNN. Alternativ könnten Sie `return_sequences=True` für alle RNN-Schichten setzen, um ein Sequence-to-Sequence-RNN zu erstellen. Dann trainieren Sie das Modell mit zufälligen Fenstern aus der Zeitserie mit Sequenzen, die so lang wie die Ziele sind. Jede Zielsequenz sollte sieben Werte pro Zeitschritt enthalten (zum Beispiel sollte das Ziel für Zeitschritt t ein Vektor mit den Werten an den Zeitschritten $t + 1$ bis $t + 7$ sein).

5. Die zwei größten Probleme beim Trainieren von RNNs sind instabile Gradienten (explodierend oder verschwindend) und ein sehr beschränktes Kurzzeitgedächtnis. Diese Probleme werden beim Umgang mit langen Sequenzen noch schlimmer. Um die instabilen Gradienten im Griff zu behalten, können Sie eine kleinere Lernrate nutzen, auf eine sättigende Aktivierungsfunktion wie `tanh` zurückgreifen (den Standard) und eventuell bei jedem Zeitschritt Gradient Clipping, Layer Normalization oder Drop-out einsetzen. Um das beschränkte Kurzzeitgedächtnis anzugehen, können Sie LSTM- oder GRU-Schichten nutzen (das hilft ebenfalls bei instabilen Gradienten).
6. Die Architektur einer LSTM-Zelle sieht kompliziert aus, aber eigentlich ist sie gar nicht so schwer zu verstehen, wenn Sie die zugrunde liegende Logik durchschaut haben. Die Zelle besitzt einen Kurzzeitstatusvektor und einen Langzeitstatusvektor. Bei jedem Zeitschritt werden die Eingaben und der vorherige Kurzzeitstatus an eine einfache RNN-Zelle und drei Gates übergeben: Das Forget Gate entscheidet, was aus dem Langzeitstatus zu entfernen ist, das Input Gate entscheidet, welche Teile der Ausgabe der einfachen RNN-Zelle zum Langzeitstatus hinzuzufügen sind, und das Output Gate entscheidet, welche Teile des Langzeitstatus bei diesem Zeitschritt ausgegeben werden sollten (nach dem Durchlauf durch die `tanh`-Aktivierungsfunktion). Der neue Kurzzeitstatus entspricht dann der Ausgabe der Zelle (siehe [Abbildung 15-9](#)).
7. Eine RNN-Schicht ist erst einmal sequenziell: Um die Ausgaben bei Zeitschritt t zu berechnen, muss sie erst die Ausgaben aller vorherigen Zeitschritte berechnen. Das macht ein Parallelisieren unmöglich. Ein 1-D-Convolutional-Layer eignet sich hingegen sehr gut zur Parallelisierung, da er zwischen den Zeitschritten keinen Status aufbewahren muss. Mit anderen Worten – er hat kein Gedächtnis. Die Ausgabe kann bei jedem Zeitschritt allein anhand eines kleinen Fensters mit Werten aus den Eingaben berechnet werden, ohne alle vergangenen Werte kennen zu müssen. Und da ein 1-D-Convolutional-Layer nicht rekurrent ist, leidet er weniger unter instabilen Gradienten. Einer oder mehrere 1-D-Convolutional-Layers können in einem RNN nützlich sein, um die Eingaben effizient vorzuverarbeiten – zum Beispiel zum Verringern ihrer zeitlichen Auflösung (Downsampling), womit die RNN-Schichten langfristige Muster besser erkennen können. Tatsächlich ist es möglich, nur mit Convolutional Layers zu arbeiten, was zum Beispiel bei der WaveNet-Architektur geschieht.
8. Um Videos anhand ihres visuellen Inhalts zu klassifizieren, könnten Sie beispielsweise einen Frame pro Sekunde nehmen, dann jeden Frame durch das gleiche Convolutional Neural Network laufen lassen (zum Beispiel ein vortrainiertes Xception-Modell, eventuell eingefroren, wenn Ihr Datensatz nicht zu groß ist), die Ausgabesequenz aus dem CNN an

ein Sequence-to-Vector-RNN übergeben und schließlich dessen Ausgabe durch eine Softmax-Schicht schicken, wodurch Sie die Wahrscheinlichkeiten für alle Kategorien erhalten. Zum Trainieren würden Sie als Kostenfunktion die Kreuzentropie verwenden. Wollen Sie auch die Audiodaten zur Klassifikation einsetzen, könnten Sie einen Stack mit Strided-1-D-Convolutional-Layers nutzen, um die zeitliche Auflösung von Tausenden Audioframes pro Sekunde auf nur einen pro Sekunde zu verringern (um zur Anzahl an Bildern pro Sekunde zu passen), und die Ausgabesequenz mit den Eingaben des Sequence-to-Vector-RNN (entlang der letzten Dimension) verbinden.

Die Lösungen zu den Übungsaufgaben 9 und 10 finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 16: Natürliche Sprachverarbeitung mit RNNs und Attention

1. Zustandslose RNNs können nur Muster erfassen, deren Länge kleiner oder gleich der Größe des Fensters ist, mit dem das RNN trainiert wird. Umgekehrt können zustandsbehaftete RNNs langfristigere Muster erfassen. Aber das Implementieren eines zustandsbehafteten RNN ist viel schwerer – insbesondere das korrekte Vorbereiten des Datensatzes. Zudem funktionieren zustandsbehaftete RNNs nicht immer besser – zum Teil, da aufeinanderfolgende Batches nicht unabhängig und gleichverteilt (Independent and Identically Distributed, IID) sind. Die Gradientenmethode ist für Nicht-IID-Datensätze nicht so gut geeignet.
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 beim nächsten Wort nicht sicher ist (wie es menschliche Dolmetscher bei einer Simultanübersetzung tun).
3. Eingabesequenzen variabler Länge können durch ein Auffüllen der kürzeren Sequenzen verarbeitet werden, sodass alle Sequenzen in einem Batch die gleiche Länge haben, und indem Maskierung verwendet wird, um sicherzustellen, dass das RNN das Padding-Token ignoriert. Für eine bessere Performance könnten Sie auch Batches erstellen, die Sequenzen ähnlicher Länge enthalten. Ragged-Tensoren können Sequenzen variabler Länge aufnehmen, und tf.keras wird sie vermutlich auch irgendwann unterstützen, was die Arbeit mit Eingabesequenzen variabler Länge deutlich vereinfachen wird (aktuell ist das noch nicht der Fall). Wenn es um Ausgabesequenzen variabler Länge geht, müssen Sie bei Kenntnis der Länge im Voraus (zum Beispiel weil Sie wissen, dass sie genauso lang wie die Eingabesequenz ist) nur die Verlustfunktion so konfigurieren, dass sie Token nach dem Ende der Sequenz ignoriert. Ebenso sollte der Code, den das Modell verwendet, Token nach dem Ende der Sequenz ignorieren. Aber im Allgemeinen ist die Länge der Ausgabesequenz

nicht im Voraus bekannt, daher liegt die Lösung darin, das Modell so zu trainieren, dass es am Ende jeder Sequenz ein End-of-Sequence-Token ausgibt.

4. Beam Search ist eine Technik, die zum Verbessern der Leistung eines trainierten Encoder-Decoder-Modells dient, zum Beispiel in einem Übersetzungssystem. Der Algorithmus merkt sich eine kurze Liste der k vielversprechendsten Ausgabesätze (zum Beispiel die Top 3), und bei jedem Decoder-Schritt versucht er, sie um ein Wort zu erweitern. Dann merkt er sich wieder nur die k wahrscheinlichsten Sätze. Der Parameter k wird als *Beam Width* bezeichnet – je größer er ist, desto mehr CPU und RAM wird gebraucht, aber desto genauer wird auch das System sein. Statt bei jedem Schritt gierig das wahrscheinlichste nächste Wort zu wählen, um einen einzelnen Satz zu erweitern, erlaubt diese Technik dem System, mehrere vielversprechende Sätze simultan zu untersuchen. Zudem kann sie gut parallelisiert werden. Sie können die Beam Search recht einfach über TensorFlow Addons implementieren.
5. Ein Attention-Mechanismus ist eine Technik, die ursprünglich in Encoder-Decoder-Modellen genutzt wurde, um dem Decoder einen direkteren Zugriff auf die Eingabesequenzen zu erlauben und es ihm damit zu ermöglichen, mit längeren Sequenzen umzugehen. Bei jedem Decoder-Zeitschritt werden der aktuelle Status des Decoders und die vollständige Ausgabe des Encoders von einem Alignment-Modell verarbeitet, das einen Alignment Score für jeden Eingabezeitschritt ausgibt. Dieser Score gibt an, welcher Teil der Eingabe für den aktuellen Decoder-Zeitschritt am relevantesten ist. Die gewichtete Summe der Encoder-Ausgabe (gewichtet nach dem Alignment Score) wird dann an den Decoder übergeben, der den nächsten Decoder-Status und die Ausgabe für diesen Zeitschritt erzeugt. Der Hauptvorteil des Attention-Mechanismus ist, dass das Encoder-Decoder-Modell erfolgreich längere Eingabesequenzen verarbeiten kann. Ein weiterer Vorteil ist, dass das Modell durch die Alignment Scores leichter zu debuggen und zu interpretieren sein wird: Macht das Modell beispielsweise einen Fehler, können Sie sich anschauen, auf welchen Teil der Eingabe es seine Aufmerksamkeit gerichtet hat, was bei der Diagnose des Problems helfen kann. Ein Attention-Mechanismus ist zudem in den Multi-Head-Attention-Schichten das Herz der Transformer-Architektur (siehe nächste Antwort).
6. Die wichtigste Schicht in der Transformer-Architektur ist die Multi-Head-Attention-Schicht (die ursprüngliche Transformer-Architektur enthielt 18 davon, unter anderem 6 Masked Multi-Head-Attention-Schichten). Sie bildet auch den Kern von Sprachmodellen wie BERT oder GPT-2. Mit dieser Schicht kann das Modell herausfinden, welche Wörter miteinander am meisten verbunden sind, und dann die Repräsentation jedes Worts mit diesen kontextuellen Hinweisen verbessern.
7. Sampled Softmax wird beim Trainieren eines Klassifikationsmodells verwendet, wenn es viele Kategorien gibt (zum Beispiel Tausende). Er berechnet eine Näherung des Kreuzentropieverlusts, die auf dem vom Modell für die korrekte Kategorie vorhergesagten Logit und den vorhergesagten Logits für ein Sample falscher Wörter basiert. Das beschleunigt das Training im Vergleich zum Berechnen des Softmax über alle Logits und dem darauffolgenden Schätzen des Kreuzentropieverlusts deutlich. Nach dem Training kann das Modell normal mit der regulären Softmax-Funktion verwendet werden, um alle Kategorienwahrscheinlichkeiten basierend auf allen Logits zu berechnen.

Die Lösungen zu den Übungsaufgaben 8 bis 11 finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 17: Representation Learning und Generative Learning mit Autoencodern und GANs

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 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 einfrieren.
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. könnte der erste Datenpunkt als 0,001 codiert sein, der zweite als 0,002, der dritte als 0,003 und so weiter). So 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 Autoencoder 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 Autoencoder, 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 Vortrainieren einsetzen möchten, sollten Sie auch die Leistung des Klassifikators auswerten.
4. Ein unturvollständiger 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 unturvollständigen 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. 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.
7. Ein Generative Adversarial Network ist eine neuronale Netzarchitektur aus zwei Teilen – dem Generator und dem Diskriminator, die gegensätzliche Ziele verfolgen. Ziel des Generators ist, Instanzen zu erzeugen, die denen im Trainingsdatensatz ähneln, um den Diskriminator zu täuschen. Der Diskriminator muss die echten Instanzen von den generierten Instanzen unterscheiden. Bei jeder Trainingsiteration wird der Diskriminator wie ein normaler Binärklassifikator trainiert, dann wird der Generator darauf trainiert, den Fehler des Diskriminators zu maximieren. GANs werden für fortgeschrittene Bildverarbeitungsaufgaben genutzt, wie zum Beispiel zum Verbessern der Auflösung, zum Einfärben, zur Bildbearbeitung (Objekte durch realistische Hintergründe ersetzen), zum Umwandeln einer einfachen Skizze in ein fotorealistisches Bild oder zum Vorhersagen der nächsten Frames in einem Video. Außerdem kommen sie beim Augmentieren eines Datensatzes zum Einsatz (um andere Modelle zu trainieren), beim Erzeugen anderer Arten von Daten (zum Beispiel Text, Audio oder Zeitserien) und beim Auffinden und Beheben von Schwächen in anderen Modellen.
8. Das Trainieren eines GAN ist aufgrund der komplexen Dynamiken zwischen dem Generator und dem Diskriminator immer schwierig. Das größte Problem ist der Mode Collapse, bei dem der Generator Ausgaben mit sehr wenig Diversität erzeugt. Zudem kann das Training furchtbar instabil sein: Es kann gut losgehen und dann plötzlich ohne offensichtlichen Grund oszillieren oder divergieren. GANs reagieren auch sehr empfindlich auf die Wahl der Hyperparameter.

Die Lösungen zu den Übungen 9, 10 und 11 finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

Kapitel 18: 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 überwachtem und unüberwachtem Lernen. Hier sind einige davon:

- Beim überwachten und unüberwachten Lernen ist das Ziel, Muster in den Daten zu entdecken und sie zur Vorhersage zu nutzen. 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 (Buffer) verwendet, um sicherzustellen, dass der Trainingsalgorithmus halbwegs unabhängige Beobachtungen erhält.

2. Hier sind, zusätzlich zu den in [Kapitel 18](#) erwähnten, einige mögliche Anwendungen für Reinforcement Learning:

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 abzuspielen (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 eine 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 normalerweise bei jedem Schritt einen Discountfaktor mit eingerechnet. Beispielsweise würde bei einem Discountfaktor 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 den Discountfaktor als ein Maß dafür vorstellen, wie stark die Zukunft im Vergleich zur Gegenwart gewertet wird: Liegt er sehr nah an 1, ist die Zukunft beinahe genauso viel wert wie die Gegenwart. Liegt er nahe bei 0, zählen nur unmittelbare Belohnungen. Natürlich hat dies einen immensen Einfluss auf die optimale Policy: Wenn Sie der Zukunft einen Wert beimesse, sind Sie vielleicht bereit, eine Menge unmittelbarer Schmerzen für die Aussicht auf eventuelle Gewinne auf sich nehmen. Messen Sie der Zukunft dagegen keinen Wert bei, 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 (ebenso Minimum, Maximum, Standardabweichung und so weiter).
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. können beim Atari-Spiel *Pong* 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 von Erfahrungen aus seiner jüngeren und früheren Vergangenheit. (Vielleicht träumen wir deshalb: 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 Belohnungen unter

Berücksichtigung der Discount-Rate), während der Agent eine andere Policy verfolgt. Q-Learning ist ein gutes Beispiel für solch einen Algorithmus. Im Gegensatz dazu lernt 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-ml2>.

Kapitel 19: TensorFlow-Modelle skalierbar trainieren und deployen

1. Ein SavedModel enthält ein TensorFlow-Modell mit seiner Architektur (einem Rechengraphen) und den Gewichten. Es wird als Verzeichnis mit einer Datei `saved_model.pb` abgelegt, in der der Rechengraph definiert ist (als serialisierter Protocol Buffer), und einem Unterverzeichnis `variables` mit den Variablenwerten. Bei Modellen mit einer großen Zahl von Gewichten können diese Werte auf mehrere Dateien verteilt sein. Ein SavedModel enthält zudem ein Unterverzeichnis `assets`, in dem sich zusätzliche Daten befinden können, wie zum Beispiel Vokabulardateien, Klassennamen oder Beispieldaten für dieses Modell. Wenn man genau sein will, kann ein SavedModel auch noch einen oder mehrere *Metagraphen* besitzen. Dabei handelt es sich um einen Rechengraphen zusammen mit Definitionen von Funktionssignaturen (einschließlich der Eingabe- und Ausgabennamen, -typen und -formen). Jeder Metagraph wird durch einen Satz Tags identifiziert. Um ein SavedModel zu untersuchen, können Sie das Befehlszeilentool `saved_model_cli` nutzen oder es einfach mit `tf.saved_model.load()` laden und in Python unter die Lupe nehmen.
2. TF Serving ermöglicht es Ihnen, mehrere TensorFlow-Modelle (oder mehrere Versionen des gleichen Modells) zu deployen und über eine REST-API oder gRPC-API ganz einfach all Ihren Anwendungen bereitzustellen. Wenn Sie Ihre Modelle in Ihren Anwendungen direkt verwenden, wird es schwerer, eine neue Version eines Modells in allen Anwendungen auszurollen. Es würde mehr Aufwand bedeuten, einen eigenen Microservice zu implementieren, der ein TF-Modell verpackt, und es wäre schwer, mit den Features von TF Serving mitzuhalten: Es kann ein Verzeichnis überwachen und automatisch die Modelle deployen, die dort abgelegt werden, und Sie müssen keine Ihrer Anwendungen ändern oder auch nur neu starten, um von den neuen Modellversionen zu profitieren. Es ist schnell, umfassend getestet und skaliert sehr gut. Es unterstützt A/B-Testing für experimentelle Modelle und das Deployen einer neuen Modellversion an nur eine Untergruppe der Anwender (das nennt sich dann ein *Canary*). TF Serving kann zudem einzelne Requests in Batches zusammenfassen, um sie gemeinsam auf der GPU laufen zu lassen. Um TF Serving zu deployen, können Sie es aus seinem Quellcode installieren, aber es ist viel einfacher, es über ein Docker-Image zu installieren. Um ein Cluster mit TF-Serving-Docker-Images zu deployen, können Sie ein Orchestrierungstool wie Kubernetes verwenden oder auf eine vollständig gehostete Lösung wie Google Cloud AI Platform zurückgreifen.
3. Um ein Modell auf mehreren Instanzen von TF Serving zu deployen, müssen Sie diese Instanzen nur so konfigurieren, dass sie alle das gleiche `models`-Verzeichnis überwachen,

und dann Ihr neues Modell als SavedModel in einem Unterverzeichnis ablegen.

4. Die gRPC-API ist effizienter als die REST-API. Aber deren Clientbibliotheken stehen nicht so umfassend zur Verfügung, und wenn Sie beim Einsatz der REST-API die Komprimierung einschalten, können Sie fast die gleiche Performance erhalten. Daher ist die gRPC-API dann sinnvoll, wenn Sie die höchstmögliche Leistung brauchen und die Clients nicht auf die REST-API beschränkt sind.
5. Um die Größe eines Modells zu reduzieren, damit es auf einem mobilen oder Embedded Device laufen kann, nutzt TFLite eine Reihe von Techniken:
 - Es stellt einen Konverter bereit, der ein SavedModel optimieren kann: Er dampft das Modell ein und verringert seine Latenz. Dazu schneidet er alle Operationen weg, die zum Treffen von Vorhersagen nicht notwendig sind (wie zum Beispiel Trainingsoperationen), und optimiert und verschmilzt wann immer möglich Operationen.
 - Der Konverter kann zudem eine Post-Training-Quantisierung durchführen: Diese Technik verringert die Modellgröße drastisch, sodass sich das Modell dann schneller herunterladen und abspeichern lässt.
 - Es sichert das optimierte Modell im FlatBuffer-Format, das ohne Parsen direkt ins RAM geladen werden kann. Damit verringern sich Ladenzeiten und Speicherbedarf.
6. Quantisierungsbewusstes Training fügt beim Training künstliche Quantisierungsoperationen hinzu. Damit kann das Modell lernen, das Quantisierungsrauschen zu ignorieren – die finalen Gewichte werden dann robuster darauf reagieren.
7. Für parallelisierte Modelle zerschneiden Sie Ihr Modell in mehrere Teile und führen diese parallel auf mehreren Devices aus, womit Sie das Modell beim Training oder bei der Inferenz hoffentlich beschleunigen. Bei parallelisierten Daten erstellen Sie mehrere gleiche Repliken Ihres Modells und deployen sie auf mehrere Devices. Während des Trainings erhält jede Replik bei jeder Iteration einen anderen Datenbatch, und sie berechnet die Gradienten des Verlusts bezüglich der Modellparameter. Bei synchron parallelisierten Daten werden dann die Gradienten aller Repliken zusammengeführt, und der Optimierer führt einen Gradientenschritt durch. Die Parameter können zentral (zum Beispiel auf Parameterservern) oder verteilt über alle Repliken vorliegen und mithilfe von AllReduce synchron gehalten werden. Bei asynchron parallelisierten Daten liegen die Parameter zentral vor, und die Repliken laufen unabhängig voneinander. Jede Replik aktualisiert die zentralen Parameter direkt am Ende jeder Trainingsiteration, ohne auf die anderen Repliken warten zu müssen. Um das Training zu beschleunigen, sind parallelisierte Daten im Allgemeinen besser als parallelisierte Modelle. Das liegt vor allem daran, dass zwischen den Devices weniger Kommunikation erforderlich ist. Zudem lässt sie sich viel leichter implementieren und funktioniert für jedes Modell gleich, während Sie bei parallelisierten Modellen das Modell analysieren müssen, um herauszufinden, wie Sie es am besten unterteilen können.
8. Beim Trainieren eines Modells auf mehreren Servern können Sie die folgenden Verteilstrategien anwenden:
 - Die `MultiWorkerMirroredStrategy` setzt auf gespiegelte parallelisierte Daten. Das Modell wird auf alle verfügbaren Server und Devices deployt, jede

Replik erhält bei jeder Trainingsiteration einen anderen Batch mit Daten und berechnet ihre eigenen Gradienten. Dann wird mit einer verteilten AllReduce-Implementierung (standardmäßig NCCL) der Mittelwert der Gradienten bestimmt und auf alle Repliken verteilt, wonach dort überall der gleiche Gradientenschritt ausgeführt wird. Diese Strategie lässt sich am einfachsten einsetzen, da alle Server und Devices genau gleich behandelt werden und sie ziemlich gut funktioniert. Im Allgemeinen sollten Sie diese Strategie verfolgen. Ihre größte Einschränkung ist, dass das Modell auf jeder Replik ins RAM passen muss.

- Die `ParameterServerStrategy` nutzt asynchron parallelisierte Daten. Das Modell wird auf alle Devices auf allen Workern deployt, und die Parameter werden per Sharding über alle Parameterserver verteilt. Jeder Worker besitzt seine eigene Trainingsschleife und läuft asynchron zu den anderen Workern. Bei jeder Trainingsiteration erhält jeder Worker seinen eigenen Datenbatch und holt sich vom Parameterserver die neueste Version der Modellparameter. Dann berechnet er die Gradienten des Verlusts bezüglich dieser Parameter und schickt sie an die Parameterserver. Schließlich führen die Parameterserver mit diesen Gradienten einen Gradientenschritt durch. Diese Strategie ist meist langsamer als die vorherige Strategie, und sie lässt sich auch nur aufwendiger deployen, da dafür Parameterserver verwaltet werden müssen. Aber sie ist nützlich, um riesige Modelle zu trainieren, die nicht in das RAM der GPU passen.

Die Lösungen zu den Übungsaufgaben 9, 10 und 11 finden Sie in den Jupyter-Notebooks unter <https://github.com/ageron/handson-ml2>.

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. Wie sehen die bisherigen Lösungen oder Übergangslösungen aus (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 auf Erfahrungen zurückgreifen 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 Informationen gelöscht oder geschützt werden (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 (nicht 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)
 - Nutzen 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 »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 nützlichen Informationen 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 Merkmalen: 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, diese Schritte so weit wie möglich zu automatisieren.
1. Trainieren Sie schnell viele unterschiedliche Modelle (z.B. lineare Modelle, Naive Bayes, SVMs, 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 vorherigen Schritte ein- oder zweimal.
 7. Erstellen Sie eine Auswahl der drei bis fünf vielversprechendsten Modelle und bevorzugen Sie dabei Modelle, die unterschiedliche Arten von Fehlern 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. (Sollten Sie z.B. 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. Dauert das Training sehr lang, erwägen Sie ein bayessches Optimierungsverfahren (z.B. mit Gauß-Prozess-Priors, wie von Jasper Snoek et al. beschrieben (<https://homl.info/134>)).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.
- 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 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 Onlinelernsystemen 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. Angenommen, 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, 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.

$$\begin{cases} \frac{\partial}{\partial x} g(x, y, \alpha) = 2x - 3\alpha \\ \frac{\partial}{\partial y} g(x, y, \alpha) = 2 - 2\alpha \\ \frac{\partial}{\partial \alpha} g(x, y, \alpha) = -3x - 2y - 1 \end{cases}$$

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 einer 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

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2} \mathbf{w}^\top \mathbf{w} - \sum_{i=1}^m \alpha^{(i)} (t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) - 1)$$

mit $\alpha^{(i)} \geq 0$ für $i = 1, 2, \dots, m$

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:
- $t^{(i)}(\hat{\mathbf{w}}^\top \mathbf{x}^{(i)} + \hat{b}) \geq 1$ für $i = 1, 2, \dots, m$.
- 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: $t^{(i)}(\hat{\mathbf{w}}^\top \mathbf{x}^{(i)} + \hat{b}) = 1$. 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 das 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

$$\mathcal{L}(\hat{\mathbf{w}}, \hat{b}, \alpha) = \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)\top} \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

mit $\alpha^{(i)} \geq 0$ für $i = 1, 2, \dots, m$

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 Supportvektor $t^{(i)}(\mathbf{w}^\top \cdot \mathbf{x}^{(i)} + b) = 1$ gilt. Wenn also der k te Datenpunkt ein Stützvektor ist (also $\alpha_k > 0$), können Sie ihn verwenden, um $\hat{b} = t^{(k)} - \hat{\mathbf{w}}^\top \mathbf{x}^{(k)}$ 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

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m [t^{(i)} - \hat{\mathbf{w}}^\top \mathbf{x}^{(i)}]$$

ANHANG D

Autodiff

Dieser Anhang erklärt, wie das Autodiff-Feature in TensorFlow funktioniert und was es im Vergleich zu seinen Alternativen leistet.

Nehmen wir an, Sie definierten 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 Finite-Differenzen-Methode, Autodiff im Forward-Modus und Autodiff im Reverse-Modus. In TensorFlow ist die letztere Möglichkeit implementiert, aber zu ihrem Verständnis ist es sinnvoll, sich zunächst die anderen anzuschauen. Betrachten wir also jede dieser Möglichkeiten etwas genauer, beginnend mit der Differenzierung von Hand.

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 passende Gleichung zu ermitteln. 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 $f(x,y)$

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial(x^2y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y \frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy \\ \frac{\partial f}{\partial y} &= \frac{\partial(x^2y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1\end{aligned}$$

Bei komplexeren Funktionen wird dieser Ansatz sehr mühselig, und das Risiko, dabei Fehler zu

machen, erhöht sich. Glücklicherweise gibt es andere Optionen. Schauen wir zuerst die Finite-Differenzen-Methode an.

Finite-Differenzen-Methode

Sie erinnern sich, dass die Ableitung $h'(x_0)$ einer Funktion $h(x)$ am Punkt x_0 die Steigung der Funktion an diesem Punkt ist. Genauer gesagt, ist die Ableitung definiert als der Grenzwert der Steigung einer gerade Linie, die durch diesen Punkt x_0 und einen anderen Punkt x der Funktion geht, wenn sich x immer mehr x_0 nähert (siehe [Formel D-2](#)).

Formel D-2: Definition der Ableitung einer Funktion $h(x,y)$ am Punkt x_0

$$\begin{aligned} h'(x_0) &= \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0} \\ &= \lim_{\varepsilon \rightarrow 0} \frac{h(x_0 + \varepsilon) - h(x_0)}{\varepsilon} \end{aligned}$$

Wollen wir also die partielle Ableitung von $f(x, y)$ in Bezug auf x bei $x = 3$ und $y = 4$ berechnen, könnten wir $f(3 + \varepsilon, 4) - f(3, 4)$ berechnen und das Ergebnis durch ε dividieren, wobei wir einen sehr kleinen Wert für ε wählen. Diese Art numerischer Näherung der Ableitung nennt man eine Finite-Differenzen-Näherung, und diese spezifische Gleichung ist Newtons Differenzenquotient. Genau das macht der folgende Code:

```
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 für kompliziertere Funktionen wird es noch schlechter). Die korrekten Ergebnisse sind 24 bzw. 10, stattdessen erhalten wir aber:

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

Beachten Sie, dass wir zum Berechnen beider partieller Ableitungen $f()$ mindestens drei Mal aufrufen müssen (im vorherigen Code haben wir es vier Mal aufgerufen, aber das könnte optimiert werden). Hätten wir 1.000 Parameter, müssten wir $f()$ mindestens 1.001 Mal aufrufen. Arbeiten Sie mit großen neuronalen Netzen, ist die Finite-Differenzen-Methode viel zu ineffizient.

Dennoch lässt sich diese Methode so einfach implementieren, dass man mit ihr sehr gut prüfen kann, ob die anderen Methoden korrekt implementiert sind. Stimmt sie beispielsweise mit Ihrer per Hand abgeleiteten Funktion nicht überein, dürfte vermutlich Ihre Funktion den Fehler enthalten.

Bisher haben wir zwei Wege kennengelernt, Gradienten zu berechnen: mit der Ableitung per Hand und mit der Finite-Differenzen-Methode. Leider sind beide beim Trainieren eines großen neuronalen Netzes zu unpraktisch. Wenden wir uns daher Autodiff zu und beginnen wir mit dem Forward-Modus.

Autodiff im Forward-Modus

Abbildung D-1 demonstriert die Funktionsweise des Autodiff im Forward-Modus anhand einer noch einfacheren Funktion, $g(x,y) = 5 + xy$. Der Graph der Funktionsgleichung wird auf der linken Seite gezeigt. Nach dem Autodiff im Forward-Modus erhalten wir den Graphen auf der rechten Seite, der die partielle Ableitung $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1) = y$ darstellt (in ähnlicher Weise könnten wir auch die partielle Ableitung nach y ermitteln).

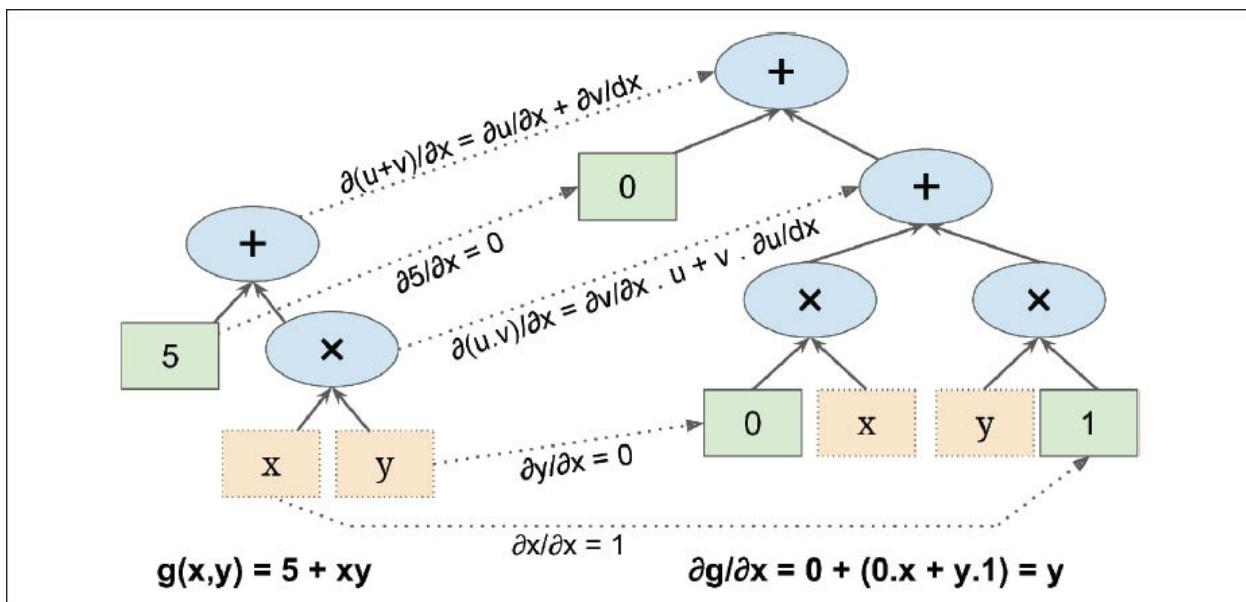


Abbildung D-1: Autodiff im Forward-Modus

Der Algorithmus durchläuft den Rechengraphen von den Eingaben zu den Ausgaben (daher der Name »Forward-Modus«). Er 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 $\frac{\partial(u \times v)}{\partial x} = \frac{\partial v}{\partial x} \times u + v \times \frac{\partial u}{\partial x}$ 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 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 das Autodiff im Forward-Modus einen riesigen Graphen erzeugen, der sehr schwer zu vereinfachen ist und zu einer suboptimalen Leistung führt.

Beachten Sie, dass wir mit einem Rechengraphen begonnen haben und Autodiff im Forward-Modus einen anderen Rechengraphen erzeugt hat. Dies wird als *symbolisches Differenzieren* bezeichnet und hat zwei nette Features: Ist der Rechengraph der Ableitung einmal erstellt, können wir ihn beliebig oft einsetzen, um die Ableitungen der gegebenen Funktion für alle Werte von x und y zu berechnen. Und wir können Autodiff im Forward-Modus erneut auf den erzeugten Rechengraphen anwenden, um zweite Ableitungen zu erzeugen, falls wir sie einmal benötigen (also Ableitungen von Ableitungen). Wir könnten sogar noch weiter gehende Ableitungen berechnen.

Aber es ist auch möglich, Autodiff im Forward-Modus laufen zu lassen, ohne einen Graphen zu erstellen (also numerisch statt symbolisch), indem wir direkt die Zwischenergebnisse berechnen. Ein möglicher Weg ist der Einsatz von *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

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$

$$(a + b\epsilon) \cdot (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

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 die partielle Ableitung von $f(x,y)$ nach x am Punkt $x = 3$ und $y = 4$ mithilfe dualer Zahlen berechnet werden kann. 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.

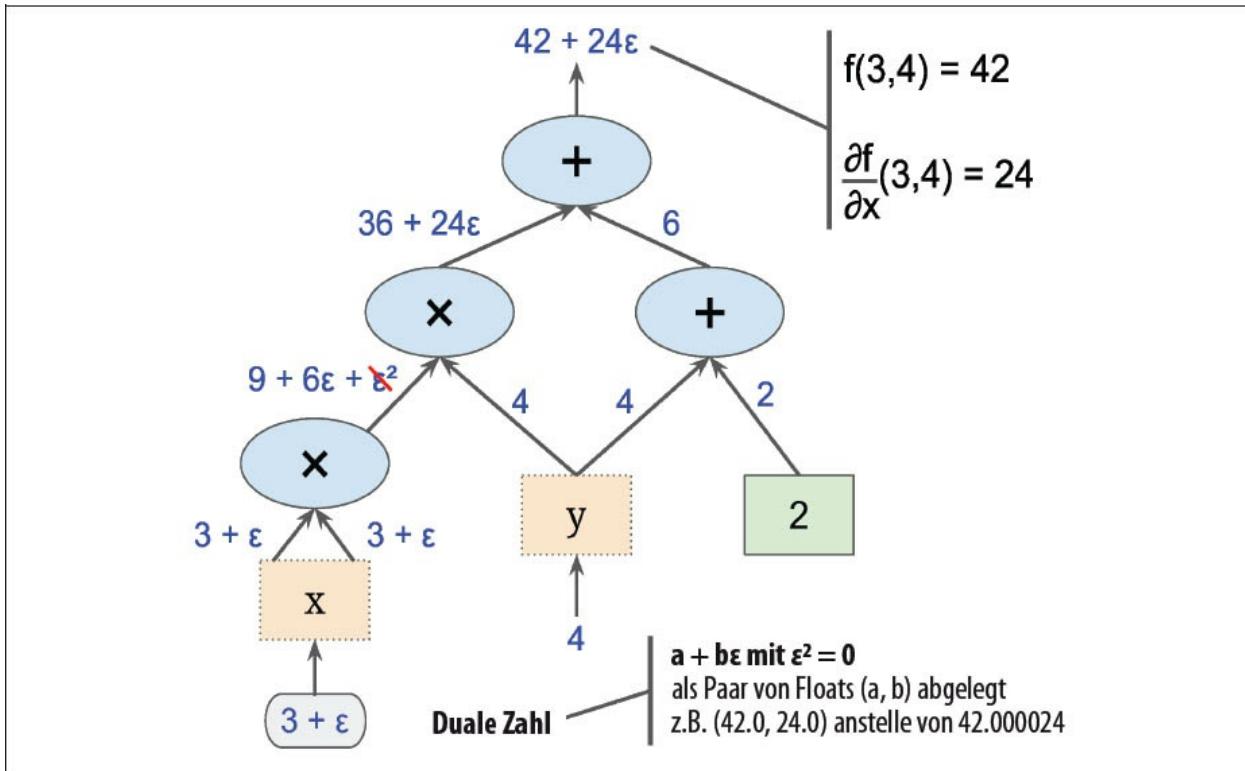


Abbildung D-2: Autodiff im Forward-Modus mit dualen Zahlen

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 Finite-Differenzen-Methode, hat aber denselben Schönheitsfehler (zumindest bei vielen Eingaben und wenigen Ausgaben, was bei neuronalen Netzen der Fall ist): Wenn es 1.000 Parameter gäbe, müssten wir den Graphen 1.000

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. Der Name »Reverse-Modus« stammt von diesem zweiten Durchgang durch den Graphen, bei dem die Gradienten in die entgegengesetzte Richtung fließen. [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 $n_7: f(3,4) = n_7 = 42$.

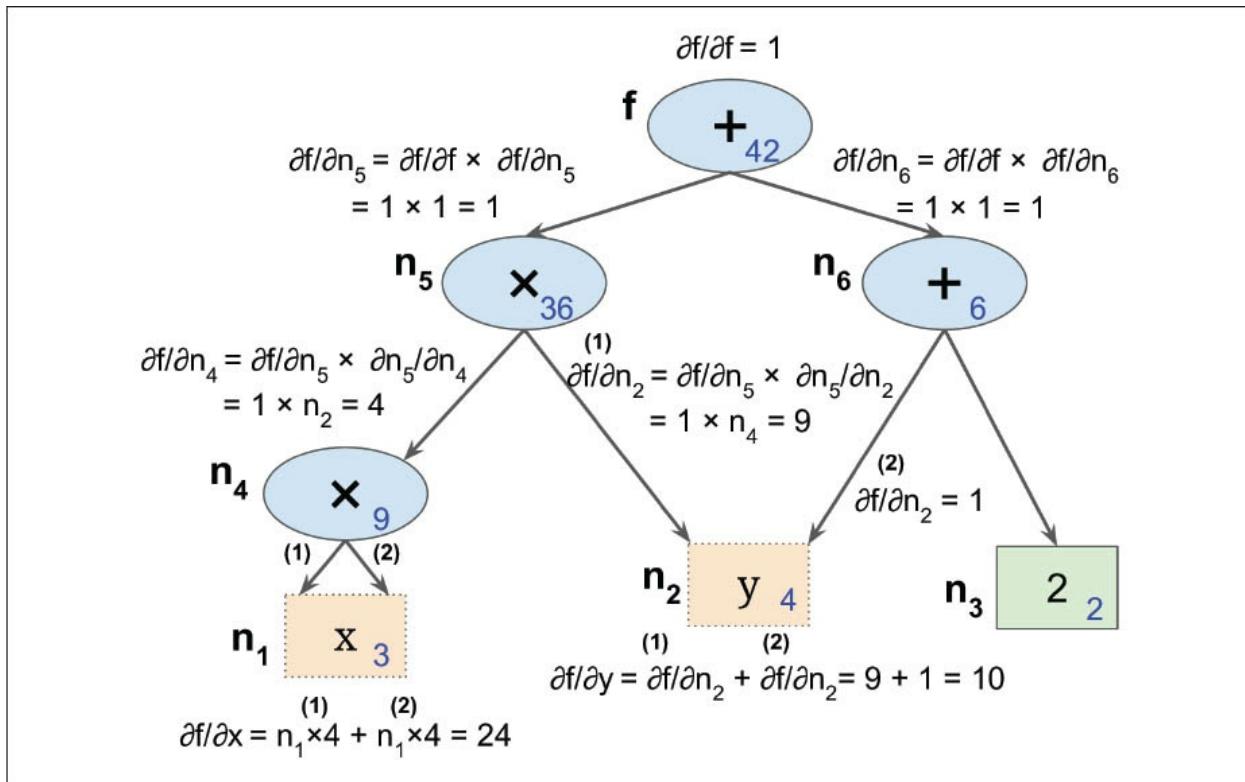


Abbildung D-3: Autodiff im Reverse-Modus

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 $\frac{\partial f}{\partial x} = 10$ und $\frac{\partial f}{\partial y} = 24$ 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. Beim Trainieren neuronaler Netze wollen wir im Allgemeinen den Verlust minimieren, daher gibt es eine einzelne Ausgabe (den Verlust) und damit nur zwei Durchgänge durch den Graphen, um die Gradienten zu berechnen. Autodiff im Reverse-Modus kann auch mit Funktionen umgehen, die nicht vollständig differenzierbar sind, solange Sie es nur um die partiellen Ableitungen an differenzierbaren Punkten bitten.

In [Abbildung D-3](#) werden die numerischen Ergebnisse direkt an jedem Knoten berechnet. TensorFlow geht allerdings ein wenig anders vor: Es erstellt stattdessen einen neuen Rechengraphen. Mit anderen Worten – es implementiert ein *symbolisches* Autodiff im Reverse-Modus. So muss der Rechengraph zum Ermitteln der Gradienten des Verlusts bezüglich aller Parameter im neuronalen Netz nur einmal erzeugt werden und kann dann wieder und wieder ausgeführt werden – wann immer der Optimierer die Gradienten ermitteln muss. Zudem ist es so möglich, bei Bedarf höhere Ableitungen zu bestimmen.

 Wenn Sie eine neue Low-Level-Operation in TensorFlow in C++ implementieren und diese zu Autodiff kompatibel machen möchten, müssen Sie eine Funktion bereitstellen, die die partiellen Ableitungen der Funktionsausgaben bezüglich ihrer Eingaben bereitstellt. 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.

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 14](#)), Recurrent Neural Networks ([Kapitel 15](#)) oder Autoencoder ([Kapitel 17](#)) eingesetzt werden. Sie werden aber häufig in der Fachliteratur erwähnt und in manchen Gebieten noch immer eingesetzt, sodass es sich lohnt, sie zu kennen. Wir werden außerdem *Deep Belief Networks* 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 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. Es 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](#)); das heißt, jedes Neuron ist mit jedem anderen verbunden. Die Bilder im Diagramm sind 6×6 Pixel groß, daher sollte das neuronale Netz 36 Neuronen (und 630 Verbindungen) enthalten. Der Klarheit wegen ist ein deutlich kleineres Netz dargestellt.

Der Trainingsalgorithmus folgt der hebbischen Lernregel (siehe »Das Perzeptron«): 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.

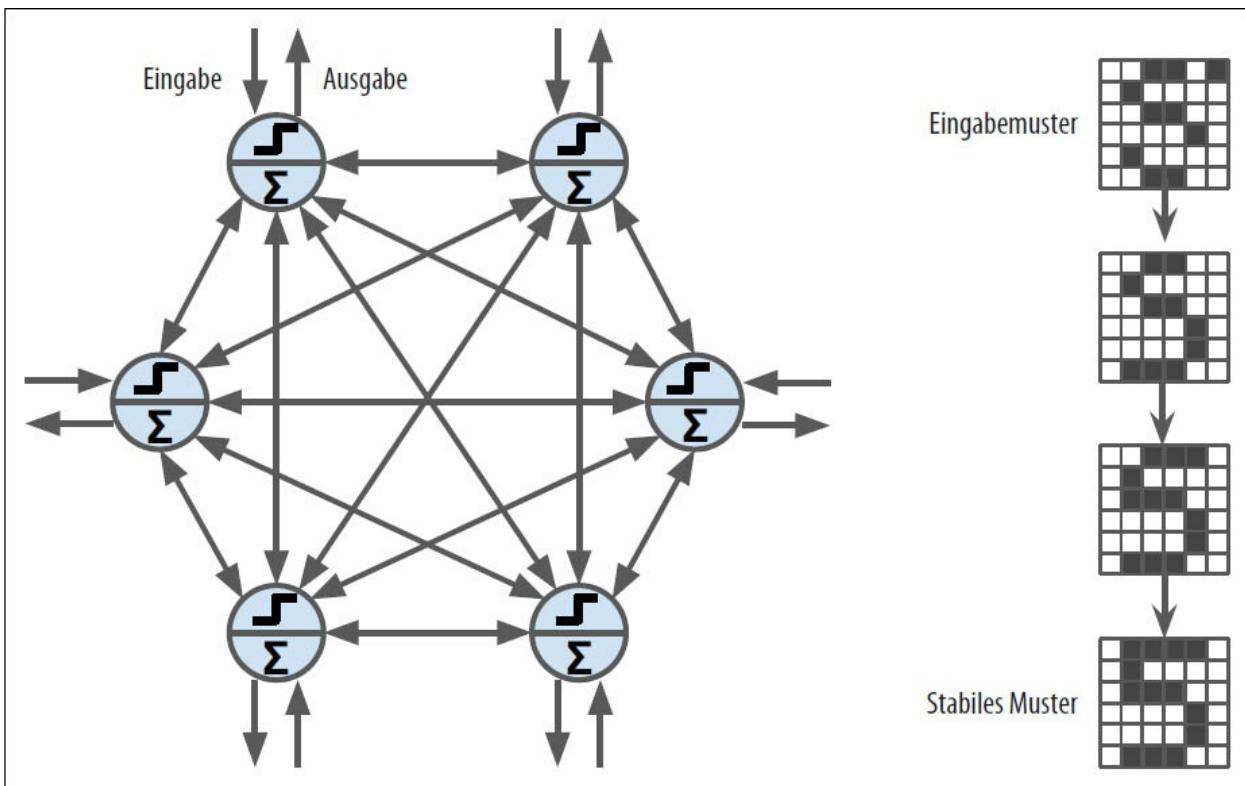


Abbildung E-1: Hopfield-Netz

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 306.936 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 es 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 $j.$ Neurons (0 oder 1).
- $w_{i,j}$ ist das Gewicht der Verbindung zwischen dem $i.$ und $j.$ Neuron. Beachten Sie, dass $w_{i,i} = 0$ gilt.
- b_i ist der Bias-Term des $i.$ 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, in dem 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 Beobachten des Zustands können wir ein breites Spektrum an Wahrscheinlichkeitsverteilungen simulieren. Da bezeichnet man als *generatives Modell*.

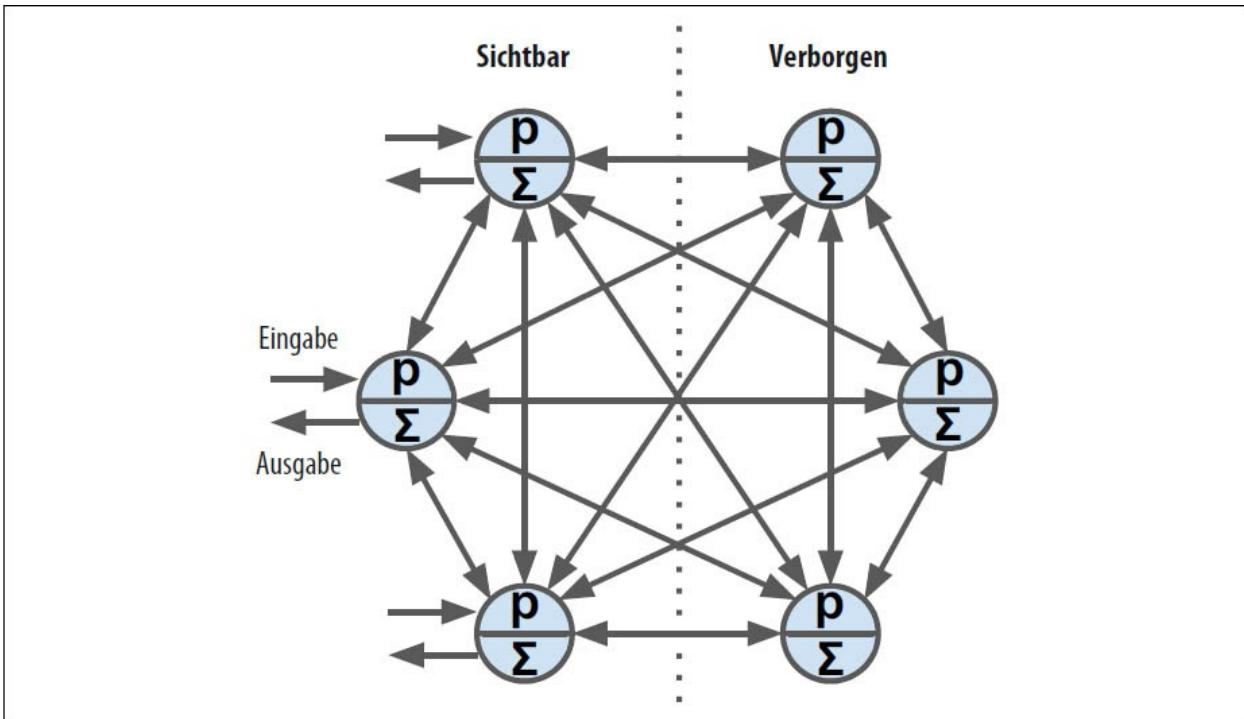


Abbildung E-2: Boltzmann Machine

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 verrausches 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 untereinander 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.

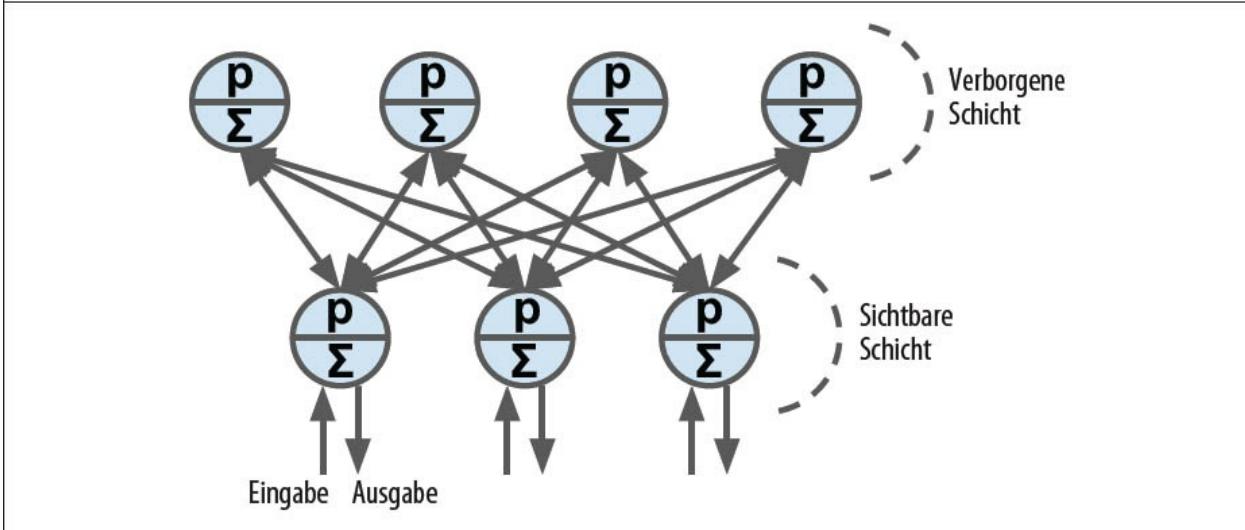


Abbildung E-3: Restricted Boltzmann Machine

Im Jahr 2005 wurde von Miguel Á. Carreira-Perpiñán und Geoffrey Hinton ein sehr effizienter Trainingsalgorithmus namens *Contrastive Divergence* vorgestellt. (<https://homl.info/135>)¹ 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 (siehe Formel E-1). Damit erhalten Sie den Hidden-Vektor \mathbf{h} (wobei h_i dem Zustand des i . 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, wobei η die Lernrate ist.

Formel E-2: Aktualisieren der Gewichte im Contrastive-Divergence-Algorithmus

$$w_{i,j} \leftarrow w_{i,j} + \eta(\mathbf{x}\mathbf{h}^T - \mathbf{x}'\mathbf{h}'^T)$$

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.

Deep Belief Networks

RBM 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 (<https://homl.info/136>)², der 2006 eine Flutwelle im Deep-Learning-Bereich auslöste.

Wie RBMs lernen auch DBNs, die Wahrscheinlichkeitsverteilung der Eingabedaten ohne Überwachung zu reproduzieren. Sie sind aber viel besser darin, und zwar 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 einfachere 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 erlernt 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 erlernt 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 Labelneuronen). Weitere RBMs ließen sich auf diese Weise hintereinanderschalten, 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 Trainingslabels zu assoziieren. Dies ist der überwachte Schritt.

Wenn Sie RBM 1 nach dem Training einen neuen Datenpunkt vorstellen, pflanzt sich dessen Signal zu RMB 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.

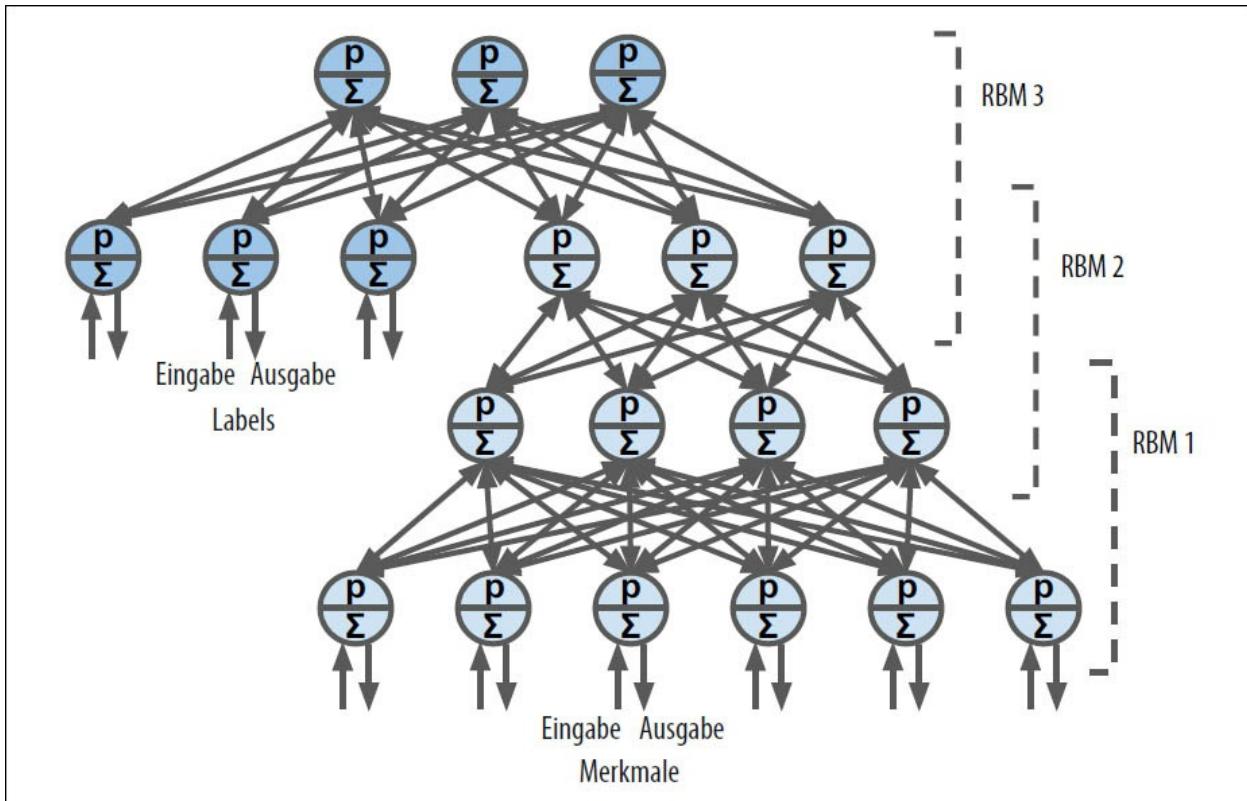


Abbildung E-4: Ein für halbüberwachtes Lernen konfiguriertes Deep Belief Network

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 von Objekten verbinden, die es bereits selbst erkennen kann. Sie müssen nicht auf jeden einzelnen Stuhl zeigen und »Stuhl« sagen; einige Beispiele reichen aus (gerade genug, somit 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 RMB 3 fort, dann weiter zu RMB 2 und RMB 1. Am Ende geben die Visible Units von RMB 1 einen neuen Datenpunkt aus. Dieser Datenpunkt sieht in der Regel 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 Bildern 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).

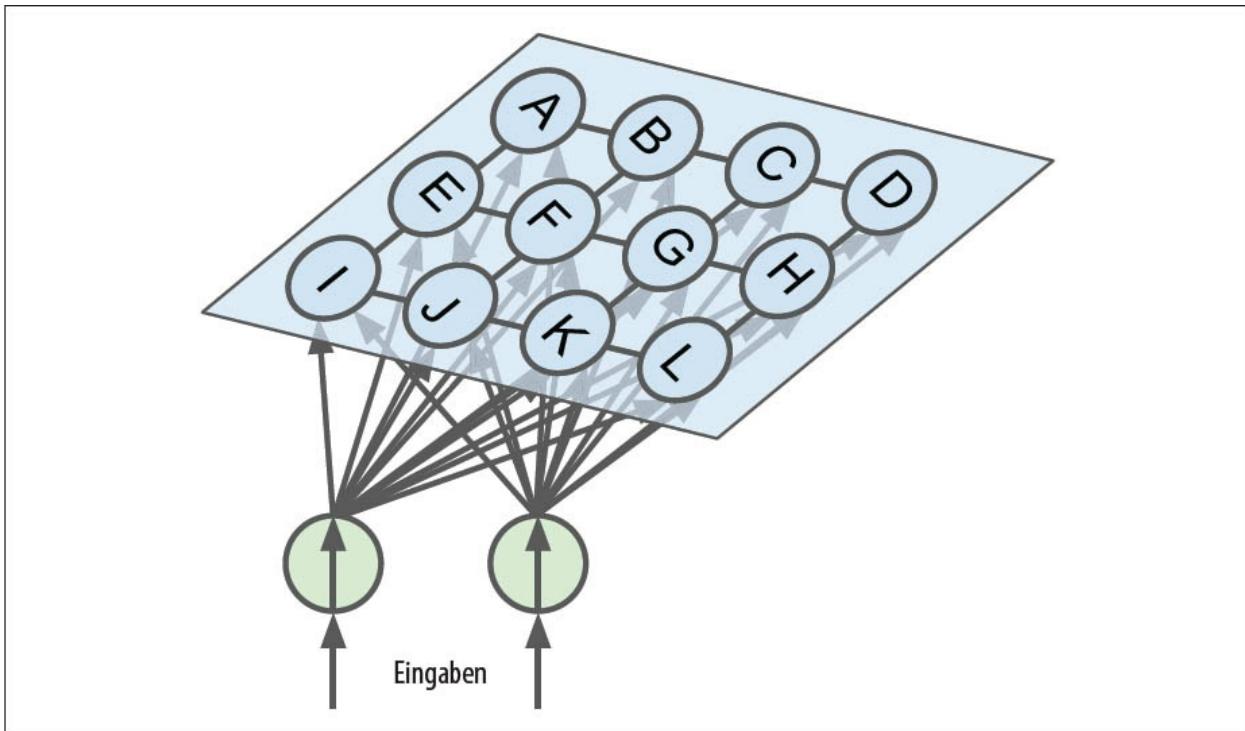


Abbildung E-5: Selbstorganisierende Karten

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, dessen Gewichtsvektor dem Eingabevektor am ähnlichsten ist. Im Allgemeinen aktivieren im Eingaberaum ähnliche Datenpunkte nahe beieinanderliegende Neuronen auf der Karte. Das macht SOMs für die 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 beieinanderliegende Neuronen nach und nach auf ähnliche Eingabedaten spezialisieren.⁴

Spezielle Datenstrukturen

In diesem Anhang wollen wir uns kurz die über normale Float- oder Integer-Tensoren hinausgehenden Datenstrukturen anschauen, die von TensorFlow unterstützt werden. Dazu gehören Strings, Ragged-Tensoren, Sparse-Tensoren, Tensor-Arrays, Sets und Queues.

Strings

Tensoren können Bytestrings enthalten, was besonders für die Verarbeitung natürlicher Sprache nützlich ist (siehe [Kapitel 16](#)):

```
>>> tf.constant(b"hello world")  
<tf.Tensor: id=149, shape=(), dtype=string, numpy=b'Hello Welt'>
```

Versuchen Sie, einen Tensor mit einem Unicodestring zu erstellen, konvertiert TensorFlow ihn automatisch nach UTF-8:

```
>>> tf.constant("café")  
<tf.Tensor: id=138, shape=(), dtype=string, numpy=b'caf\xc3\xa9'>
```

Es ist auch möglich, Tensoren zu erstellen, die Unicodestrings repräsentieren. Erzeugen Sie einfach ein Array mit 32-Bit-Integers, die jeweils einen einzelnen Unicode-Codepoint repräsentieren:¹

```
>>> tf.constant([ord(c) for c in "café"])  
<tf.Tensor: id=211, shape=(4,), dtype=int32,  
numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

- ▀ In Tensoren vom Typ `tf.string` ist die String-Länge nicht Teil der Form des Tensors. Mit anderen Worten: Strings werden als atomare Werte angesehen. In einem Unicodestring-Tensor (also einem `int32`-Tensor) ist die Länge des Strings tatsächlich Teil der Form des Tensors.

Das Paket `tf.strings` enthält eine Reihe von Funktionen zum Bearbeiten von String-Tensoren, wie zum Beispiel `length()`, um die Bytes in einem Bytestring zu zählen (oder die Anzahl an

Codepoints, wenn Sie `unit="UTF8_CHAR"` setzen), `uni_code_encode()` zum Konvertieren eines `UnicodeString`-Tensors (also eines `int32`-Tensors) in einen `ByteString`-Tensor und `unicode_decode()` für die Gegenrichtung:

```
>>> b = tf.strings.unicode_encode(u, "UTF-8")
>>> tf.strings.length(b, unit="UTF8_CHAR")
<tf.Tensor: id=386, shape=(), dtype=int32, numpy=4>
>>> tf.strings.unicode_decode(b, "UTF-8")
<tf.Tensor: id=393, shape=(4,), dtype=int32,
    numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

Sie können auch Tensoren mit mehreren Strings bearbeiten:

```
>>> p = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
>>> tf.strings.length(p, unit="UTF8_CHAR")
<tf.Tensor: id=299, shape=(4,), dtype=int32,
    numpy=array([4, 6, 5, 2], dtype=int32)>
>>> r = tf.strings.unicode_decode(p, "UTF8")
>>> r
tf.RaggedTensor(values=tf.Tensor(
[ 67   97   102   233   67   111   102   102   101   101   99   97
 102   102   232 21654 21857], shape=(17,), dtype=int32),
  row_splits=tf.Tensor([ 0  4 10 15 17], shape=(5,), dtype=int64))
>>> print(r)
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101],
  [99, 97, 102, 102, 232], [21654, 21857]]>
```

Beachten Sie, dass die decodierten Strings in einem `RaggedTensor` gespeichert werden. Was ist das?

Ragged-Tensoren

Ein `Ragged-Tensor` ist eine spezielle Form eines Tensors, die eine Liste von Arrays unterschiedlicher Größe repräsentiert. Allgemeiner gesprochen, handelt es sich um einen Tensor

mit einer oder mehreren »ausgefransten« Dimensionen, also solchen, bei denen die Slices unterschiedliche Längen haben können. Im Ragged-Tensor `r` ist die zweite Dimension eine ausgefranste Dimension. Bei allen Ragged-Tensoren ist die erste Dimension immer eine reguläre Dimension (auch als *einheitliche Dimension* bezeichnet).

Alle Elemente des Ragged-Tensors `r` sind reguläre Tensoren. Schauen wir uns beispielsweise das zweite Element des Ragged-Tensors an:

```
>>> print(r[1])
tf.Tensor([ 67 111 102 102 101 101], shape=(6,), dtype=int32)
```

Das Paket `tf.ragged` enthält eine Reihe von Funktionen, um Ragged-Tensoren zu erstellen und zu bearbeiten. Erzeugen wir einen zweiten Ragged-Tensor mit `tf.ragged.constant()` und verketten wir ihn mit dem ersten Ragged-Tensor entlang der Achse 0:

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]])
>>> print(tf.concat([r, r2], axis=0))
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

Das Resultat ist nicht allzu überraschend: Die Tensoren in `r2` wurden hinter den Tensoren in `r` entlang von Achse 0 angefügt. Aber was passiert, wenn wir `r` und einen anderen Ragged-Tensor entlang der Achse 1 verketten?

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]])
>>> print(tf.concat([r, r3], axis=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67, 111, 102, 102, 101, 101,
71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

Dieses Mal sehen Sie, dass der *i*. Tensor in `r` und der *i*. Tensor in `r3` verkettet wurden. Das ist nun ungewöhnlicher, da alle diese Tensoren unterschiedliche Längen haben können.

Rufen Sie die Methode `to_tensor()` auf, wird er zu einem regulären Tensor umgewandelt, wobei kürzere Tensoren mit Nullen aufgefüllt werden, um Tensoren gleicher Länge zu bekommen (Sie können den Standardwert ändern, indem Sie das Argument `default_value` setzen):

```
>>> r.to_tensor()
<tf.Tensor: id=1056, shape=(4, 6), dtype=int32, numpy=
array([[ 67,    97,   102,   233,      0,      0],
```

```
[ 67,   111,   102,   102,   101,   101],
[ 99,   97,   102,   102,   232,     0],
[21654, 21857,     0,     0,     0,     0]], dtype=int32)>
```

Viele TF-Operationen unterstützen Ragged-Tensoren. Eine vollständige Liste finden Sie in der Dokumentation der Klasse `tf.RaggedTensor`.

Sparse-Tensoren

TensorFlow kann auch *Sparse-Tensoren* effizient verwalten (sogenannte dünn besetzte Tensoren, die größtenteils mit Nullen besetzt sind). Erstellen Sie einfach einen `tf.SparseTensor`, geben Sie die Indizes und Werte der Elemente an, die nicht null sind, und definieren Sie die Form des Tensors. Die Indizes müssen in normaler »Lesereihenfolge« aufgeführt werden (von links nach rechts und oben nach unten). Wenn Sie sich nicht sicher sind, nutzen Sie einfach `tf.sparse.reorder()`. Sie können einen Sparse-Tensor zu einem Dense-Tensor (also einen normalen Tensor) umwandeln, indem Sie `tf.sparse.to_dense()` verwenden:

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
                           values=[1., 2., 3.],
                           dense_shape=[3, 4])

>>> tf.sparse.to_dense(s)

<tf.Tensor: id=1074, shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

Beachten Sie, dass Sparse-Tensoren nicht so viele Operationen wie Dense-Tensoren unterstützen. So können Sie beispielsweise einen Sparse-Tensor mit einem Skalarwert multiplizieren, um einen neuen Sparse-Tensor zu erhalten, aber Sie können keinen Skalarwert addieren, da dies nicht wieder einen Sparse-Tensor zurückliefern würde:

```
>>> s * 3.14
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x13205d470>
>>> s + 42.0
[...] TypeError: unsupported operand type(s) for +: 'SparseTensor' and 'float'
```

Tensor-Arrays

Ein `tf.TensorArray` repräsentiert eine Liste mit Tensoren. Das kann in dynamischen Modellen mit Schleifen nützlich sein, um Ergebnisse zu sammeln und später Statistiken zu berechnen. Sie können Tensoren im Array an beliebigen Positionen lesen und schreiben:

```
array = tf.TensorArray(dtype=tf.float32, size=3)

array = array.write(0, tf.constant([1., 2.]))

array = array.write(1, tf.constant([3., 10.]))

array = array.write(2, tf.constant([5., 7.]))

tensor1 = array.read(1) # => liefert (und entfernt!) tf.constant([3., 10.])
```

Beachten Sie, dass das Lesen eines Eintrags diesen vom Array entfernt und durch einen Tensor gleicher Form, gefüllt mit Nullen, ersetzt.



Schreiben Sie in das Array, müssen Sie die Ausgabe wieder dem Array zuweisen, wie in diesem Codebeispiel zu sehen ist. Tun Sie das nicht, wird Ihr Code zwar im Eager-Modus funktionieren, nicht aber im Graph-Modus (diese Modi wurden in [Kapitel 12](#) vorgestellt).

Erstellen Sie ein `TensorArray`, müssen Sie dessen `size` angeben – außer im Graph-Modus. Alternativ lassen Sie `size` undefiniert und setzen stattdessen `dynamic_size=True`, aber das beeinträchtigt die Performance. Kennen Sie also die `size` im Voraus, sollten Sie sie setzen. Sie müssen auch den `dtype` angeben, und alle Elemente müssen die gleiche Form wie das haben, das als Erstes ins Array geschrieben wird.

Sie können alle Einträge in einem regulären Tensor stapeln, indem Sie die Methode `stack()` aufrufen:

```
>>> array.stack()

<tf.Tensor: id=2110875, shape=(3, 2), dtype=float32, numpy=
array([[1., 2.],
       [0., 0.],
       [5., 7.]], dtype=float32)>
```

Sets

TensorFlow unterstützt Sets mit Integer- oder String-Werten (aber nicht mit Floats). Es nutzt dazu normale Tensoren. So wird beispielsweise das Set $\{1, 5, 9\}$ durch den Tensor $[[1, 5, 9]]$ repräsentiert. Beachten Sie, dass der Tensor mindestens zwei Dimensionen besitzen muss, und die Sets müssen sich in der letzten Dimension befinden. So ist zum Beispiel $[[1, 5, 9],$

[2, 5, 11]] ein Tensor mit zwei unabhängigen Sets: {1, 5, 9} und {2, 5, 11}. Wenn manche Sets kürzer sind als andere, müssen Sie sie mit einem Padding-Wert auffüllen (standardmäßig ist dieser 0, aber Sie können auch einen anderen wählen).

Das Paket `tf.sets` enthält viele Funktionen zum Bearbeiten von Sets. Erstellen wir zwei Sets und berechnen wir deren Vereinigungsmenge (das Ergebnis ist ein Sparse-Tensor, daher werden wir `to_dense()` aufrufen, um ihn anzuzeigen):

```
>>> a = tf.constant([[1, 5, 9]])  
>>> b = tf.constant([[5, 6, 9, 11]])  
>>> u = tf.sets.union(a, b)  
>>> u  
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x132b60d30>  
>>> tf.sparse.to_dense(u)  
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11]], dtype=int32)>
```

Sie können auch die Vereinigungsmenge mehrerer Set-Paare gleichzeitig berechnen:

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]])  
>>> b = tf.constant([[5, 6, 9, 11], [13, 0, 0, 0]])  
>>> u = tf.sets.union(a, b)  
>>> tf.sparse.to_dense(u)  
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],  
[ 0, 10, 13,  0,  0]], dtype=int32)>
```

Möchten Sie einen anderen Padding-Wert verwenden, müssen Sie beim Aufruf von `to_dense()` den `default_value` setzen:

```
>>> tf.sparse.to_dense(u, default_value=-1)  
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],  
[ 0, 10, 13, -1, -1]], dtype=int32)>
```



Der Standard für `default_value` ist 0. Arbeiten Sie mit String-Sets, müssen Sie den `default_value` auf jeden Fall setzen (zum Beispiel auf einen leeren String).

Andere verfügbare Funktionen in `tf.sets` sind `difference()`, `intersection()` und `size()`, die selbsterklärend sind. Wollen Sie prüfen, ob ein Set bestimmte Werte enthält,

können Sie die Schnittmenge zwischen dem Set und den Werten bilden. Wollen Sie Werte zu einem Set hinzufügen, können Sie die Vereinigungsmenge zwischen dem Set und den Werten berechnen.

Queues

Eine Queue ist eine Datenstruktur, in die Sie Datensätze schieben und später wieder herausholen können. TensorFlow implementiert verschiedene Queue-Typen im Paket `tf.queue`. Sie waren beim Implementieren effizienter Pipelines zum Laden und Vorverarbeiten von Daten sehr wichtig, aber die `tf.data`-API hat sie mehr oder weniger überflüssig gemacht (außer vielleicht in wenigen Fällen), weil sie viel einfacher zu nutzen ist und alle Tools bereitstellt, die Sie zum Bauen effizienter Pipelines brauchen. Aus Gründen der Vollständigkeit wollen wir sie uns aber kurz anschauen.

Die einfachste Form der Queue ist die First-In/First-Out-Queue (FIFO). Um sie zu bauen, müssen Sie die Maximalzahl an Datensätzen angeben, die sie enthalten kann. Zudem handelt es sich bei jedem Datensatz um ein Tupel aus Tensoren, daher müssen Sie den Typ jedes Tensors und optional ihre Formen festlegen. Der folgende Code erstellt beispielsweise eine FIFO-Queue mit maximal drei Datensätzen, die jeweils aus einem Tupel mit einem 32-Bit-Integer und einem String bestehen. Dann schiebt er zwei Datensätze hinein, schaut sich die Größe an (in dem Moment 2) und holt einen Datensatz heraus:

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], shapes=[(), ()])  
>>> q.enqueue([10, b"windy"])  
>>> q.enqueue([15, b"sunny"])  
>>> q.size()  
<tf.Tensor: id=62, shape=(), dtype=int32, numpy=2>  
>>> q.dequeue()  
[<tf.Tensor: id=6, shape=(), dtype=int32, numpy=10>,  
<tf.Tensor: id=7, shape=(), dtype=string, numpy=b'windy'>]
```

Es ist auch möglich, mehrere Datensätze auf einmal einzustellen oder herauszuholen (für Letzteres müssen Sie beim Erstellen der Queue die Formen angeben):

```
>>> q.enqueue_many([[13, 16], [b'cloudy', b'rainy']])  
>>> q.dequeue_many(3)  
[<tf.Tensor: [...] numpy=array([15, 13, 16], dtype=int32)>,  
<tf.Tensor: [...] numpy=array([b'sunny', b'cloudy', b'rainy'], dtype=object)>]
```

Andere Queue-Typen sind:

PaddingFIFOQueue

Wie eine FIFOQueue, aber die Methode `dequeue_many()` unterstützt das Herausholen mehrerer Datensätze unterschiedlicher Form. Sie füllt die kürzesten Datensätze automatisch auf, um sicherzustellen, dass im Batch alle die gleiche Form besitzen.

PriorityQueue

Eine Queue, die Datensätze priorisiert herausholt. Die Priorität muss eine 64-Bit-Ganzzahl sein, die das erste Element jedes Datensatzes ist. Überraschenderweise werden Datensätze mit einer niedrigeren Priorität zuerst herausgegeben. Datensätze mit der gleichen Priorität werden in FIFO-Reihenfolge ausgegeben.

RandomShuffleQueue

Eine Queue, deren Datensätze in zufälliger Reihenfolge herausgegeben werden. Das war nützlich, um einen durchmischten Puffer zu implementieren, bevor `tf.data` eingeführt wurde.

Ist eine Queue schon voll und versuchen Sie, einen weiteren Datensatz zu übergeben, wird die Methode `enqueue*()` stehen bleiben, bis ein Datensatz von einem anderen Thread herausgeholt wurde. Ist umgekehrt eine Queue leer und versuchen Sie, einen Datensatz herauszuholen, bleibt die Methode `dequeue*()` stehen, bis Datensätze von einem anderen Thread in die Queue geschoben wurden.

TensorFlow-Graphen

In diesem Anhang werden wir uns die Graphen anschauen, die von TF Functions erzeugt werden (siehe [Kapitel 12](#)).

TF Functions und konkrete Funktionen

TF Functions sind polymorph, das heißt, sie unterstützen Eingaben unterschiedlichen Typs (und mit verschiedenen Formen). Schauen Sie sich beispielsweise die folgende Form `tf_cube()` an:

```
@tf.function  
def tf_cube(x):  
    return x ** 3
```

Jedes Mal, wenn Sie eine TF Function mit einer neuen Kombination aus Eingabetypen oder -formen aufrufen, generiert sie eine neue *konkrete Funktion* mit ihrem eigenen Graphen, der auf diese bestimmte Kombination spezialisiert ist. Solche eine Kombination aus Argumenttypen und -formen wird als *Eingabesignatur* bezeichnet. Rufen Sie die TF Function mit einer Eingabesignatur auf, die sie schon gesehen hat, wird sie die zuvor erzeugte konkrete Funktion wiederverwenden. Rufen Sie beispielsweise `tf_cube(tf.constant(3.0))` auf, wird die TF Function die gleiche konkrete Funktion einsetzen, die sie schon für `tf_cube(tf.constant(2.0))` genutzt hat (für skalare float32-Tensoren). Aber sie wird eine neue konkrete Funktion generieren, wenn Sie `tf_cube(tf.constant([2.0]))` oder `tf_cube(tf.constant([3.0]))` (für float32-Tensoren der Form [1]) aufrufen und noch eine andere für `tf_cube(tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (für float32-Tensoren der Form [2, 2]). Sie können die konkrete Funktion für eine bestimmte Kombination aus Eingabewerten erhalten, indem Sie die Methode `get_concrete_function()` der TF Function aufrufen. Sie kann wie eine reguläre Funktion aufgerufen werden, unterstützt aber nur eine Eingabesignatur (in diesem Beispiel skalare float32-Tensoren):

```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))  
>>> concrete_function  
<tensorflow.python.eager.function.ConcreteFunction at 0x155c29240>  
>>> concrete_function(tf.constant(2.0))
```

```
<tf.Tensor: id=19068249, shape=(), dtype=float32, numpy=8.0>
```

Abbildung G-1 zeigt die TF Function `tf_cube()`, nachdem wir `tf_cube(2)` und `tf_cube(tf.constant(2.0))` aufgerufen haben: Es wurden zwei konkrete Funktionen generiert – eine für jede Signatur, jeweils mit ihrem eigenen optimierten Funktionsgraphen (`FuncGraph`) und ihrer eigenen Funktionsdefinition (`FunctionDef`). Eine Funktionsdefinition verweist auf die Teile des Graphen, die zu den Eingaben und Ausgaben der Funktion gehören. In jedem `FuncGraph` repräsentieren die Knoten (Ovale) Operationen (zum Beispiel Potenzieren, Konstanten oder Platzhalter für Argumente wie `x`), während die durchgehenden Pfeile zwischen den Operationen für die Tensoren stehen, die durch den Graphen fließen. Die linke konkrete Funktion ist für $x = 2$ spezialisiert, daher konnte TensorFlow sie so vereinfachen, dass sie immer 8 ausgibt (beachten Sie, dass die Funktionsdefinition nicht einmal Eingabewerte besitzt). Die rechte konkrete Funktion ist auf skalare float32-Tensoren spezialisiert und kann nicht weiter vereinfacht werden. Rufen wir `tf_cube(tf.constant(5.0))` auf, wird die zweite konkrete Funktion aufgerufen, die Platzhalteroperation für `x` wird 5,0 zurückgeben, dann wird die Potenzieroperation $5.0^{**} 3$ berechnen, und die Ausgabe wird 125,0 sein.

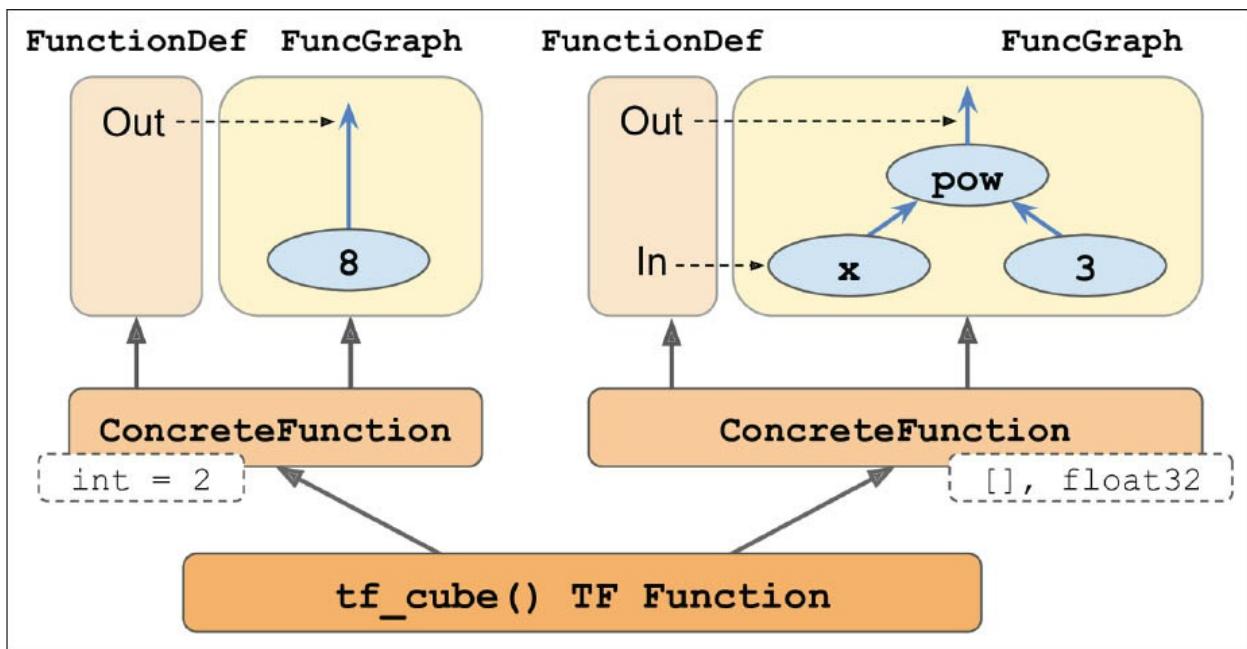


Abbildung G-1: Die TF Function `tf_cube()` mit ihren `ConcreteFunctions` und deren `FunctionGraphs`

Die Tensoren in diesen Graphen sind *symbolische Tensoren*, was heißt, dass sie keinen echten Wert enthalten, sondern nur einen Datentyp, eine Form und einen Namen. Sie stehen für die zukünftigen Tensoren, die durch den Graphen fließen werden, sobald der Platzhalter `x` mit einem echten Wert gefüttert und der Graph ausgeführt wird. Symbolische Tensoren ermöglichen es, im Voraus zu definieren, wie Operationen verbunden werden, und mit ihnen kann TensorFlow rekursiv die Datentypen und -formen aller Tensoren ermitteln, wenn die Datentypen und -formen der Eingabewerte gegeben sind.

Schauen wir weiter hinter die Kulissen, um herauszufinden, wie wir auf Funktionsdefinitionen und Funktionsgraphen zugreifen und die Operationen und Tensoren eines Graphen untersuchen können.

Funktionsdefinitionen und -graphen untersuchen

Sie können auf den Rechengraphen einer konkreten Funktion über das Attribut `graph` zugreifen und erhalten dann die Liste seiner Operationen durch den Aufruf von dessen Methode `get_operations()`:

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x14db5ef98>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

In diesem Beispiel steht die erste Operation für das Eingabeargument `x` (sie wird als *Platzhalter* bezeichnet), die zweite »Operation« für die Konstante 3, die dritte repräsentiert die Potenzoperation (`**`), und die letzte Operation steht für die Ausgabe dieser Funktion (es ist eine Identitätsoperation – sie tut nicht mehr, als die Ausgabe der Potenzoperation zu kopieren.¹⁾) Jede Operation hat eine Liste von Ein- und Ausgabetensoren, auf die Sie über die Attribute `inputs` und `outputs` der Operationen zugreifen können. Holen wir uns beispielsweise die Liste der Ein- und Ausgaben der Potenzoperation:

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

Dieser Rechengraph ist in [Abbildung G-2](#) dargestellt.

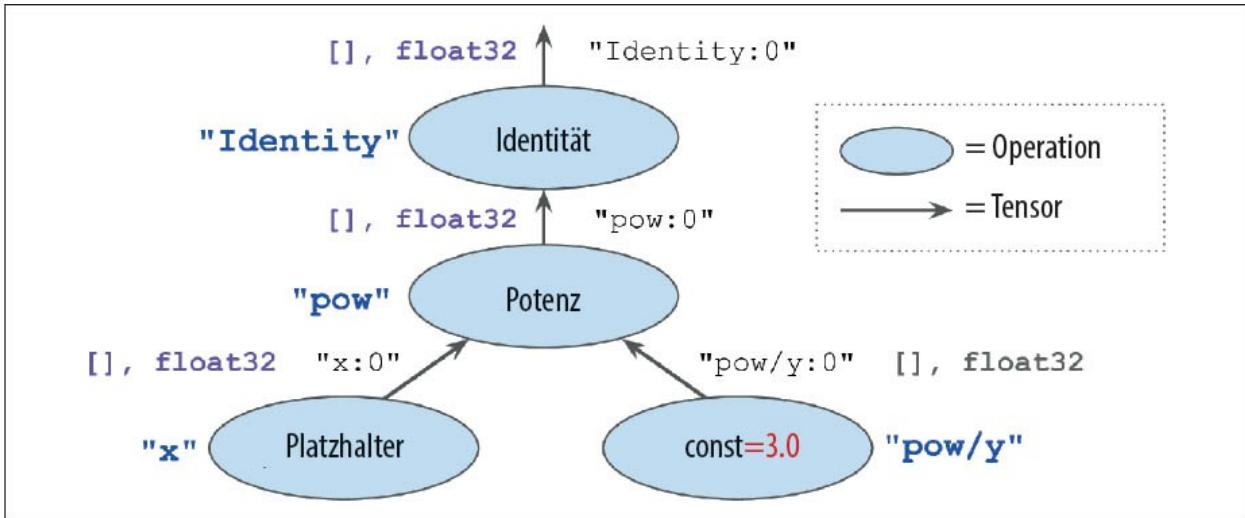


Abbildung G-2: Beispiel für einen Rechengraphen

Beachten Sie, dass jede Operation einen Namen besitzt. Standardmäßig ist es der Name der Operation (zum Beispiel "pow"), aber Sie können ihn auch manuell definieren, wenn Sie die Operation aufrufen (zum Beispiel `tf.pow(x, 3, name="other_name")`). Ist ein Name schon vorhanden, fügt TensorFlow automatisch einen eindeutigen Index hinzu (zum Beispiel "pow_1", "pow_2" und so weiter). Auch jeder Tensor hat einen eindeutigen Namen: Es ist immer der Name der Operation, die diesen Tensor ausgibt, plus :0, wenn es die erste Ausgabe der Operation ist, :1 für die zweite Ausgabe und so weiter. Sie können eine Operation oder einen Tensor über den Namen finden, indem Sie die Methoden `get_operation_by_name()` oder `get_tensor_by_name()` des Graphen aufrufen:

```

>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>

```

Die konkrete Funktion enthält auch die Funktionsdefinition (als Protocol Buffer²), in der sich auch die Funktionssignatur findet. Durch diese Signatur weiß die konkrete Funktion, welche Platzhalter mit den Eingabewerten zu füttern und welche Tensoren zurückzugeben sind:

```

>>> concrete_function.function_def.signature
name: "__inference_cube_19068241"
input_arg {
    name: "x"
    type: DT_FLOAT
}

```

```

}

output_arg {
    name: "identity"
    type: DT_FLOAT
}

```

Beschäftigen wir uns nun genauer mit dem Tracing.

Tracing

Wir verändern die Funktion `tf_cube()` so, dass sie ihre Eingabewerte ausgibt:

```

@tf.function

def tf_cube(x):
    print("x =", x)
    return x ** 3

```

Rufen wir sie auf:

```

>>> result = tf_cube(tf.constant(2.0))

x = Tensor("x:0", shape=(), dtype=float32)

>>> result

<tf.Tensor: id=19068290, shape=(), dtype=float32, numpy=8.0>

```

Das `result` sieht gut aus, aber wir sehen, was ausgegeben wurde: `x` ist ein symbolischer Tensor! Er hat eine Form und einen Datentyp, aber keinen Wert. Zudem hat er einen Namen ("`x:0`"). Das liegt daran, dass es sich bei der Funktion `print()` nicht um eine TensorFlow-Operation handelt, daher wird sie nur ausgeführt, wenn die Python-Funktion getraced wird. Das wiederum passiert im Graph-Modus, in dem die Argumente durch symbolische Tensoren ersetzt wurden (gleicher Typ und gleiche Form, aber kein Wert). Da die Funktion `print()` nicht im Graphen erfasst wurde, wird beim nächsten Aufruf von `tf_cube()` mit skalaren float32-Tensoren nichts ausgegeben:

```

>>> result = tf_cube(tf.constant(3.0))

>>> result = tf_cube(tf.constant(4.0))

```

Rufen wir hingegen `tf_cube()` mit einem Tensor eines anderen Typs oder einer anderen Form auf oder mit einem neuen Python-Wert, wird die Funktion erneut getraced und die Funktion

`print()` wieder aufgerufen:

```
>>> result = tf_cube(2) # neuer Python-Wert: Trace!
x = 2

>>> result = tf_cube(3) # neuer Python-Wert: Trace!
x = 3

>>> result = tf_cube(tf.constant([[1., 2.]])) # Neue Form: Trace!
x = Tensor("x:0", shape=(1, 2), dtype=float32)

>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]])) # Neue Form: Trace!
x = Tensor("x:0", shape=(None, 2), dtype=float32)

>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]])) # Gleiche Form:
kein Trace
```



Hat Ihre Funktion Nebeneffekte in Python (speichert sie zum Beispiel Logs auf der Festplatte), achten Sie darauf, dass dieser Code nur läuft, wenn die Funktion getraced wird (also jedes Mal, wenn die TF Function mit einer neuen Eingabesignatur aufgerufen wird). Am besten gehen Sie davon aus, dass die Funktion bei jedem Aufruf der TF Function getraced werden könnte (oder auch nicht).

In manchen Fällen wollen Sie eine TF Function auf eine bestimmte Eingabesignatur beschränken. Nehmen wir beispielsweise an, Sie wissen, dass Sie eine Funktion immer nur mit Batches aus 28×28 -Pixel-Bildern aufrufen werden, diese Batches aber sehr unterschiedliche Größen haben können. Sie wollen eventuell nicht, dass TensorFlow für jede Batchgröße eine andere konkrete Funktion generiert oder selbst versucht, herauszufinden, wann es `None` nutzen kann. In diesem Fall können Sie die Eingabesignatur wie folgt festlegen:

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])  
def shrink(images):  
    return images[:, ::2, ::2] # die Hälfte der Zeilen und Spalten verwerfen
```

Diese TF Function wird nur `float32`-Tensoren der Form $[*, 28, 28]$ akzeptieren und immer die gleiche konkrete Funktion wiederverwenden:

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])  
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])  
preprocessed_images = shrink(img_batch_1) # Funktioniert. Traced die Funktion.
```

```
preprocessed_images = shrink(img_batch_2) # Funktioniert.  
Gleiche konkrete Funktion.
```

Versuchen Sie aber, diese TF Function mit einem Python-Wert oder einem Tensor mit einem unerwarteten Datentyp oder einer anderen Form aufzurufen, werden Sie eine Exception erhalten:

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])  
preprocessed_images = shrink(img_batch_3) # ValueError! Unerwartete Signatur.
```

Den Kontrollfluss mit AutoGraph erfassen

Was passiert, wenn Ihre Funktion eine einfache `for`-Schleife enthält? Schreiben wir eine Funktion, die zum Eingabewert 10 addiert, indem sie 10 Mal eine 1 addiert:

```
@tf.function  
  
def add_10(x):  
    for i in range(10):  
        x += 1  
  
    return x
```

Das funktioniert problemlos, aber wenn wir uns den Graphen anschauen, erkennen wir, dass gar keine Schleife enthalten ist – sondern nur 10 Additionsoperationen!

```
>>> add_10(tf.constant(0))  
<tf.Tensor: id=19280066, shape=(), dtype=int32, numpy=10>  
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()  
[<tf.Operation 'x' type=Placeholder>, [...],  
 <tf.Operation 'add' type=Add>, [...],  
 <tf.Operation 'add_1' type=Add>, [...],  
 <tf.Operation 'add_2' type=Add>, [...],  
 [...]  
 <tf.Operation 'add_9' type=Add>, [...],  
 <tf.Operation 'Identity' type=Identity>]
```

Das ist tatsächlich sinnvoll: Wird die Funktion getraced, läuft die Schleife 10 Mal, daher wird

die Operation `x += 1` genau 10 Mal ausgeführt, und da sie sich im Graph-Modus befand, wurde diese Operation im Graphen 10 Mal erfasst. Sie können sich diese `for`-Schleife als »statische« Schleife vorstellen, die beim Erstellen des Graphen entrollt wird.

Soll der Graph stattdessen eine »dynamische« Schleife enthalten (also eine, die läuft, wenn der Graph ausgeführt wird), können Sie sie manuell mit der Operation `tf.while_loop()` erzeugen, aber das ist nicht sehr intuitiv (siehe den Abschnitt »Using AutoGraph to Capture Control Flow« im Notebook zu [Kapitel 12](#)). Stattdessen ist es viel einfacher, TensorFlows *AutoGraph*-Feature zu nutzen, das in [Kapitel 12](#) besprochen wurde. AutoGraph ist standardmäßig aktiv (müssen Sie es einmal abschalten, können Sie `autograph=False` an `tf.function()` übergeben). Wenn es also an ist – warum erfasst es dann nicht die `for`-Schleife in der Funktion `add_10()`? Nun, sie erfasst nur `for`-Schleifen, die über `tf.range()` iterieren, nicht aber solche, die über `range()` laufen. So haben Sie die Wahl:

- Nutzen Sie `range()`, wird die `for`-Schleife statisch sein und nur ausgeführt, wenn die Funktion getraced wird. Wie wir gesehen haben, wird die Schleife dann in eine Folge von Operationen für jede Iteration »entrollt«.
- Nutzen Sie `tf.range()`, wird die `for`-Schleife dynamisch sein und selbst im Graphen zu finden sein (aber nicht während des Tracings laufen).

Schauen wir uns den Graphen an, der erzeugt wird, wenn Sie in der Funktion `add_10()` einfach `range()` durch `tf.range()` ersetzen:

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()  
[<tf.Operation 'x' type=Placeholder>, [...],  
 <tf.Operation 'range' type=Range>, [...],  
 <tf.Operation 'while' type=While>, [...],  
 <tf.Operation 'Identity' type=Identity>]
```

Wie Sie sehen, enthält der Graph nun eine `While`-Schleifen-Operation – so als hätten Sie die Funktion `tf.while_loop()` aufgerufen.

Variablen und andere Ressourcen in TF Functions

In TensorFlow werden Variablen und andere zustandsbehaftete Objekte, wie zum Beispiel Queues oder Datasets, als *Ressourcen* bezeichnet. TF Functions behandeln sie mit besonderer Sorgfalt: Jede Operation, die eine Ressource liest oder aktualisiert, wird als zustandsbehaftet betrachtet, und TF Functions stellen sicher, dass solche Operationen in der Reihenfolge ausgeführt werden, in der sie im Code erscheinen (im Gegensatz zu zustandslosen Operationen, die parallel laufen können, sodass ihre Ausführungsreihenfolge nicht garantiert ist). Übergeben Sie eine Ressource als Argument an eine TF Function, wird sie zudem als Referenz übergeben, sodass die Funktion sie verändern kann. Ein Beispiel:

```

counter = tf.Variable(0)

@tf.function

def increment(counter, c=1):
    return counter.assign_add(c)

increment(counter) # counter ist jetzt 1
increment(counter) # counter ist jetzt 2

```

Schauen Sie sich die Funktionsdefinition an, ist das erste Argument als Ressource gekennzeichnet:

```

>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
name: "counter"
type: DT_RESOURCE

```

Es ist auch möglich, eine `tf.Variable` zu verwenden, die außerhalb der Funktion definiert wurde, ohne sie explizit als Argument zu übergeben:

```

counter = tf.Variable(0)

@tf.function

def increment(c=1):
    return counter.assign_add(c)

```

Die TF Function behandelt sie als implizites erstes Argument, daher führt das tatsächlich zur gleichen Signatur wie oben (abgesehen vom Namen des Arguments). Aber der Einsatz globaler Variablen kann schnell ins Chaos führen, daher sollten Sie Variablen (und andere Ressourcen) im Allgemeinen in Klassen verpacken. Das Gute ist, dass `@tf.function` auch mit Methoden funktioniert:

```

class Counter:

    def __init__(self):
        self.counter = tf.Variable(0)

    @tf.function

```

```
def increment(self, c=1):
    return self.counter.assign_add(c)
```

- 💡 Verwenden Sie nicht `=`, `+=`, `-=` oder andere Zuweisungsoperatoren von Python für TF-Variablen. Stattdessen müssen Sie die Methoden `assign()`, `assign_add()` oder `assign_sub()` einsetzen. Versuchen Sie, einen Zuweisungsoperator von Python zu nutzen, werden Sie beim Aufruf der Methode eine Exception erhalten.

Ein gutes Beispiel für diesen objektorientierten Ansatz ist natürlich tf.keras. Schauen wir uns an, wie wir TF Functions in tf.keras verwenden.

TF Functions mit tf.keras (nicht) verwenden

Standardmäßig wird jede eigene Funktion, jede Schicht und jedes Modell, das Sie mit tf.keras verwenden, automatisch in eine TF Function umgewandelt – Sie müssen dafür gar nichts tun! Aber in manchen Fällen wollen Sie diese automatische Konvertierung deaktivieren – zum Beispiel wenn Ihr eigener Code nicht in eine TF Function umgewandelt werden kann oder wenn Sie Ihren Code nur debuggen wollen, was im Eager-Modus viel einfacher ist. Dazu können Sie einfach `dynamic=True` übergeben, wenn Sie das Modell oder eine seiner Schichten erstellen:

```
model = MyModel(dynamic=True)
```

Ist Ihr eigenes Modell oder Ihre eigene Schicht immer dynamisch, rufen Sie stattdessen den Konstruktor der Basisklasse mit `dynamic=True` auf:

```
class MyLayer(keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
    [...]
```

Alternativ können Sie `run_eagerly=True` übergeben, wenn Sie die Methode `compile()` aufrufen:

```
model.compile(loss=my_mse, optimizer="nadam", metrics=[my_mae],
               run_eagerly=True)
```

Jetzt wissen Sie, wie TF Functions mit Polymorphismus umgehen (mit mehreren konkreten Funktionen), wie Graphen automatisch durch AutoGraph und Tracing generiert werden, wie Graphen aussehen, wie Sie deren symbolische Operationen und Tensoren untersuchen können, wie Sie mit Variablen und Ressourcen umgehen und wie Sie TF Functions mit tf.keras verwenden.

Fußnoten

Vorwort

- 1 Geoffrey Hinton et al., »A Fast Learning Algorithm for Deep Belief Nets«, *Neural Computation* 18 (2006): 1527–1554.
- 2 Obwohl die Konvolutionsnetze von Yann Lecun bei der Bilderkennung seit den 1990ern gut funktioniert hatten, auch wenn sie nicht allgemein anwendbar waren.

Kapitel 1 Die Machine-Learning-Umgebung

- 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 Vortrainieren.
- 3 Beachten Sie, dass die Tiere recht gut von Fahrzeugen unterschieden werden und wie nahe sich Pferde und Rehe sind, aber wie weit entfernt von Vögeln. Abbildung mit Erlaubnis reproduziert, Quelle: Richard Socher et al., »Zero-Shot Learning Through Cross-Modal Transfer«, *Proceedings of the 26th International Conference on Neural Information Processing Systems* 1 (2013): 935–943.
- 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 ein paar Cluster von Hand aufräumen müssen.
- 5 Der griechische Buchstabe θ (Theta) wird konventionsgemäß häufig zum Darstellen von Modellparametern verwendet.
- 6 Die Definition der Funktion `prepare_country_stats()` ist hier nicht gezeigt (Sie finden sie im Jupyter-Notebook zu diesem Kapitel, wenn Sie sie in all ihrer Schönheit betrachten wollen). Es handelt es nur um langweiligen pandas-Code, der die Daten zur Zufriedenheit der OECD mit den BIP-Daten des IMF verbindet.
- 7 Es ist in Ordnung, wenn Sie nicht gleich den gesamten Code nachvollziehen können; wir werden Scikit-Learn in den folgenden Kapiteln vorstellen.
- 8 Beispielsweise aus dem Kontext zu ermitteln, ob man »to«, »two« oder »too« schreiben muss.
- 9 Die Abbildung wurde mit Erlaubnis von Michele Banko und Eric Brill reproduziert aus »Scaling to Very Very Large Corpora for Natural Language Disambiguation« *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics* (2001), 26–33.
- 10 Peter Norvig et al., »The Unreasonable Effectiveness of Data«, *IEEE Intelligent Systems* 24, no. 2 (2009): 8–12.
- 11 David Wolpert, »The Lack of A Priori Distinctions Between Learning Algorithms«, *Neural*

Computation 8 (1996): 1341–1390.

Kapitel 2 Ein Machine-Learning-Projekt von A bis Z

- 1 Dieses Beispielprojekt ist frei erfunden; das Ziel ist, die wichtigsten Schritte eines Machine-Learning-Projekts zu illustrieren, nicht etwas über den Immobilienmarkt zu erfahren.
- 2 Der ursprüngliche Datensatz erschien in: R. Kelley Pace and Ronald Barry, »Sparse Spatial Autoregressions«, *Statistics & Probability Letters* 33, no. 3 (1997): 291–297.
- 3 Eine in ein Machine-Learning-System eingegebene Information wird oft nach Claude Shannons Informationstheorie, die er an den Bell Labs entwickelt hat, um die Telekommunikation zu verbessern, als *Signal* bezeichnet: Ein hoher Quotient von Signal und Hintergrundrauschen ist wünschenswert.
- 4 Der Transponierungsoperator wandelt einen Spaltenvektor in einen Zeilenvektor um und umgekehrt.
- 5 Empfohlen ist die neueste Version von Python 3. Python 2.7+ kann auch funktionieren, aber da es mittlerweile überholt ist, beenden alle großen Wissenschaftsbibliotheken den Support dafür, und Sie sollten so schnell wie möglich zu Python 3 wechseln.
- 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 Wollen Sie pip nicht nur für sich, sondern für alle Benutzer auf Ihrem Rechner aktualisieren, sollten Sie die Option `--user` weglassen und sicherstellen, dass Sie Administratorrechte haben (beispielsweise durch das Präfix `sudo` vor dem Befehl unter Linux oder macOS).
- 8 Es gibt auch Alternativen wie `venv` (sehr ähnlich wie `virtualenv` und schon in der Standardbibliothek enthalten), `virtualenv-wrapper` (bietet zusätzliche Funktionalität über `virtualenv` hinaus), `pyenv` (erlaubt einen einfachen Wechsel zwischen Python-Versionen) und `pipenv` (ein großartiges Packaging-Tool von denselben Autoren wie die beliebte Bibliothek `requests`, das auf `pip` und `virtualenv` aufbaut).
- 9 Jupyter kann mit mehreren Python-Versionen und sogar vielen anderen Sprachen wie R oder Octave umgehen.
- 10 Sie müssten eventuell auch gesetzliche Vorgaben berücksichtigen, z.B. Felder mit persönlichen Daten, die niemals in ungeschützte Datenspeicher kopiert werden dürfen.
- 11 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.
- 12 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 .
- 13 Enthält in diesem Buch ein Codebeispiel einen Mix aus Code und Ausgaben – wie in diesem Fall –, ist es zur besseren Lesbarkeit wie im Python-Interpreter formatiert: Den Codezeilen ist `>>>` (oder `...` für eingerückte Blöcke) vorangestellt, die Ausgabezeilen haben keinen Präfix.
- 14 Sie werden häufig sehen, dass der Seed-Wert auf 42 gesetzt wird. Diese Zahl hat keine besondere Bedeutung, außer dass sie die Antwort auf die ultimative Frage nach dem Leben, dem Universum und dem ganzen Rest ist.
- 15 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.
- 16 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.

- 17 Details zu den Designprinzipien finden Sie in Lars Buitinck et al., »API Design for Machine Learning Software: Experiences from the Scikit-Learn Project«, arXiv preprint arXiv: 1309.0238 (2013).
- 18 Einige Prädiktoren besitzen außerdem Methoden, um die Konfidenz ihrer Vorhersagen zu bestimmen.
- 19 Diese Klasse steht in Scikit-Learn 0.20 und neuer zur Verfügung. Verwenden Sie eine ältere Version, sollten Sie über ein Upgrade nachdenken oder die pandas-Methode `Series.factorize()` nutzen.
- 20 Vor Scikit-Learn 0.20 konnte die Methode nur ganzzahlige kategorische Merkmalswerte codieren, aber seitdem kommt sie auch mit anderen Eingabetypen klar, unter anderem mit Textwerten.
- 21 Details finden Sie in der Dokumentation von SciPy.
- 22 So wie für Pipelines können die Namen beliebig sein, dürfen aber keine doppelten Unterstriche enthalten.
- 23 Eine REST- (oder RESTful-)API ist kurz gesagt eine HTTP-basierte API, die bestimmte Konventionen befolgt, beispielsweise den Einsatz der Standardverben von HTTP zum Lesen, Aktualisieren, Erstellen oder Löschen von Ressourcen (GET, POST, PUT und DELETE) oder das Verwenden von JSON für die Ein- und Ausgabe.
- 24 Ein Captcha ist ein Test, der sicherstellt, dass ein Anwender kein Roboter ist. Diese Tests wurden oft als billige Möglichkeit genutzt, um Trainingsdaten zu labeln.

Kapitel 3 Klassifikation

- 1 Scikit-Learn speichert heruntergeladene Daten standardmäßig im Verzeichnis `$HOME/scikit_learn_data`.
- 2 In einigen Fällen ist das Mischen keine gute Idee – beispielsweise wenn Sie mit Zeitreihen arbeiten (wie Aktienkursen oder Wetterdaten). Diese Fälle werden wir in den nächsten Kapiteln betrachten.
- 3 Bedenken Sie aber, dass unser Gehirn ein ausgezeichnetes System zur Mustererkennung ist und unser Gesichtssinn eine Menge komplexer Vorverarbeitung vornimmt, bevor Informationen in unser Bewusstsein vordringen. Deshalb bedeutet der Umstand, dass etwas leicht aussieht, nicht gleichzeitig auch, dass es leicht ist.
- 4 Scikit-Learn stellt einige weitere Optionen zur Mittelwertbildung und Metriken bei mehreren Labels zur Verfügung; schlagen Sie bitte die Details in der Dokumentation nach.
- 5 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.

Kapitel 4 Trainieren von Modellen

- 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.
- 2 Beachten Sie, dass Scikit-Learn den Bias-Term (`intercept_`) von den Gewichten getrennt ablegt (`coef_`).
- 3 Der Fachbegriff hierfür ist, ihre Ableitung ist *Lipschitz-stetig*.
- 4 Da Merkmal 1 kleiner ist, ist eine stärkere Änderung von θ_1 nötig, um sich auf die Kostenfunktion auszuwirken. Deshalb ist die Schüttel entlang der θ_1 -Achse länglich.
- 5 Eta (η) ist der 7. Buchstabe im griechischen Alphabet.
- 6 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.

- 7 Eine quadratische Gleichung entspricht der Form $y = ax^2 + bx + c$.
- 8 Diese Art Bias ist nicht mit dem Bias-Term linearer Modelle zu verwechseln.
- 9 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.
- 10 Normen werden in [Kapitel 2](#) besprochen.
- 11 Eine quadratische Matrix voller Nullen, die Einsen auf der Hauptdiagonalen enthält (von links oben nach rechts unten).
- 12 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« (<https://homl.info/12>) von Mark Schmidt et al. von der University of British Columbia.
- 13 Sie können sich einen Subgradientenvektor bei einem nicht differenzierbaren Punkt als mittleren Vektor zwischen den Gradientenvektoren um diesen Punkt herum vorstellen.
- 14 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.
- 15 NumPys Funktion `reshape()` erlaubt, dass eine Dimension -1 ist, was »unspecified« bedeutet: Der Wert wird aus der Länge des Arrays und den verbleibenden Dimensionen ermittelt.
- 16 Die Menge aller Punkte x , für die $\theta_0 + \theta_1x_1 + \theta_2x_2 = 0$ gilt, ergibt eine gerade Linie.

Kapitel 5 Support Vector Machines

- 1 Chih-Jen Lin et al., »A Dual Coordinate Descent Method for Large-Scale Linear SVM«, *Proceedings of the 25th International Conference on Machine Learning* (2008): 408–415.
- 2 John Platt, »Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines« (Microsoft Research Technical Report, 21. April 1998), <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-98-14.pdf>.
- 3 Wenn es n Merkmale gibt, ist die Entscheidungsfunktion allgemein eine n -dimensionale Hyperebene, und die Entscheidungsgrenze ist eine $(n - 1)$ -dimensionale Hyperebene.
- 4 Zeta (ζ) ist der 8. Buchstabe des griechischen Alphabets.
- 5 Um mehr über quadratische Programme zu erfahren, können Sie das Buch *Convex Optimization* (<https://homl.info/15>) von Stephen Boyd und Lieven Vandenberghe lesen, *Convex Optimization* (Cambridge University Press, 2004), oder sich eine Reihe Vorlesungsvideos (<https://homl.info/16>) von Richard Brown ansehen.
- 6 Die Zielfunktion ist konvex, und die Nebenbedingungen sind stetig differenzierbare und konvexe Funktionen.
- 7 Wie in [Kapitel 4](#) beschrieben, wird das Skalarprodukt zweier Vektoren \mathbf{a} und \mathbf{b} normalerweise als $\mathbf{a} \cdot \mathbf{b}$ notiert. Aber im Machine Learning werden Vektoren häufig als Spaltenvektoren repräsentiert (also Matrizen mit einer Spalte), daher lässt sich das Skalarprodukt auch durch Berechnen von $\mathbf{a}^\top \mathbf{b}$ erhalten. Um zum restlichen Buch konsistent zu sein, werden wir diese Notation hier verwenden und die Tatsache ignorieren, dass dies technisch gesehen zu einer Matrix mit einem Feld führt und nicht zu einem skalaren Wert.
- 8 Gert Cauwenberghs und Tomaso Poggio, »Incremental and Decremental Support Vector Machine

- Learning«, *Proceedings of the 13th International Conference on Neural Information Processing Systems* (2000): 388–394.
- 9 Antoine Bordes et al., »Fast Kernel Classifiers with Online and Active Learning«, *Journal of Machine Learning Research* 6 (2005): 1579–1619.

Kapitel 6 Entscheidungsbäume

- 1 Graphviz ist ein Open-Source-Softwarepaket zur Visualisierung von Graphen; es ist unter <http://www.graphviz.org/> verfügbar.
- 2 P ist die Menge der in polynomieller Zeit lösbarer 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 $\log_2(m) = \log(m) / \log(2)$.
- 4 Ein Reduzieren der Entropie wird oft als *Zugewinn an Information* bezeichnet.
- 5 Details finden Sie in einer interessanten Analyse (<https://homl.info/19>) von Sebastian Raschka.
- 6 Die bei jedem Knoten zu evaluierenden Merkmale werden zufällig ausgewählt.

Kapitel 7 Ensemble Learning und Random Forests

- 1 Leo Breiman, »Bagging Predictors«, *Machine Learning* 24, no. 2 (1996): 123–140.
- 2 In der Statistik nennt man das Bilden von Stichproben mit Zurücklegen auch *Bootstrapping*.
- 3 Leo Breiman, »Pasting Small Votes for Classification in Large Databases and On-Line«, *Machine Learning* 36, no. 1–2 (1999): 85–103.
- 4 Bias und Varianz wurden in [Kapitel 4](#) vorgestellt.
- 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.
- 6 Bei steigendem m nähert sich dieser Anteil $I - \exp(-1) \approx 63,212\%$.
- 7 Gilles Louppe and Pierre Geurts, »Ensembles on Random Patches«, *Lecture Notes in Computer Science* 7523 (2012): 346–361.
- 8 Tin Kam Ho, »The Random Subspace Method for Constructing Decision Forests«, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, no. 8 (1998): 832–844.
- 9 Tin Kam Ho, »Random Decision Forests«, *Proceedings of the Third International Conference on Document Analysis and Recognition* 1 (1995): 278.
- 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).
- 12 Pierre Geurts et al., »Extremely Randomized Trees«, *Machine Learning* 63, no. 1 (2006): 3–42.
- 13 Yoav Freund und Robert E. Schapire, »A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting«, *Journal of Computer and System Sciences* 55, no. 1 (1997): 119–139.
- 14 Dies geschieht nur zur Veranschaulichung. SVMs sind grundsätzlich keine guten Prädiktoren für Ada-

- Boost, weil sie langsam sind und bei Ada-Boost zur Instabilität neigen.
- 15 Der ursprüngliche AdaBoost-Algorithmus verwendet keine Lernrate als Hyperparameter.
 - 16 Details können Sie nachlesen in Ji Zhu et al., »Multi-Class AdaBoost«, *Statistics and Its Interface* 2, no. 3 (2009): 349–360.
 - 17 Gradient Boosting wurde erstmals im Artikel »Arcing the Edge« (<https://homl.info/arcing>) von Leo Breiman aus dem Jahr 1997 erwähnt und 1999 in »Greedy Function Approximation: A Gradient Boosting Machine« (<https://homl.info/gradboost>) von Jerome H. Friedman weiter ausgearbeitet.
 - 18 David H. Wolpert, »Stacked Generalization«, *Neural Networks* 5, no. 2 (1992): 241–259.
 - 19 Alternativ kann man auch Out-of-Fold-Vorhersagen nutzen. In einigen Situationen bezeichnet man dies als *Stacking*, während man den Hold-out-Datensatz *Blending* nennt. Die meisten Programmierer setzen diese Begriffe aber synonym ein.

Kapitel 8 Dimensionsreduktion

- 1 In Ordnung, vier Dimensionen, wenn Sie die Zeit dazuzählen, und einige mehr, wenn Sie sich mit der String-Theorie beschäftigen.
- 2 Einen in den 3-D-Raum projizierten rotierenden Tesserakt finden Sie auf <https://homl.info/30>. 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>.
- 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).
- 4 Karl Pearson, »On Lines and Planes of Closest Fit to Systems of Points in Space«, *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, no. 11 (1901): 559–572, <https://homl.info/pca>
- 5 Scikit-Learn verwendet den in David A. Ross et al., »Incremental Learning for Robust Visual Tracking«, *International Journal of Computer Vision* 77, no. 1–3 (2008): 125–141, beschriebenen Algorithmus.
- 6 Bernhard Schölkopf et al., »Kernel Principal Component Analysis«, in *Lecture Notes in Computer Science* 1327 (Berlin: Springer, 1997): 583–588.
- 7 Scikit-Learn verwendet den Kernel-Ridge-Regressionsalgorithmus, beschrieben in Gokhan H. Bakýr et al., »Learning to Find Pre-Images« (<https://homl.info/34>), *Proceedings of the 16th International Conference on Neural Information Processing Systems* (2004): 449–456.
- 8 Sam T. Roweis und Lawrence K. Saul, »Nonlinear Dimensionality Reduction by Locally Linear Embedding«, *Science* 290, no. 5500 (2000): 2323–2326.
- 9 Die geodätische Distanz zwischen zwei Knoten eines Graphen ist die Anzahl der Knoten auf dem kürzesten Pfad zwischen diesen Knoten.

Kapitel 9 Techniken des unüberwachten Lernens

- 1 Stuart P. Lloyd, »Least Squares Quantization in PCM«, *IEEE Transactions on Information Theory* 28, no. 2 (1982): 129–137.
- 2 Das liegt daran, dass der mittlere quadratische Abstand zwischen den Instanzen und ihrem nächstgelegenen Schwerpunkt mit jedem Schritt immer nur kleiner werden kann.
- 3 David Arthur und Sergei Vassilvitskii, »k-Means++: The Advantages of Careful Seeding«, *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms* (2007): 1027–1035.
- 4 Charles Elkan, »Using the Triangle Inequality to Accelerate k-Means«, *Proceedings of the 20th International Conference on Machine Learning* (2003): 147–153.

- 5 Die Dreiecksungleichung lautet $AC \leq AB + BC$, wobei A, B und C drei Punkte und AB, AC und BC die Abstände zwischen diesen Punkten sind.
- 6 David Sculley, »Web-Scale K-Means Clustering«, *Proceedings of the 19th International Conference on World Wide Web* (2010): 1177–1178.
- 7 Phi (ϕ oder φ) ist der 21. Buchstabe des griechischen Alphabets.
- 8 Der größte Teil ist Standardnotation, nur ergänzt durch den Wikipedia-Artikel zur Plate-Notation (https://en.wikipedia.org/wiki/Plate_notation).

Kapitel 10 Einführung in künstliche neuronale Netze mit Keras

- 1 Wenn Sie sich von der Biologie inspirieren lassen, ohne sich vor biologisch unrealistischen Modellen zu fürchten, erhalten Sie das Beste aus beiden Welten.
- 2 Warren S. McCulloch und Walter Pitts, »A Logical Calculus of the Ideas Immanent in Nervous Activity«, *The Bulletin of Mathematical Biology* 5, no. 4 (1943): 115–113.
- 3 Sie sind nicht direkt miteinander verbunden, sind sich aber so nahe, dass sie sehr schnell chemische Signale austauschen können.
- 4 Bild von Bruce Blaus (Creative Commons 3.0, <https://creativecommons.org/licenses/by/3.0/>). Reproduziert nach <https://en.wikipedia.org/wiki/Neuron>.
- 5 Im Zusammenhang mit Machine Learning sind mit dem Begriff »neuronales Netz« grundsätzlich ANNs und keine BNNs gemeint.
- 6 Drawing of a cortical lamination von S. Ramon y Cajal (Public Domain). Reproduziert nach https://en.wikipedia.org/wiki/Cerebral_cortex.
- 7 Der Name *Perceptron* wird manchmal für ein winziges Netz mit einer einzelnen TLU verwendet.
- 8 Diese Lösung ist nicht eindeutig: Wenn die Datenpunkte linear separierbar sind, gibt es eine unendliche Menge Hyperebenen, die diese separieren.
- 9 In den 1990er-Jahren wurde ein ANN mit mehr als zwei verborgenen Schichten als tief angesehen. Heutzutage sind ANNs mit Dutzenden oder gar Hunderten von Schichten nicht selten, daher ist die Definition von »tief« ein bisschen unscharf.
- 10 David Rumelhart et al. »Learning Internal Representations by Error Propagation«, (Defense Technical Information Center technical report, September 1985).
- 11 Diese Technik wurde von mehreren Wissenschaftlern in unterschiedlichen Fachgebieten unabhängig voneinander entdeckt, darunter P. Werbos im Jahr 1974.
- 12 Biologische Neuronen scheinen eine Aktivierungsfunktion zu nutzen, die wie die Sigmoid-Funktion mehr oder weniger s-förmig ist, daher haben die Forscher eine sehr lange Zeit selbst Sigmoid-Funktionen eingesetzt. Aber es zeigt sich, dass ReLU in ANNs im Allgemeinen besser funktioniert. Das ist einer der Fälle, in denen die biologische Analogie in die Irre geführt hat.
- 13 Projekt ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System).
- 14 Sie können `keras.utils.plot_model()` nutzen, um eine Abbildung Ihres Modells zu erzeugen.
- 15 Entsprüchen Ihre Trainings- oder Validierungsdaten nicht der erwarteten Form, werden Sie eine Exception erhalten. Das ist der vielleicht häufigste Fehler, daher sollten Sie sich mit der Fehlermeldung vertraut machen. Die Meldung ist sogar ziemlich klar: Versuchen Sie beispielsweise, dieses Modell mit einem Array mit flachgeklopften Bildern zu trainieren (`X_train.reshape(-1, 784)`), werden Sie die folgende Meldung erhalten: »`ValueError: Error when checking input: expected flatten_input to have 3 dimensions, but got array with shape (60000, 784)`«.
- 16 Heng-Tze Cheng et al., »Wide & Deep Learning for Recommender Systems«, *Proceedings of the First Workshop on Deep Learning for Recommender Systems* (2016): 7–10.
- 17 Der kurze Pfad kann auch genutzt werden, um das neuronale Netzwerk mit manuell entwickelten Features zu versorgen.

- 18 Der Name `input_` wird genutzt, um zu vermeiden, dass Pythons eingebaute Funktion `input()` überlagert wird.
- 19 Alternativ können Sie auch ein Dictionary übergeben, das die Eingabenamen auf die Eingabewerte abbildet, zum Beispiel `{"wide_input": X_train_A, "deep_input": X_train_B}`. Das ist besonders nützlich, wenn es viele Eingaben gibt, um eine falsche Reihenfolge zu vermeiden.
- 20 Alternativ können Sie ein Dictionary übergeben, das jeden Ausgabenamen auf die zugehörige Verlustfunktion abbildet. Wie bei der Eingabe ist das hilfreich, wenn es viele Ausgaben gibt, um eine falsche Reihenfolge zu verhindern. Die Verlustgewichte und -metriken (siehe weiter unten) können auch über Dictionaries gesetzt werden.
- 21 Modelle haben in Keras ein Attribut `output`, daher können wir diesen Namen nicht für die Hauptausgabeschicht verwenden und sind auf `main_output` ausgewichen.
- 22 Lisha Li et al., »Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization«, *Journal of Machine Learning Research* 18 (April 2018): 1–52.
- 23 Max Jaderberg et al., »Population Based Training of Neural Networks«, arXiv preprint arXiv:1711.09846 (2017).
- 24 Dominic Masters und Carlo Luschi, »Revisiting Small Batch Training for Deep Neural Networks«, arXiv preprint arXiv:1804.07612 (2018).
- 25 Elad Hoffer et al., »Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks«, *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 1729–1739.
- 26 Priya Goyal et al., »Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour«, arXiv preprint arXiv:1706.02677 (2017).
- 27 Leslie N. Smith, »A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 – Learning Rate, Batch Size, Momentum, and Weight Decay«, arXiv preprint arXiv:1803.09820 (2018).
- 28 Einige zusätzliche Architekturen von ANNs werden in [Anhang E](#) vorgestellt.

Kapitel 11 Trainieren von Deep-Learning-Netzen

- 1 Xavier Glorot und Yoshua Bengio, »Understanding the Difficulty of Training Deep Feedforward Neural Networks«, *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010): 249–256.
- 2 Hier besteht eine Analogie: Wenn Sie den Verstärkerknopf eines Mikrofons zu nah gen null drehen, 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 hineinkommt.
- 3 Zum Beispiel Kaiming He et al., »Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification«, *Proceedings of the 2015 IEEE International Conference on Computer Vision* (2015): 1026–1034.
- 4 Sofern es nicht Teil der ersten verborgenen Schicht ist, kann ein totes Neuron manchmal wieder zum Leben erweckt werden: Die Gradientenmethode kann durchaus Neuronen in tieferen Schichten so verändern, dass die gewichtete Summe der Eingaben des toten Neurons wieder positiv wird.
- 5 Bing Xu et al., »Empirical Evaluation of Rectified Activations in Convolutional Network«, arXiv preprint arXiv:1505.00853 (2015).
- 6 Djork-Arné Clevert et al., »Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)«, *Proceedings of the International Conference on Learning Representations* (2016).
- 7 Günter Klambauer et al., »Self-Normalizing Neural Networks«, *Proceedings of the 31st International*

- Conference on Neural Information Processing Systems* (2017): 972–981.
- 8 Sergey Ioffe und Christian Szegedy, »Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift«, *Proceedings of the 32nd International Conference on Machine Learning* (2015): 448–456.
 - 9 Sie werden aber während des Trainings basierend auf den Trainingsdaten geschätzt, daher sind sie durchaus trainierbar. In Keras heißt »nicht trainierbar« also eigentlich »nicht von der Backpropagation betroffen«.
 - 10 Die Keras-API spezifiziert auch eine Funktion `keras.backend.learning_phase()`, die während des Trainings 1 und ansonsten 0 zurückgeben sollte,
 - 11 Hongyi Zhang et al., »Fixup Initialization: Residual Learning Without Normalization«, arXiv preprint arXiv:1901.09321 (2019).
 - 12 Razvan Pascanu et al., »On the Difficulty of Training Recurrent Neural Networks«, *Proceedings of the 30th International Conference on Machine Learning* (2013): 1310–1318.
 - 13 Boris T. Polyak, »Some Methods of Speeding Up the Convergence of Iteration Methods«, *USSR Computational Mathematics and Mathematical Physics* 4, no. 5 (1964): 1–17.
 - 14 Yurii Nesterov, »A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$ «, *Doklady AN USSR* 269 (1983): 543–547.
 - 15 John Duchi et al., »Adaptive Subgradient Methods for Online Learning and Stochastic Optimization«, *Journal of Machine Learning Research* 12 (2011): 2121–2159.
 - 16 Dieser Algorithmus wurde von Geoffrey Hinton und Tijmen Tieleman im Jahr 2012 entwickelt und von Geoffrey Hinton in seinem Coursera-Kurs zu neuronalen Netzen präsentiert (Folien: <https://homl.info/57>, Video: <https://homl.info/58>). Da die Autoren keinen Fachartikel über das Verfahren verfasst haben, zitieren Forscher in ihren Artikeln häufig »Folie 29 in Vorlesung 6«.
 - 17 Diederik P. Kingma und Jimmy Ba, »Adam: A Method for Stochastic Optimization«, arXiv preprint arXiv:1412.6980 (2014).
 - 18 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.
 - 19 Timothy Dozat, »Incorporating Nesterov Momentum into Adam« (2016).
 - 20 Ashia C. Wilson et al., »The Marginal Value of Adaptive Gradient Methods in Machine Learning«, *Advances in Neural Information Processing Systems* 30 (2017): 4148–4158.
 - 21 Leslie N. Smith, »A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay«, arXiv preprint arXiv:1803.09820 (2018).
 - 22 Andrew Senior et al., »An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition«, *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2013): 6724–6728.
 - 23 Geoffrey E. Hinton et al., »Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors«, arXiv preprint arXiv:1207.0580 (2012).
 - 24 Nitish Srivastava et al., »Dropout: A Simple Way to Prevent Neural Networks from Overfitting«, *Journal of Machine Learning Research* 15 (2014): 1929–1958.
 - 25 Yarin Gal und Zoubin Ghahramani, »Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning«, *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1050–1059.
 - 26 Insbesondere zeigen die Autoren, dass das Trainieren eines Drop-out-Netzes mathematisch äquivalent zur Approximate Bayesian Inference in einem bestimmten Typ von Wahrscheinlichkeitsmodellen namens *Deep Gaussian Process* ist.
 - 27 Diese Klasse `MCDropout` arbeitet mit allen Keras-APIs zusammen, einschließlich der Sequential API.

Nutzen Sie nur die Functional API oder die Subclassing API, müssen Sie keine Klasse `MCDropout` erzeugen – erstellen Sie einfach eine normale Dropout-Schicht und rufen Sie diese mit `training=True` auf.

Kapitel 12 Eigene Modelle und Training mit TensorFlow

- 1 TensorFlow enthält noch eine weitere Deep-Learning-API namens *Estimators API*, aber das Team hinter TensorFlow empfiehlt stattdessen `tf.keras`.
- 2 Wenn es sein muss (aber das muss es vermutlich nie), können Sie Ihre eigenen Operationen mithilfe der C++-API schreiben.
- 3 Mehr über TPUs und ihre Funktionsweise erfahren Sie unter <https://homl.info/tpus>.
- 4 Eine erwähnenswerte Ausnahme ist `tf.math.log()`, das oft genutzt wird, aber keinen Alias `tf.log()` besitzt (weil man es mit Logging verwechseln könnte).
- 5 Es wäre keine gute Idee, ein gewichtetes Mittel zu verwenden. Denn dann würden zwei Instanzen mit dem gleichen Gewicht, aber in verschiedenen Batches, einen unterschiedlichen Einfluss auf das Training haben – abhängig vom Gesamtgewicht jedes Batchs.
- 6 Allerdings wird der Huber-Fehler nur selten als Metrik eingesetzt (MAE oder MSE werden vorgezogen).
- 7 Diese Klasse dient nur der Demonstration. Für eine einfachere und bessere Implementierung würde man eine Subklasse von `keras.metrics.Mean` erstellen – im Abschnitt »Streaming Metrics« des Notebooks finden Sie ein Beispiel.
- 8 Diese Funktion ist spezifisch für `tf.keras`. Stattdessen können Sie auch `keras.layers.Activation` verwenden.
- 9 Die Keras-API nennt dieses Argument `input_shape`, aber da auch die Batchdimensionen enthalten sind, nenne ich es lieber `batch_input_shape`. Das Gleiche gilt für `compute_output_shape()`.
- 10 Der Name »Subclassing API« bezieht sich im Allgemeinen nur auf das Erzeugen von eigenen Modellen per Subclassing, auch wenn sich viele andere Dinge per Unterklassen erstellen lassen, wie wir in diesem Kapitel gesehen haben.
- 11 Sie können `add_loss()` auch auf jeder Schicht innerhalb des Modells aufrufen, da es die Verluste rekursiv von allen Schichten einsammelt.
- 12 Verlässt die Aufzeichnung den Scope, zum Beispiel weil die Funktion, in der sie verwendet wurde, zurückkehrt, wird der Garbage Collector von Python sie für Sie löschen.
- 13 In Wirklichkeit verarbeiten wir gar nicht jede Instanz des Trainingsdatensatzes, da wir sie zufällig auswählen: Manche werden mehr als einmal verarbeitet, andere gar nicht. Und auch wenn der Trainingsdatensatz nicht ein Vielfaches der Batchgröße ist, werden wir ein paar Instanzen auslassen. In der Praxis ist das kein Problem.
- 14 Mit Ausnahme von Optimierern, da sehr wenige diese jemals anpassen – siehe den Abschnitt »Custom Optimizers« im Notebook für ein Beispiel.
- 15 In diesem trivialen Beispiel ist der Rechengraph allerdings so klein, dass es nichts zu optimieren gibt und `tf_cube()` tatsächlich sogar langsamer ist als `cube()`.

Kapitel 13 Daten mit TensorFlow laden und vorverarbeiten

- 1 Stellen Sie sich einen sortierten Satz Spielkarten links von Ihnen vor: Nun nehmen Sie die obersten drei Karten weg und mischen diese, dann wählen Sie zufällig eine davon aus und legen sie nach rechts, während Sie die beiden anderen in der Hand behalten. Nehmen Sie eine weitere Karte von der linken Seite, mischen Sie die drei Karten auf der Hand und wählen Sie wieder eine zufällig aus, um sie rechts abzulegen. Wenn Sie alle Karten auf diesem Weg nach rechts befördert haben – meinen Sie, dass diese

- jetzt perfekt durchmischt sind?
- 2 Im Allgemeinen reicht es aus, nur einen Batch Vorsprung zu haben, aber in manchen Fällen müssen Sie mehr Daten prefetchen. Alternativ können Sie TensorFlow automatisch die Entscheidung treffen lassen, indem Sie `tf.data.experimental.AUTOTUNE` übergeben (dies ist aktuell noch ein experimentelles Feature).
 - 3 Schauen Sie sich aber auch die Funktion `tf.data.experimental.prefetch_to_device()` an, die Daten direkt auf die GPU prefetchen kann.
 - 4 Die Unterstützung von Datasets ist spezifisch für `tf.keras` – sie steht in anderen Implementierungen der Keras-API nicht zur Verfügung.
 - 5 Die Methode `fit()` kümmert sich darum, die Datasets zu wiederholen. Alternativ können Sie auch `repeat()` für das Trainings-Dataset aufrufen, sodass es unendlich oft wiederholt wird, und beim Aufruf von `fit()` das Argument `steps_per_epoch` angeben. Das kann in seltenen Fällen nützlich sein, wenn Sie beispielsweise einen Durchmischungspuffer nutzen wollen, der über eine Epoche hinausgeht.
 - 6 Dieses Kapitel enthält wirklich nur die wichtigsten Informationen, die Sie über Protobufs wissen müssen, um TFRecords zu verwenden. Um mehr über Protobufs zu lernen, schauen Sie bei <https://homl.info/protobuf> vorbei.
 - 7 Warum wurde `Example` überhaupt definiert, da es doch nicht mehr als ein `Features`-Objekt enthält? Nun, die Entwickler von TensorFlow entscheiden sich vielleicht eines Tages, mehr Felder hinzuzufügen. Solange die neue `Example`-Definition weiterhin das Feld `features` mit der gleichen ID enthält, wird es abwärtskompatibel sein. Diese Erweiterbarkeit ist eines der tollen Features von Protobufs.
 - 8 Tomas Mikolov et al., »Distributed Representations of Words and Phrases and Their Compositionality«, *Proceedings of the 26th International Conference on Neural Information Processing Systems 2* (2013): 3111–3119.
 - 9 Malvina Nissim et al., »Fair Is Better Than Sensational: Man Is to Doctor as Woman Is to Doctor«, arXiv preprint arXiv:1905.09866 (2019).
 - 10 Bei großen Bildern können Sie stattdessen `tf.io.encode_jpeg()` nutzen. Damit sparen Sie viel Platz, verlieren aber ein bisschen Bildqualität.

Kapitel 14 Deep Computer Vision mit Convolutional Neural Networks

- 1 David H. Hubel, »Single Unit Activity in Striate Cortex of Unrestrained Cats«, *The Journal of Physiology* 147 (1959): 226–238.
- 2 David H. Hubel und Torsten N. Wiesel, »Receptive Fields of Single Neurons in the Cat’s Striate Cortex«, *The Journal of Physiology* 148 (1959): 574–591.
- 3 David H. Hubel und Torsten N. Wiesel, »Receptive Fields and Functional Architecture of Monkey Striate Cortex«, *The Journal of Physiology* 195 (1968): 215–243.
- 4 Kunihiko Fukushima, »Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position«, *Biological Cybernetics* 36 (1980): 193–202.
- 5 Yann LeCun et al., »Gradient-Based Learning Applied to Document Recognition« *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.
- 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 <https://homl.info/76> finden Sie Details).

- 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!
- 8 Im internationalen Einheitensystem (SI) gilt $1 \text{ MB} = 1.000 \text{ KB} = 1.000 \times 1.000 \text{ Bytes} = 1.000 \times 1.000 \times 8 \text{ Bits}$.
- 9 Andere bisher besprochene Kernels hatten Gewichte, aber Pooling-Kernels haben das nicht: Es handelt sich einfach um zustandslose gleitende Fenster.
- 10 Yann LeCun et al., »Gradient-Based Learning Applied to Document Recognition«, *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.
- 11 Alex Krizhevsky et al., »ImageNet Classification with Deep Convolutional Neural Networks«, *Proceedings of the 25th International Conference on Neural Information Processing Systems* 1 (2012): 1097–1105.
- 12 Matthew D. Zeiler und Rob Fergus, »Visualizing and Understanding Convolutional Networks«, *Proceedings of the European Conference on Computer Vision* (2014): 818–833.
- 13 Christian Szegedy et al., »Going Deeper with Convolutions«, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 1–9.
- 14 Im Film *Inception* aus dem Jahr 2010 reisen die Akteure durch mehrere Traumebenen immer tiefer. Daher stammt der Name dieser Module.
- 15 Karen Simonyan und Andrew Zisserman, »Very Deep Convolutional Networks for Large-Scale Image Recognition«, arXiv preprint arXiv:1409.1556 (2014).
- 16 Kaiming He et al., »Deep Residual Learning for Image Recognition«, arXiv preprint arXiv:1512:03385 (2015).
- 17 Beim Beschreiben eines neuronalen Netzes ist es üblich, nur die Schichten mit Parametern zu zählen.
- 18 Christian Szegedy et al., »Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning«, arXiv preprint arXiv:1602.07261 (2016).
- 19 François Chollet, »Xception: Deep Learning with Depthwise Separable Convolutions«, arXiv preprint arXiv:1610.02357 (2016).
- 20 Der Name ist eventuell mehrdeutig, weil räumlich trennbare Convolutions auch oft als »Separable Convolutions« bezeichnet werden.
- 21 Xingyu Zeng et al., »Crafting GBD-Net for Object Detection«, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, no. 9 (2018): 2109–2123.
- 22 Jie Hu et al., »Squeeze-and-Excitation Networks«, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 7132–7141.
- 23 Im ImageNet-Datensatz ist jedes Bild mit einem Wort im WordNet-Datensatz (<https://wordnet.princeton.edu/>) verbunden: Die Kategorien-ID ist einfach eine WordNet-ID.
- 24 Adriana Kovashka et al., »Crowdsourcing in Computer Vision«, *Foundations and Trends in Computer Graphics and Vision* 10, no. 3 (2014): 177–243.
- 25 Jonathan Long et al., »Fully Convolutional Networks for Semantic Segmentation«, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 3431–3440.
- 26 Es gibt eine kleine Ausnahme: Ein Convolutional Layer mit "valid"-Padding wird sich beschweren, wenn die Eingabegröße kleiner als die Kernelgröße ist.
- 27 Dabei wird davon ausgegangen, dass wir im Netz nur "same"-Padding verwenden: Tatsächlich würde ein "valid"-Padding die Größe der Feature Map verringern. Zudem kann 448 mehrfach durch 2 dividiert werden, bis wir ohne Rundungsfehler 7 erreichen. Nutzt eine Schicht eine andere Schrittweite als 1 oder 2, kann es zu Rundungsfehlern kommen, und die Feature Maps werden eventuell am Ende kleiner.
- 28 Joseph Redmon et al., »You Only Look Once: Unified, Real-Time Object Detection«, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016): 779–788.

- 29 Joseph Redmon und Ali Farhadi, »YOLO9000: Better, Faster, Stronger«, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017): 6517–6525.
- 30 Joseph Redmon und Ali Farhadi, »YOLOv3: An Incremental Improvement«, arXiv preprint arXiv: 1804.02767 (2018).
- 31 Wei Liu et al., »SSD: Single Shot Multibox Detector«, *Proceedings of the 14th European Conference on Computer Vision* 1 (2016): 21–37.
- 32 Shaoqing Ren et al., »Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks«, *Proceedings of the 28th International Conference on Neural Information Processing Systems* 1 (2015): 91–99.
- 33 Diese Art von Schicht wird manchmal auch als *Deconvolution Layer* bezeichnet, aber er führt *nicht* das durch, was Mathematiker als Dekonvolution (oder Entfaltung) bezeichnen, daher sollte dieser Name vermieden werden.
- 34 Kaiming He et al., »Mask R-CNN«, arXiv preprint arXiv:1703.06870 (2017).
- 35 Geoffrey Hinton et al., »Matrix Capsules with EM Routing«, *Proceedings of the International Conference on Learning Representations* (2018).

Kapitel 15 Verarbeiten von Sequenzen mit RNNs und CNNs

- 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://homl.info/91>) aus dem Jahr 2013 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://homl.info/92>) aus dem Jahr 2015 von Quoc V. Le et al. gezeigt.
- 2 Beachten Sie, dass eine `TimeDistributed(Dense(n))`-Schicht einer `Conv1D(n, kernel_size=1)`-Schicht entspricht.
- 3 César Laurent et al., »Batch Normalized Recurrent Neural Networks«, *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2016): 2657–2661.
- 4 Jimmy Lei Ba et al., »Layer Normalization«, arXiv preprint arXiv:1607.06450 (2016).
- 5 Es wäre einfacher gewesen, stattdessen von `SimpleRNNCell` abzuleiten, sodass wir keine interne `Simple RNNCell` erstellen oder die Attribute `state_size` und `output_size` behandeln müssten, aber unser Ziel war hier, zu zeigen, wie Sie eine eigene Zelle von Grund auf erstellen.
- 6 Eine Figur aus den Animationsfilmen *Findet Nemo* und *Findet Dory*, die mit einem Verlust des Kurzzeitgedächtnisses zu kämpfen hat.
- 7 Sepp Hochreiter und Jürgen Schmidhuber, »Long Short-Term Memory«, *Neural Computation* 9, no. 8 (1997): 1735–1780.
- 8 Haşim Sak et al., »Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition«, arXiv preprint arXiv:1402.1128 (2014).
- 9 Wojciech Zaremba et al., »Recurrent Neural Network Regularization«, arXiv preprint arXiv:1409.2329 (2014).
- 10 F. A. Gers und J. Schmidhuber, »Recurrent Nets That Time and Count«, *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks* (2000): 189–194.
- 11 Kyunghyun Cho et al., »Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation«, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (2014): 1724–1734.
- 12 Ein Artikel von Klaus Greff et al. aus dem Jahr 2015, »LSTM: A Search Space Odyssey« (<https://homl.info/98>), deutet darauf hin, dass sämtliche LSTM-Varianten etwa gleich gut abschneiden.
- 13 Aaron van den Oord et al., »WaveNet: A Generative Model for Raw Audio«, arXiv preprint arXiv:

1609.03499 (2016).

- 14 Das vollständige WaveNet nutzt ein paar Tricks mehr, wie zum Beispiel Skip-Verbindungen wie in einem ResNet und *Gated Activation Units* ähnlich denen in einer GRU-Zelle. Im Notebook finden Sie mehr Details.

Kapitel 16 Natürliche Sprachverarbeitung mit RNNs und Attention

- 1 Alan Turing, »Computing Machinery and Intelligence«, *Mind* 49 (1950): 433–460.
- 2 Natürlich entstand das Wort *Chatbot* erst viel später. Turing nannte seinen Test das *Imitationsspiel*: Maschine A und Mensch B chatten mit dem menschlichen Prüfer C via Textnachrichten – der Prüfer stellt Fragen, um herauszufinden, welcher von beiden (A oder B) die Maschine ist. Die Maschine besteht den Test, wenn sie den Prüfer täuschen kann, während der Mensch B versuchen muss, dem Prüfer zu helfen.
- 3 Per definitionem ändern sich Mittelwert, Varianz und *Autokorrelationen* (also die Korrelationen zwischen Werten in der Zeitserie, die durch ein gegebenes Intervall getrennt sind) einer stationären Zeitserie nicht mit der Zeit. Das ist ziemlich einschränkend – so werden beispielsweise Zeitserien ausgeschlossen, die Trends oder zyklische Muster besitzen. RNNs sind da toleranter – sie können Trends und zyklische Muster erlernen.
- 4 Taku Kudo, »Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates«, arXiv preprint arXiv:1804.10959 (2018).
- 5 Taku Kudo und John Richardson, »SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing«, arXiv preprint arXiv:1808.06226 (2018).
- 6 Rico Sennrich et al., »Neural Machine Translation of Rare Words with Subword Units«, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* 1 (2016): 1715–1725.
- 7 Yonghui Wu et al., »Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation«, arXiv preprint arXiv:1609.08144 (2016).
- 8 Ihre ID ist nur deshalb 0, weil es sich um das häufigste »Wort« im Datensatz handelt. Es wäre vermutlich eine gute Idee, sicherzustellen, dass die Padding-Token immer als 0 codiert werden, auch wenn sie nicht am häufigsten vorkommen.
- 9 Genauer gesagt, entspricht das Sentence Embedding dem mittleren Word Embedding multipliziert mit der Wurzel der Anzahl an Wörtern im Satz. Das kompensiert die Tatsache, dass der Mittelwert von n Vektoren kleiner wird, wenn n wächst.
- 10 Ilya Sutskever et al., »Sequence to Sequence Learning with Neural Networks«, arXiv preprint arXiv: 1409.3215 (2014).
- 11 Sébastien Jean et al., »On Using Very Large Target Vocabulary for Neural Machine Translation«, *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing* 1 (2015): 1–10.
- 12 Samy Bengio et al., »Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks«, arXiv preprint arXiv:1506.03099 (2015).
- 13 Dzmitry Bahdanau et al., »Neural Machine Translation by Jointly Learning to Align and Translate«, arXiv preprint arXiv:1409.0473 (2014).
- 14 Die in NMT am häufigsten genutzte Metrik ist der BiLingual Evaluation Understudy (BLEU) Score, der jede vom Modell produzierte Übersetzung mit einer Reihe von Menschen erstellten guten Übersetzungen vergleicht: Er zählt die Anzahl an n -Grammen (Folgen von n Wörtern), die in jeder der Zielaussagen auftauchen, und passt den Score so an, dass er die Häufigkeit der produzierten n -Gramme in den Zielaussagen berücksichtigt.
- 15 Denken Sie daran, dass eine zeitverteilte Dense-Schicht einer normalen Dense-Schicht entspricht, die

Sie unabhängig bei jedem Zeitschritt anwenden (nur viel schneller).

- 16 Minh-Thang Luong et al., »Effective Approaches to Attention-Based Neural Machine Translation«, *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015): 1412–1421.
- 17 Kelvin Xu et al., »Show, Attend and Tell: Neural Image Caption Generation with Visual Attention«, *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2048–2057.
- 18 Dies ist ein Teil aus Abbildung 3 des Artikels. Es wurde mit freundlicher Genehmigung der Autoren übernommen.
- 19 Marco Tulio Ribeiro et al., »Why Should I Trust You?: Explaining the Predictions of Any Classifier«, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144.
- 20 Ashish Vaswani et al., »Attention Is All You Need«, *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 6000–6010.
- 21 Da der Transformer zeitverteilte Dense-Schichten einsetzt, könnten Sie argumentieren, dass er 1-D-Convolutional-Schichten mit einer Kernelgröße von 1 verwendet.
- 22 Dies ist Abbildung 1 aus dem Artikel, die mit freundlicher Genehmigung durch die Autoren wiedergegeben wird.
- 23 Dies ist der rechte Teil von Abbildung 2 aus dem Artikel, der mit freundlicher Genehmigung durch die Autoren wiedergegeben wird.
- 24 Matthew Peters et al., »Deep Contextualized Word Representations«, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2018): 2227–2237.
- 25 Jeremy Howard and Sebastian Ruder, »Universal Language Model Fine-Tuning for Text Classification«, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics* 1 (2018): 328–339.
- 26 Alec Radford et al., »Improving Language Understanding by Generative Pre-Training« (2018).
- 27 So folgt aus dem Satz »Jane had a lot of fun at her friend's birthday party« der Satz »Jane enjoyed the party«, er steht aber im Gegensatz zu »Everyone hated the party« und hat nichts zu tun mit »The Earth is flat«.
- 28 Alec Radford et al., »Language Models Are Unsupervised Multitask Learners« (2019).
- 29 Jacob Devlin et al., »BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding«, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2019).
- 30 Maha Elbayad et al., »Pervasive Attention: 2D Convolutional Neural Networks for Sequence-to-Sequence Prediction«, arXiv preprint arXiv:1808.03867 (2018).
- 31 Shuai Li et al., »Independently Recurrent Neural Network (IndRNN): Building a Longer and Deeper RNN«, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 5457–5466.

Kapitel 17 Representation Learning und Generative Learning mit Autoencodern und GANs

- 1 William G. Chase und Herbert A. Simon, »Perception in Chess«, *Cognitive Psychology* 4, no. 1 (1973): 55–81.
- 2 Vielleicht möchten Sie die Genauigkeit als Metrik nutzen, aber das würde nicht gut funktionieren, da diese Metrik davon ausgeht, dass die Labels für jedes Pixel entweder 1 oder 0 sind. Sie können dieses Problem umgehen, indem Sie eine eigene Metrik erstellen, die die Genauigkeit berechnet, nachdem die

Ziele und Vorhersagen auf 0 oder 1 gerundet wurden.

- 3 Joshua Bengio et al., »Greedy Layer-Wise Training of Deep Networks«, *Proceedings of the 19th International Conference on Neural Information Processing Systems* (2006): 153–160.
- 4 Jonathan Masci et al., »Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction«, *Proceedings of the 21st International Conference on Artificial Neural Networks* 1 (2011): 52–59.
- 5 Pascal Vincent et al., »Extracting and Composing Robust Features with Denoising Autoencoders«, *Proceedings of the 25th International Conference on Machine Learning* (2008): 1096–1103.
- 6 Pascal Vincent et al., »Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion«, *Journal of Machine Learning Research* 11 (2010): 3371–3408.
- 7 Diederik Kingma und Max Welling, »Auto-Encoding Variational Bayes«, arXiv preprint arXiv:1312.6114 (2013).
- 8 Variational Autoencoder sind noch allgemeiner; die Codings sind nicht auf Normalverteilungen beschränkt.
- 9 Die mathematischen Details finden Sie im ursprünglichen Artikel zu Variational Autoencodern und im großartigen Tutorial (<https://homl.info/116>) von Carl Doersch (2016).
- 10 Ian Goodfellow et al., »Generative Adversarial Nets«, *Proceedings of the 27th International Conference on Neural Information Processing Systems* 2 (2014): 2672–2680.
- 11 Einen guten Vergleich der wichtigsten GAN-Verlustfunktionen finden Sie in diesem großartigen GitHub-Projekt (<https://homl.info/ganloss>) von Hwalsuk Lee.
- 12 Mario Lucic et al., »Are GANs Created Equal? A Large-Scale Study«, *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (2018): 698–707.
- 13 Alec Radford et al., »Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks«, arXiv preprint arXiv:1511.06434 (2015).
- 14 Wiedergabe mit freundlicher Genehmigung durch die Autoren.
- 15 Mehdi Mirza und Simon Osindero, »Conditional Generative Adversarial Nets«, arXiv preprint arXiv: 1411.1784 (2014).
- 16 Tero Karras et al., »Progressive Growing of GANs for Improved Quality, Stability, and Variation«, *Proceedings of the International Conference on Learning Representations* (2018).
- 17 Der Dynamikumfang einer Variablen ist das Verhältnis zwischen dem größten und kleinsten Wert, den sie annehmen kann.
- 18 Tero Karras et al., »A Style-Based Generator Architecture for Generative Adversarial Networks«, arXiv preprint arXiv:1812.04948 (2018).
- 19 Wiedergabe mit freundlicher Genehmigung durch die Autoren.

Kapitel 18 Reinforcement Learning

- 1 Mehr Details finden Sie im RL-Buch (<https://homl.info/126>) von Richard Sutton und Andrew Barto, *Reinforcement Learning: An Introduction* (MIT Press).
- 2 Volodymyr Mnih et al., »Playing Atari with Deep Reinforcement Learning«, arXiv preprint arXiv: 1312.5602 (2013).
- 3 Volodymyr Mnih et al., »Human-Level Control Through Deep Reinforcement Learning«, *Nature* 518 (2015): 529–533.
- 4 Schauen Sie sich die Videos auf <https://homl.info/dqn3> an, in denen das DeepMind-System lernt, *Space Invaders*, *Breakout* und weitere Spiele zu spielen.
- 5 Bild (a) stammt von der NASA (Public Domain), (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). Die

Bilder (c) und (d) stammen von Wikipedia. (c) wurde vom User Stevertigo erstellt und unter Creative Commons BY-SA 2.0 (<https://creativecommons.org/licenses/by-sa/2.0/>) veröffentlicht, (d) ist Public Domain. (e) wurde von Pixabay übernommen und ist veröffentlicht unter Creative Commons CC0 (<https://creativecommons.org/publicdomain/zero/1.0/>).

- 6 Es zahlt sich häufig aus, den Leistungsschwächeren eine geringe Überlebenschance zu geben, um ein wenig 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.
- 8 Ein interessantes Beispiel eines genetischen Algorithmus zum Reinforcement Learning ist der NEAT-Algorithmus (*NeuroEvolution of Augmenting Topologies*, <https://homl.info/neat>).
- 9 Das nennt sich *Gradientenaufstieg* – genau wie Gradientenabstieg, aber in die entgegengesetzte Richtung: Maximieren statt Minimieren.
- 10 OpenAI ist ein Forschungsunternehmen im Bereich künstlicher Intelligenz, das zum Teil von Elon Musk gegründet wurde. Sein erklärtes Ziel ist, freundlich gesinnte AIs zu ermöglichen und zu entwickeln, die der Menschheit nützen (anstatt sie auszulöschen).
- 11 Ronald J. Williams, »Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning«, *Machine Learning* 8 (1992) : 229–256.
- 12 Richard Bellman, »A Markovian Decision Process«, *Journal of Mathematics and Mechanics* 6, no. 5 (1957): 679–684.
- 13 Ein Post (<https://homl.info/rllhard>) von Alex Irpan beschreibt die größten Schwierigkeiten und Einschränkungen von RL sehr gut.
- 14 Hado van Hasselt et al., »Deep Reinforcement Learning with Double Q-Learning«, *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (2015): 2094–2100.
- 15 Tom Schaul et al., »Prioritized Experience Replay«, arXiv preprint arXiv:1511.05952 (2015).
- 16 Es könnte auch sein, dass die Belohnungen einfach verrauscht sind – dann gibt es bessere Methoden für das Schätzen der Wichtigkeit einer Erfahrung (im Artikel finden Sie ein paar Beispiele).
- 17 Ziyu Wang et al., »Dueling Network Architectures for Deep Reinforcement Learning«, arXiv preprint arXiv:1511.06581 (2015).
- 18 Matteo Hessel et al., »Rainbow: Combining Improvements in Deep Reinforcement Learning«, arXiv preprint arXiv:1710.02298 (2017): 3215–3222.
- 19 Da es nur drei Primärfarben gibt, können Sie nicht einfach ein Bild mit vier Farbkanälen anzeigen. Daher habe ich den letzten Kanal mit den ersten drei Kanälen kombiniert, um das hier dargestellte RGB-Bild zu erhalten. Lila ist in Wirklichkeit eine Mischung aus Blau und Rot, aber der Agent sieht vier unabhängige Kanäle.
- 20 Aktuell gibt es noch keinen priorisierten Experience Replay Buffer, aber vermutlich wird bald einer als Open Source veröffentlicht werden.
- 21 Einen Vergleich der Leistung dieses Algorithmus bei verschiedenen Atari-Spielen finden Sie in Abbildung 3 des DeepMind-Artikels (<https://homl.info/dqn2>) aus dem Jahr 2015.
- 22 Volodymyr Mnih et al., »Asynchronous Methods for Deep Reinforcement Learning«, *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1928–1937.
- 23 Tuomas Haarnoja et al., »Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor«, *Proceedings of the 35th International Conference on Machine Learning* (2018): 1856–1865.
- 24 John Schulman et al., »Proximal Policy Optimization Algorithms«, arXiv preprint arXiv:1707.06347 (2017).
- 25 John Schulman et al., »Trust Region Policy Optimization«, *Proceedings of the 32nd International Conference on Machine Learning* (2015): 1889–1897.

- 26 Deepak Pathak et al., »Curiosity-Driven Exploration by Self-Supervised Prediction, *Proceedings of the 34th International Conference on Machine Learning* (2017): 2778–2787.

Kapitel 19 TensorFlow-Modelle skalierbar trainieren und deployen

- 1 Ein A/B-Experiment besteht aus dem Testen zweier verschiedener Versionen Ihres Produkts mit unterschiedlichen Untermengen von Anwendern, um herauszufinden, welche Version am besten funktioniert, und um andere Einsichten zu gewinnen.
- 2 Eine REST- (oder RESTful-)API ist eine API, die die Standardverben von HTTP nutzt, wie zum Beispiel GET, POST, PUT und DELETE, und JSON-Ein- und -Ausgaben verwendet. Das gRPC-Protokoll ist komplexer, aber effizienter. Daten werden über Protocol Buffer ausgetauscht (siehe [Kapitel 13](#)).
- 3 Falls Sie mit Docker nicht vertraut sind: Es ermöglicht Ihnen, einen Satz von Anwendungen herunterzuladen, die in einem *Docker-Image* verpackt sind (einschließlich all ihrer Abhängigkeiten und normalerweise mit einer guten Standardkonfiguration), und es auf Ihrem System über eine *Docker-Engine* auszuführen. Lassen Sie ein Image laufen, erzeugt dieses einen *Docker-Container*, der die Anwendung gut isoliert von Ihrem eigenen System hält (Sie können ihm aber begrenzten Zugriff gewähren, wenn Sie das wollen). Es entspricht einer virtuellen Maschine, ist aber viel schneller und schlanker, da der Container direkt auf dem Kernel des Hosts aufsetzt. Das Image muss also nicht seinen eigenen Kernel enthalten oder laufen lassen.
- 4 Fairerweise sei gesagt, dass dies abgemildert werden kann, indem man zuerst die Daten serialisiert und sie dann mit Base64 codiert, bevor man den REST-Request stellt. Und REST-Requests können mit gzip komprimiert werden, was die Payload-Größe deutlich verringert.
- 5 Enthält das SavedModel Beispielinstanzen im Verzeichnis *assets/extras*, können Sie TF Serving so konfigurieren, dass es das Modell mit diesen Instanzen ausführt, bevor es neue Requests bedient. Das wird als *Model Warmup* bezeichnet: Es stellt sicher, dass alles sauber geladen ist, und vermeidet lange Response-Zeiten für die ersten Requests.
- 6 Aktuell steht TensorFlow 2 noch nicht auf AI Platform zur Verfügung, aber das ist in Ordnung – Sie können 1.13 verwenden, damit laufen Ihre TF 2 Saved Models problemlos.
- 7 Erhalten Sie einen Fehler, der besagt, dass das Modul `google.appengine` nicht gefunden wurde, setzen Sie im Aufruf der Methode `build()` `cache_discovery=False` – siehe <https://stackoverflow.com/q/55561354>.
- 8 Schauen Sie sich auch die Graph Transform Tools (<https://homl.info/tfgtt>) von TensorFlow an, mit denen Sie Rechengraphen modifizieren und optimieren können.
- 9 Wenn Sie an diesem Thema interessiert sind, schauen Sie sich *Federated Learning* (<https://tensorflow.org/federated>) an.
- 10 Bitte schauen Sie in der Dokumentation nach detaillierten und aktuellen Installationsanweisungen, da sich diese recht häufig ändern.
- 11 Viele Codebeispiele in diesem Kapitel verwenden experimentelle APIs. Sie werden in zukünftigen Versionen sehr wahrscheinlich in die Code-API wandern. Schlägt also bei Ihnen eine experimentelle Funktion fehl, versuchen Sie, einfach das Wort `experimental` zu entfernen – vielleicht funktioniert es dann schon. Wenn nicht, hat sich eventuell die API ein bisschen geändert – schauen Sie dann bitte im Jupyter-Notebook nach, da ich dort immer den korrekten Code nutzen werde.
- 12 Diese Kontingente sollen vermutlich böse Buben davon abhalten, GCP mit gestohlenen Kreditkartendaten zum Schürfen von Kryptowährung einzusetzen.
- 13 Martín Abadi et al., »TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems«, Google Research whitepaper (2015).

- 14 Wie wir in [Kapitel 12](#) gesehen haben, ist ein Kernel die Implementierung einer Variablen oder Operation für einen bestimmten Daten- und Device-Typ. So gibt es zum Beispiel einen GPU-Kernel für die `float32-tf.matmul()`-Operation, aber nicht für `int32-tf.matmul()` (dort ist nur ein CPU-Kernel vorhanden).
- 15 Sie können auch `tf.debugging.set_log_device_placement(True)` verwenden, um alle Device-Platzierungen loggen zu lassen.
- 16 Das kann hilfreich sein, wenn Sie eine perfekte Reproduzierbarkeit garantieren wollen, wie ich in diesem Video (<https://homl.info/repro>) erklärt habe (das auf TF 1 basiert).
- 17 Aktuell werden die Daten nur in das CPU-RAM vorgeladen, aber Sie können `tf.data.experimental.prefetch_to_device()` einsetzen, um die Daten zu laden und auf das Device Ihrer Wahl schicken zu lassen, sodass die GPU keine Zeit damit verschwendet, auf die Übermittlung der Daten warten zu müssen.
- 18 Wollen Sie mehr über parallelisierte Modelle erfahren, schauen Sie sich Mesh TensorFlow (<https://github.com/tensorflow/mesh>) an.
- 19 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).
- 20 Jianmin Chen et al., »Revisiting Distributed Synchronous SGD«, arXiv preprint arXiv:1604.00981 (2016).
- 21 Mehr Informationen zu den AllReduce-Algorithmen finden Sie in einem ausgezeichneten Post (<https://homl.info/uenopost>) von Yuichiro Ueno und auf dieser Seite (<https://homl.info/ncclalgo>) über das Skalieren mit NCCL.
- 22 Aktuell ist die 2.0-Runtime noch nicht verfügbar, aber sie sollte bereitstehen, wenn Sie diese Zeilen lesen. Schauen Sie sich die Liste der verfügbaren Runtimes (<https://homl.info/runtimes>) an.
- 23 Daniel Golovin et al., »Google Vizier: A Service for Black-Box Optimization«, *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017): 1487–1495.

Anhang A: Lösungen zu den Übungsaufgaben

- 1 Wenn Sie zwei beliebige Punkte der Kurve über eine gerade Linie verbinden, schneidet diese niemals die Kurve.
- 2 \log_2 ist der binäre Logarithmus $\log_2(m) = \log(m) / \log(2)$.
- 3 Wenn sich die vorherzusagenden Werte um viele Größenordnungen unterscheiden, sollten Sie statt der Zielgröße den Logarithmus der Zielgröße vorhersagen. Durch Berechnen der Exponentialfunktion aus der Ausgabe des Netzes erhalten Sie den geschätzten Wert (da $\exp(\log v) = v$).
- 4 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), den Schwellenwert beim Gradient Clipping, die Art des Optimierungsverfahrens und dessen Hyperparameter (z.B. den 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.

Anhang B: Checkliste für Machine-Learning-Projekte

- 1 Jasper Snoek et al., »Practical Bayesian Optimization of Machine Learning Algorithms«, *Proceedings of the 25th International Conference on Neural Information Processing Systems* 2 (2012): 2951–2959.

Anhang E: Weitere verbreitete Architekturen neuronaler Netze

- 1 Miguel Á. Carreira-Perpiñán und Geoffrey E. Hinton, »On Contrastive Divergence Learning«, *Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics* (2005): 59–66.
- 2 Geoffrey E. Hinton et al., »A Fast Learning Algorithm for Deep Belief Nets«, *Neural Computation* 18 (2006): 1527–1554.
- 3 Details und eine Demonstration finden Sie in diesem Video von Geoffrey Hinton: <https://homl.info/137>.
- 4 Stellen Sie sich eine Klasse Kinder mit ähnlichen Fähigkeiten vor. Ein Kind ist vielleicht ein wenig besser im Basketball als die anderen. Das 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 sich die anderen Kinder auf andere Gebiete spezialisieren. Nach einer Weile besteht die gesamte Klasse aus kleinen Gruppen von Spezialisten.

Anhang F: Spezielle Datenstrukturen

- 1 Wenn Sie mit Unicode-Codepoints nicht vertraut sind, schauen Sie sich <https://homl.info/unicode> an.

Anhang G: TensorFlow-Graphen

- 1 Sie können sie getrost ignorieren – es gibt sie hier nur aus technischen Gründen, um sicherzustellen, dass TF Functions keine internen Strukturen nach außen geben.
- 2 Ein verbreitetes Binärformat, das in [Kapitel 13](#) behandelt wurde.

Index

Symbole

χ^2 -Test *siehe* Chi-Quadrat-Test
 ℓ_∞ -Norm 43
 ℓ_0 -Norm 43
 ℓ_1 - und ℓ_2 -Regularisierung 367
 ℓ_1 -Norm 43, 139, 149, 362, 367
 ℓ_2 -Norm 43, 138, 139, 149, 152, 367
 ℓ_k -Norm 43
1cycle Scheduling 364

A

A/B-Experimente 667
Abstimmverfahren unter Klassifikatoren 192
Achsenabschnitt 116
Actor-Critic-Algorithmus 626, 664
AdaBoost 202
AdaGrad 357
Adam-Optimierung 359, 361
Adaptive Instance Normalization (AdaIN) 608
adaptive Lernrate 358
adaptive Optimierung von Momenten 359
Additive Attention 555
Advantage Actor-Critic (A2C) 664
Adversarial Learning 499
Adversarial Training 572
affine Transformationen 607
Affinität 239
Affinity Propagation 261
Agenten 612
agglomeratives Clustering 260
Ähnlichkeitsfunktion 161
Ähnlichkeitsmaß 19
AI Platform 680
Akaike-Informationskriterium (AIC) 269
Aktionen, evaluieren 621
Aktionsschritt 658
aktives Lernen 257
Aktivierungsfunktionen 286, 293
logistisch (sigmoid) 304

nicht sättigende 337
ReLU-Funktion 293
Scaled Exponential Linear Unit (SELU) 336, 339, 371
Softmax 295, 491
Softplus 294
Tangens hyperbolicus (\tanh) 293
AlexNet, Architektur 467
Algorithmen
Actor-Critic-Algorithmus 626, 664
Advantage Actor-Critic (A2C) 664
AllReduce 705
Anomalieerkennung 276
Asynchronous Advantage Actor-Critic (A3C) 664
BIRCH-Algorithmus 261
Daten vorbereiten für 64
Dueling-DQN-Algorithmus 643
Expectation-Maximization-(EM-)Algorithmus 264
genetische 614
Isolation Forest 276
K-Means 240
Lloyd-Forgy 241
Mean-Shift-Algorithmus 261
Off-Policy-Algorithmus 634
One-Class-SVM 276
On-Policy-Algorithmus 634
Proximal Policy Optimization (PPO) 665
randomisierte PCA 227
Soft Actor-Critic-Algorithmus 664
unüberwachtes Lernen 11
Vorbereiten von Daten für 74
zur Visualisierung 12
AllReduce-Algorithmus 705
AlphaGo 16, 611
Alpha-Kanäle 252
Anaconda 45
Anchor Boxes 493
Annahmen überprüfen 44
Anomalieerkennung 13
gaußsche Mischverteilung 267
per Clustering 240
weitere Algorithmen 276
Ziel 238
Anteil erklärter Varianz 224
A-posteriori-Verteilung 273
Apples Siri 281
A-priori-Verteilung 273
Äquidistanz 462
Arbeitsverzeichnis 44
Area under the Curve (AUC) 101
array_split() 227
Assoziationsregeln, Lernen mit 14

assoziative Speicher 777
asynchrone Updates 707
Asynchronous Advantage Actor-Critic (A3C) 664
Atari-Vorverarbeitung 647
Attention-Mechanismen
 Definition 530
 Explainability 557
 Transformer-Architektur 558
 Übersicht 553
 visuelle Attention 556
Attribute 10
Aufgaben abstecken 39
Ausgabeschicht 290
ausgeglichene Lernraten 606
Ausreißererkennung 240
Aussagenlogik 282
Auswahl von Merkmalen 28
Autodiff 769
 Differenzierung von Hand 769
 Forward-Modus 771
 Reverse-Modus 774
Autoencoder 571
 Convolutional 583
 Denoising 584
 effiziente Repräsentation von Daten 572
 generative 589
 PCA mit einem unvollständigen linearen Autoencoder 574
 probabilistische 589
 Rekonstruktionen 574
 rekurrente 584
 Sparse 586
 Stacked 575
 untvollständig 574
 Variational 589
AutoGraph 411
Automatic Differentiation (Autodiff) 291, 402
AutoML 325
Autoregressive Integrated Moving Average (ARIMA), Modelle 510
Average Precision (AP) 494
Average-Pooling Layer 462

B

Backpropagation 290–294, 334
Bagging und Pasting 195
 Out-of-Bag-Evaluation 197
 in Scikit-Learn 196
Bahdanau-Attention 555
Batched-Aktionsschritt 658
Batched-Trajektorie 658
Batched-Zeitschritt 658

Batch-Gradientenverfahren 124
Batchgröße 328
Batch-Learning 16
Batchnormalisierung 341, 475
 Zusammenfassung 341
bayessche gaußsche Mischverteilungsmodelle 272
bayessches Informationskriterium (BIC) 269
Beam Search 551
Beam Width 552
bedingte Wahrscheinlichkeiten 552
Beispielprojekt 759
Belohnungen, in RL 612
Beobachter 655
beobachtete Variablen 263
beschleunigter Gradient nach Nesterov (NAG) 356
Betrugserkennung 240
Bias-Neuronen 287
Bias-Term 116
bidirektionale rekurrente Schichten 551
Bilderzeugung 499
Bildklassifikation 466
 Multitask-Klassifikation 313
Bildsegmentierung 240, 251
binäre Klassifikatoren 91, 144
biologische Neuronen 282
BIRCH-Algorithmus 261
Black Box Stochastic Variational Inference (BBSVI) 275
Black-Box-Modelle 180
Blending 210
Boltzmann Machines *siehe* Restricted Boltzmann Machines (RBMs)
Boosting 202
 AdaBoost 202
 Gradient Boosting 205
Bootstrap-Aggregation *siehe* Bagging und Pasting
Bootstrapping 79, 195
Bounding Box Priors 493
Brechen der Symmetrie 292
brew 212
Byte-Pair Encoding 540

C

Callbacks 317
Canary Testing 684
CART-Algorithmus (Classification and Regression Tree) 179, 186
Character-RNNs (Char-RNNs)
 Einsatz 535
 sequenzielle Datensätze aufteilen 532
 Shakespeare-Text erzeugen 536
 Trainingsdatensatz erzeugen 531
 Überblick 530

zustandsbehaftete RNNs 537
Chatbot 529
Chi-Quadrat-Test 184
Clustering, hierarchisches 12
Clustering-Algorithmen 12
 Bildsegmentierung 240, 251
 DBSCAN 257
 teilüberwachtes Lernen 254
 Überblick 238
 Vorverarbeitung 253
 weitere Algorithmen 260
 Ziel 238
Cluster-Spezifikation 712
CNN-Architekturen 464–481
Coderraum 591
Coding 571
Colab Runtime 694
Colaboratory (Colab) 694
Collect Policy 650
Complementary-Slackness-Bedingung 766
components_ 224
Compute Unified Device Architecture (CUDA), Bibliothek 691
concat() 471
Concatenative Attention 555
Contrastive Divergence 781
Convolutional Autoencoder 583
Convolutional Neural Networks (CNNs) 449
 Architekturen 464
 AlexNet 467
 GoogleNet 470
 LeNet5 466
 ResNet 474
 Convolutional Layer 451, 471
 Feature Maps 454
 Filter 453
 Speicherbedarf 459
 Entstehung von 450
 Implementierung in TensorFlow 456
 Klassifikation und Lokalisation 487
 Objekterkennung 488–495
 Pooling Layer 460
 ResNet-34 mit Keras 481
 semantische Segmentierung 495
 vortrainierte Modelle aus Keras 482
 vortrainierte Modelle zum Transfer Learning 484
Convolution-Kernels 453, 464, 471
Credit-Assignment-Problem 621
CUDA Deep Neural Network (cuDNN), Bibliothek 691
Curiosity-Based Exploration 665

D

Data Mining 6

Data-API (TensorFlow)

- Datasets mit tf.keras verwenden 427
- Daten durchmischen 420
- Prefetching 425
- Transformationen verketten 419
- Überblick 418
- Vorverarbeiten von Daten 423

DataFrame 65

Dataquest [XXI](#)

Data-Snooping-Bias 53

Daten 49

- siehe auch* Testdatensatz; Trainingsdaten
- Analyse durch Clustering 239
- Annahmen treffen über 34
- Arbeitsumgebung erstellen für 44
- Datasets mit tf.keras verwenden 427
- Datendiskrepanz 33
- durchmischen 420
- Fashion-MNIST-Datensatz 578, 593
- Flat Datasets 533
- Google News 7B Corpus 546
- herunterladen 48, 49
- Internet Movie Database 539
- Korrelationen finden in 60
- laden und vorverarbeiten mit TensorFlow 417–446
- Nested Datasets 533
- Prefetching 425
- sequenzielle Datensätze aufteilen 532
- Testdatensatz erstellen 53
- Trainingsdatensatz erstellen 531
- Umgang mit realen Daten 37
- vorbereiten für Machine-Learning-Algorithmen 64
- vorverarbeiten 253, 423, 434–443

Datenaufbereitung 64

Datenpipeline 40

Datenstruktur 52

Datenvisualisierung 58

- Fashion-MNIST-Datensatz 578
- TensorBoard 318

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) 257

Decision Stumps 205

Decision Trees 75

Decoder 573

Deep Autoencoder *siehe* Stacked Autoencoder

Deep Belief Networks (DBNs) 15, 782

Deep Convolutional GANs 601

Deep Learning *siehe* Reinforcement Learning; TensorFlow

Deep Learning VM Images 693

Deep-Learning-Netze (DNNs) 333

Definition [XVII](#)

instabile Gradienten 334
Regularisierung 367
Richtlinien zum Trainieren 374
schnellere Optimierer für 354
schwindende und explodierende Gradienten 333
Wiederverwenden vortrainierter Schichten 348
DeepMind 16, 611, 635
Deep-Q-Learning 635
 Double DQN 641
 Dueling DQN 643
 feste Q-Wert-Ziele 640
 Implementierung 636
 priorisiertes Experience Replay 642
Deep-Q-Netze (DQNs) 635, 652
Denoising Autoencoder 584
Dense Layer 287
Dense-Vektoren 560
Depth Concat Layer 470
Depthwise Separable Convolution 477
Deque 636
describe() 50
Dev Set 32
Dichteabschätzung 238, 265
Dienstkonten 682
Differencing 510
Dimensionsreduktion 12, 215, 571
 Ansätze zur
 Manifold Learning 219
 Projektion 217
 Auswählen der richtigen Anzahl Dimensionen 225
 per Clustering 239
 und Datenvisualisierung 215
 Fluch der Dimensionalität 215
 Isomap 234
 LLE (Locally Linear Embedding) 231
 multidimensionale Skalierung 233, 234
 PCA (Hauptkomponentenzerlegung) 221
 t-verteiltes Stochastic Neighbor Embedding (t-SNE) 234
Diskriminatoren 572
Distribution Strategies API 710
Double DQN 641
Double Dueling DQN 644
DQN-Agenten 654
drop() 64
dropna() 65
Drop-out-Rate 368
duale Zahlen 772
duales Problem 170
Dualität 765
Dueling-DQN-Algorithmus 643
Dummy-Merkmale 68

dynamische Modelle 315
dynamische Programmierung 629

E

Eager Execution/Eager-Modus 411
Early Stopping 143, 327
eigene Modelle 379
Aktivierungsfunktionen, Initialisierer, Regularisierer, Constraints 391
Gradienten mit Autodiff berechnen 402
Metriken 392
Modelle 398
Schichten 395
sichern und laden 389
Trainingsschleifen 406
Verlustfunktion 388
Verlustfunktionen und Metriken 400
eindimensionale Convolutional Layer 524
Eingabeneuronen 287
Eingabeschichten 290
Eingabesignaturen 795
Elastic Net 142
Embedded Devices 685
Embedded Reber Grammars 569
Embedding-Matrix 439
Embeddings 70, 417, 437
Empfehlungssysteme 239
Encoder 573
Encoder-Decoder 506
Encoder-Decoder-Modell 547–553
End-of-Sequence-(EoS-)Token 548, 559
Energiefunktion 778
Ensemble Learning 77, 81, 191
 Bagging und Pasting 195
 Beispiele 191
 Boosting 202
 Random Forests *siehe* Random Forests
 Stacking 210
 zufällige Patches und zufällige Subräume 198
Entailment 566
Entropie als Maß für Unreinheit 182
Entscheidungsbäume 77, 189
 Anzahl Kinder 179
 Binärbäume 179
 Entscheidungsgrenzen 180
 geschätzte Wahrscheinlichkeiten für Kategorien 181
 Gini-Unreinheit 182
 Hyperparameter zur Regularisierung 183
 Instabilität bei 187
 Komplexität der Berechnung 182
 Random Forests *siehe* Random Forests

Regressionsaufgaben 185
trainieren und visualisieren 177
Vorhersagen 178
Entscheidungsfunktion 97, 166
Entscheidungsgrenzen 97, 147, 152, 180
Entwicklungsdatensatz 32
Episoden (beim RL) 618, 622
Epochen 128, 291
 ϵ -greedy-Policy 634
 ϵ -insensitiv 164
Erkundungspolicies 634
Estimatoren 66
euklidischer Abstand 43
Event Files 318
Evidence Lower Bound 274
Expectation-Maximization-(EM-)Algorithmus 264
Expectation-Schritt 264
Experience Replay 600
Explainability 557
explodierende Gradienten *siehe* Gradienten, schwindende und explodierende
Exponential Linear Unit (ELU) 338
Extra-Trees 200

F

F1-Score 96
Falsch-positiv-Rate (FPR) 100
Fan-in 335
Fan-out 335
Farbsegmentierung 252
Fashion-MNIST-Datensatz 578, 593
Fast-MCD (Minimum Covariance Determinant) 276
Feature Maps 454, 475
Feed-Forward-Netze (FNNs) 290
Fehler, nicht reduzierbarer 136
Fehleranalyse 106
feste Q-Wert-Ziele 640
fillna() 65
First-In-First-Out-(FIFO-)Queues 387
fit() 65, 72, 227
Fitnessfunktion 21
fit_transform() 66, 72
Flaschenhalsschichten 471
Flat Datasets 533
Fluch der Dimensionalität 217
 siehe auch Dimensionsreduktion
Folds 76, 91
Forecasting 507
Forget Gate 520
Forward-Modus in Autodiff 771
Fotodienste 15

Freiheitsgrade 29, 136
frühes Anhalten 208
Fully Convolutional Networks (FCNs) 490
Functional API 310–314
Funktionsdefinitionen 796
Funktionsgraphen 796

G

gamma, Wert 162
Gate-Controller 521
gaußsche Mischverteilungsmodelle
 Anomalieerkennung 267
 bayessche gaußsche Mischverteilungsmodelle 272
 Cluster-Anzahl 269
 grafisches Modell 262
 Überblick 262
 Varianten 262
 weitere Algorithmen zur Anomalie- und Novelty-Erkennung 276
gaußsche RBF 161
gaußscher RBF-Kernel 162, 172
Gedächtniszellen 504
Genauigkeit 4, 92
Generative Adversarial Networks (GANs)
 Deep Convolutional GANs (DCGANs) 601
 Einsatz 571
 progressiv wachsend 604
 Schwierigkeiten beim Training 599
 StyleGANs 607
 Überblick 595
generatives Modell 265
Generatoren 572
geodätische Distanz 234
Gesetz der großen Zahl 193
Gesichtserkennung 109
Gini-Unreinheit 179, 182
Gittersuche 78, 161
Glättungsterme 357, 360
Gleichgewicht zwischen Bias und Varianz 136
Gleichheit als Nebenbedingung 766
Gleichung mit geschlossener Form 115, 146
Global Average Pooling Layer 463
Glorot-Initialisierung 335
Google Cloud Platform (GCP)
 Vorhersageservice erstellen 677–681
 Vorhersageservice verwenden 682–685
Google Cloud Storage (GCS) 679
Google Images 281
Google News 7B Corpus 546
Google Photos 15
GoogleNet, Architektur 470

GPUs (Graphics Processing Units)
auswählen 690
Berechnungen beschleunigen 689
Colaboratory (Colab) 694
das RAM der GPU verwalten 695
Operationen und Variablen auf Devices verteilen 698
paralleles Ausführen auf mehreren Devices 700
virtuelle Maschinen mit GPU 693
zu einer einzelnen Maschine hinzufügen 690

Gradient Boosted Regression Trees (GBRT) 206

Gradient Boosting 205

Gradient Tree Boosting 206

Gradienten, schwindende und explodierende 333
Batchnormalisierung 341
Gradient Clipping 347
Initialisierung nach Glorot und He 334
nicht sättigende Aktivierungsfunktionen 337

Gradientenaufstieg 615

Gradientenverfahren (GD) 115, 121, 173, 355, 357
Batch GD 124
Definition 121
lokales versus globales Minimum 123
Mini-Batch-GD 130
stochastisches Gradientenverfahren 127, 158
Vergleich von Algorithmen 130

graphviz 178

Greedy Layer-Wise Pretraining 352

Greedy-Algorithmus 182

Grenzübergänge 662

GRU-(Gated-Recurrent-Unit-)Zelle 522

Grundlagen von Machine Learning
Arten von Systemen 9
Batch- und Online-Learning 16
instanzbasiertes versus modellbasiertes Lernen 18
überwachtes/unüberwachtes Lernen 9

Attribute 10

Beispiel Arbeitsablauf 24

Definition 4

Gründe für die Verwendung 4

Herausforderungen 24
Probleme mit Algorithmen 28
Probleme mit Trainingsdaten 28

Merkmale 10

Testen und Validieren 31

Überblick 3

Workflow-Beispiel 20

Zusammenfassung 31

H

halbüberwachtes Lernen 14

Hard Clustering 242
Hard-Margin-Klassifikation 156
Hard-Voting-Klassifikatoren 192
harmonischer Mittelwert 96
Hauptkomponente 222
Hauptkomponentenzerlegung (PCA) 221
 als Komprimierungsverfahren 226
 Anteil erklärter Varianz 224
 Finden von Hauptkomponenten 222
 inkrementelle PCA 227
 Kernel-PCA (kPCA) 228
 Projektion auf d Dimensionen 223
 randomisiert 227
 Scikit-Learn für 224
 Varianz, erhalten 221
 zur Komprimierung 227
Heaviside-Funktion 286
hebbische Lernregel 288, 777
He-Initialisierung 335
Hierarchical DBSCAN (HDBSCAN) 260
Hinge-Loss-Funktion 174
Hinton, Geoffrey XVII
Histogramm 51
Hold-out-Datensatz *siehe* Blending
Hold-out-Validierung 32
Hopfield-Netze 777
Hyperas 324
Hyperband 325
Hyperebene 167, 220, 223, 234
Hyperopt 324
Hyperparameter 30, 71, 78, 121, 161, 164
 siehe auch Hyperparameter neuronaler Netze
 anpassen 32
 Python-Bibliotheken zum Optimieren 324
Hyperparameter neuronaler Netze 322
 Anzahl verborgener Schichten 326
Hypothese 42
 Manifold 220
Hypothesenfunktion 117
Hypothesis Boosting *siehe* Boosting

I

Identitätsmatrix 139, 169
ILSVRC ImageNet Challenge 466
Importance Sampling (IS) 642
Imputation 507
Inception-Module 470
Inferenz 24, 376, 459
info() 49
Informationstheorie 183

Initialisierung
Methoden zur Schwerpunktinitialisierung 245
nach Glorot und He 335
nach Xavier 334
inkrementelles Lernen 18, 227
Inlier 267
Input Gate 520
instabile Gradienten 516
Instance Segmentation 498
instanzbasiertes Lernen 19, 23
Instanzsegmentierung 251
Internet Movie Database 539
Inter-Op Thread Pool 700
Intra-Op Thread Pool 700
inverse_transform() 231
Isolation-Forest-Algorithmus 276
isolierte Umgebung 45, 46
Isomap 234

J

Jupyter 44, 46, 52

K

Karte-Kuhn-Tucker-(KKT-)Bedingungen 766
katastrophales Vergessen 639
kategoriale Verteilung 263
Kategorie, vorhergesagte 94
kategorische Merkmale 67
Codieren mit Embeddings 437
kausales Modell 514
keep-Wahrscheinlichkeit 370
Keras
Batchnormalisierung 344
Codebeispiele von keras.io 301
Drop-out mit 370
komplexe Architekturen 316
MLPs implementieren 297
ResNet-34 implementieren 481
Sichern und Wiederherstellen von Modellen 316
Stacked Autoencoder 576
Vorteile XIX
vortrainierte Modelle 482
Vorverarbeitungsschichten 441
Keras Tuner 324
Kernel 160, 380
Kernel-PCA (kPCA) 228
Kernel-SVM 171
Kerneltrick 160, 162, 171, 228
Kerninstanz 257
Kettenregel 292

k-fache Kreuzvalidierung 76, 93
Klassifikation versus Regression 10, 111
Klassifikationsprobleme
 Klassifikation und Lokalisation 487
 Klassifikations-MLPs 295
 Multitask-Klassifikation 313
Klassifikatoren
 Abstimmverfahren 192
 auswerten 105
 binäre 91
 Fehleranalyse 106
 für das stochastische Gradientenverfahren (SGD) 91
 mehrere Ausgaben 110
 mehrere Kategorien 103
 mehrere Labels 109
 MNIST-Datensatz 89
 Qualitätsmaße 92
 Relevanz von 95
K-Means
 Algorithmus 243
 Bildsegmentierung 251
 Eingangsmerkmale skalieren 251
 Grenzen 250
 Hard und Soft Clustering 242
 Methoden zur Schwerpunktinitialisierung 245
 optimale Cluster-Anzahl 247
 teilüberwachtes Lernen 254
 Überblick 240
 Verbesserungsvorschlag 245
 Vorverarbeitung 253
K-Means-Algorithmus 240–251
k-nächste-Nachbarn 23, 110
Komplexität der Berechnung 120, 163, 182
Konfusionsmatrix 94, 95, 106
konkrete Funktionen 795
Konnektionismus 282
Konvergenzrate 127
konvexe Funktion 123
Kopplung von Gewichten 580
kopt (Bibliothek) 324
Korrelationen, finden 60
Korrelationskoeffizient 60
Kostenfunktion 21, 43
 bei AdaBoost 203
 bei AdaGrad 357
 bei Elastic Net 142
 bei der Gradientenmethode 209
 beim Gradientenverfahren 115, 121, 124, 127, 334
 Kreuzentropie 467
 bei der Lasso-Regression 139
 bei der linearen Regression 118, 123

bei der logistischen Regression 146
Mean Squared Error 577
mittlerer absoluter Fehler (MAE) 294
mittlerer quadratischer Fehler 294, 310
bei der Momentum Optimization 355
bei der Ridge-Regression 137
Stale Gradients und 708
bei Variational Autoencodern 591
Kreuzentropie 151
Kreuzentropie-Verlustfunktion (Log-Loss) 296
Kreuzvalidierung 33, 76, 92
Kullback-Leibler-Divergenz 151
Kundensegmentierung 239
künstliche neuronale Netze (KNNs) 281
 Boltzmann Machines 779
 Deep Belief Networks (DBNs) 782
 Evolution von 282
 Hopfield-Netze 777
 mit Keras implementieren 297
 Optimieren von Hyperparametern 322
 Perzeptrons 286
 selbstorganisierende Karten 784
 Überblick 281
Kurzzeitgedächtnisprobleme 518–526

L

Label 9, 41, 241
Label Propagation 256
Lagrange-Funktion 766
Lagrange-Multiplikator 765
Landmarken 161
lange Sequenzen
 instabile Gradienten 516
 Kurzzeitgedächtnis 518–526
 Überblick 515
Large-Margin-Klassifikation 155
Lasso-Regression 139
latente Variablen 263
latenter Raum 591
latenter Verlust 591
Layer Normalization 517
Leaky ReLU 337
Learning-Algorithmus, Deep-Q-Learning 635
LeCun-Initialisierung 336
LeNet-5-Architektur 451, 466
Lernrate 18, 121, 125, 328, 606
 vorgefertigte stückweise konstante 363
Levenshtein-Distanz 163
liblinear, Bibliothek 163
libsvm, Bibliothek 164

Likelihood-Funktion 269
Linear Threshold Units (LTUs) 286
lineare Diskriminantanalyse (LDA) 234
lineare Klassifikation mit SVMs 155
lineare Modelle
 Early Stopping 143
 Elastic Net 142
 Lasso-Regression 139
 lineare Regression *siehe* lineare Regression
 Regression *siehe* lineare Regression
 Regularisieren von Modellen *siehe* Regularisierung
 Ridge-Regression 137, 142
 SVM 155
lineare Regression 21, 74, 115, 131, 142
 Gradientenverfahren bei 121, 131
 Komplexität der Berechnung 120
 Lernkurven in 133
 mit dem stochastischen Gradientenverfahren (SGD) 129
 Normalgleichung 118, 121
lineare SVM-Klassifikation 158
LinearSVR, Klasse 166
Lipschitz-stetig 123
Listen von Listen 433
LLE (Locally Linear Embedding) 231
Lloyd-Forgy-Algorithmus 241
load_sample_images() 456
Local Outlier Factor (LOF) 276
Local Response Normalization 469
log loss 146
logische GPU-Devices 696
logistische Funktion 145
logistische Funktion (sigmoid) 304
logistische Regression 10, 144
 Abschätzen von Wahrscheinlichkeiten 145
 Entscheidungsgrenzen 147
 Softmax-Regressionsmodell 149, 152
 trainieren und Kostenfunktion 146
Logit 145
Log-Odds 146
lokales Wahrnehmungsfeld 450
Lokalisation 487
Long-Short-Term-Memory-(LSTM-)Zelle 519
Luong-Attention 555

M

Machine Learning (ML)
 Anwendungen XVIII, 7
 behandelte Themen XIX
 Beispielprozess 37
 Checkliste für Projekte 39, 759

Geschichte [XVII](#)
Lernansatz [XVIII](#)
Ressourcen zu [XXI](#)
Schreibweisen [42](#)
Voraussetzungen [XIX](#)
Manhattan-Metrik [43](#)
Manifold Learning [219, 231](#)
siehe auch LLE (Locally Linear Embedding)
Manifold-Annahme/Hypothese [220](#)
MapReduce [41](#)
Margin-Verletzungen [157](#)
Markov-Entscheidungsprozesse [627](#)
Markov-Ketten [627](#)
Maschinensteuerung *siehe* Reinforcement Learning (RL)
Mask R-CNN [498](#)
Masked Language Model (MLM) [567](#)
Masked Multi-Head-Attention-Schichten [559](#)
Maskieren [543](#)
Maskierungstensoren [543](#)
Maße für Unreinheit [179, 182](#)
Matplotlib [44, 52, 100, 106](#)
Maximization-Schritt [264](#)
Maximum-a-posteriori-Schätzung [271](#)
Maximum-Likelihood-Schätzung (MLE) [270](#)
Max-Norm-Regularisierung [374](#)
Max-Pooling Layer [460](#)
mean Average Precision (mAP) [494](#)
Mean Field Variational Inference [274](#)
Mean-Shift-Algorithmus [261](#)
mehrschichtige Perzeptrons (MLP) [281](#)
memmap [228](#)
Mercers Theorem [172](#)
Merkmale [10](#)
siehe auch Datenstruktur
entwickeln [28](#)
extrahieren [12](#)
Kombinationen von [63](#)
skalieren [71](#)
vorverarbeitete [52](#)
Wichtigkeit von [200](#)
Ziel [52](#)
Merkmalsauswahl [80, 140, 201, 762](#)
Merkmalserkennung [571](#)
Merkmalsraum [228](#)
Merkmalsvektor [42, 117, 166](#)
Merkmalszuordnung [230](#)
Metagraphen [671](#)
Meta-Lerner *siehe* Blending
Metriken
 Genauigkeit [392](#)
 Mean Average Precision [494](#)

mittlerer absoluter Fehler (MAE) 294
Microsoft Cognitive Toolkit (CNTK) 297
Mini-Batch Discrimination 600
Mini-Batches 17
Mini-Batch-Gradientenverfahren 130, 147
Min-Max-Skalierung 71
mittlere quadratische Abweichung (MSE) 117
mittlerer absoluter Fehler (MAE) 43
mittleres Coding 590
Mixing Regularization 609
ML Engine 680
MNIST-Datensatz 89
mobile Geräte 685
Modalwert 196
Mode Collapse 600
Modellauswahl 21, 32
modellbasiertes Lernen 19
Modelle
 siehe auch eigene Modelle
 analysieren 81
 auf mehreren Devices trainieren 702–718
 Callbacks 317
 dynamische mit der Subclassing API 315
 eigene mit TensorFlow 388–409
 evaluieren 31
 evaluieren auf dem Testdatensatz 82, 83
 generative 571, 780
 kausale Modelle 514
 komplexe Modelle mit der Functional API 310–314
 parameterfreie 183
 parametrische 183
 Sequence-to-Sequence-Modelle 514
 Sichern und Wiederherstellen 316
 TensorBoard zur Visualisierung 318
 trainieren 21
 logistische Regression 152
 vortrainierte Modelle aus Keras 482
 vortrainierte Modelle zum Transfer Learning 484
 Modellparameter 124, 126, 144, 166
 Definition 21
 Module, TensorFlow Hub 545
 Moment 359
 Momentum Optimization nach Nesterov 356
 Multibackend-Keras 297
 multidimensionale Skalierung (MDS) 233
 Multi-Head-Attention-Schicht 559, 562
 Multi-Layer-Perceptrons (MLPs)
 Backpropagation 290–294
 Klassifikations-MLPs 295
 Regressions-MLPs 294
 multinomiale logistische Regression *siehe* Softmax-Regression

multiple Regression [41](#)
multiplikative Attention [555](#)
Multitask-Klassifikation [313](#)
multivariate Regression [41](#)
multivariate Zeitserie [507](#)

N

naive Bayes-Klassifikatoren 103
naives Forecasting 508
Nash-Gleichgewicht 599
natürliche Sprachverarbeitung (NLP) 354
 aktuelle Entwicklungen 566
 Attention-Mechanismen 553–565
 Encoder-Decoder-Netzwerk 547–553
 Sentimentanalyse 539–547
 Text mit Character-RNNs erzeugen 530–539
 Übersicht 529
Nebenbedingungen, aktive 766
Nested Datasets 533
Netzwerkbandbreite, erschöpfen 709
neuronale maschinelle Übersetzung (NMT) 547–565
neuronale Netze als Policies 619
Neuronen
 biologische 282
 logische Berechnungen mit 285
 pro verborgener Schicht 327
Newtons Differenzenquotient 770
Next Sentence Prediction (NSP) 567
nicht sequenzielle neuronale Netze 310
nichtlineare Dimensionsreduktion (NLDR) *siehe* Kernel-PCA; LLE (Locally Linear Embedding)
nichtlineare SVM-Klassifikation 159
 ähnlichkeitsbasierte Merkmale, hinzufügen 161, 162
 gaußscher RBF-Kernel 162
 Komplexität der Berechnung 163
 mit polynomiellen Merkmalen 159
 polynomielles Kernel 160
No-Free-Lunch-Theorem 34
Non-Max Suppression 489
Normalengleichung 118
Normalisieren 71
normalisierte Exponentialfunktion 150
Normalverteilung 590
Normen 43
Novelty Detection 13, 268, 276
NP-vollständige Probleme 182
Nullhypothese 184
NumPy 44
 TensorFlow einsetzen 383–388
NumPy-Arrays 68
Nutzenfunktion 21
NVIDIA Collective Communications Library (NCCL) 711
Nvidia-GPU-Karten 691

O

Objectness-Ausgabe 489

Objekterkennung
 Fully Convolutional Networks (FCNs) 490
 Überblick 488
 You Only Look Once (YOLO) 492
Offline-Learning 16
Off-Policy-Algorithmus 634
One-Class-SVM-Algorithmus 276
One-Hot-Codierung 68
One-versus-All-(OvA-)Strategie 104
One-versus-One-(OvO-)Strategie 104
One-versus-the-Rest-(OvR-)Strategie 104, 175
Online-Learning 17
Onlinemodell 640
Online-SVMs 173
On-Policy-Algorithmen 634
OpenAI Gym 615
Optimalitätsprinzip von Bellman 629
Optimierer 354
 AdaGrad 357
 Adam-Optimierung 359, 361
 beschleunigter Gradient nach Nesterov (NAG) 356
 Gradientenverfahren *siehe* Gradientenverfahren (GD)
 Momentum Optimization 354
 RMSProp 358
 Scheduling der Lernrate 362
Optimierung mit Nebenbedingungen 168, 765
optische Zeichenerkennung (OCR) 3
Outlier Detection 267
Out-of-Bag-Evaluation 197
Out-of-Core-Learning 17
out-of-sample error 31
Out-of-Vocabulary-(OOV-)Buckets 436
Output Gate 520
Overfitting 28, 53, 157, 162, 183, 186, 327
 vermeiden durch Regularisierung 367

P

pandas 44, 48
 scatter_matrix 61
parallelisierte Daten 702, 705
 synchrone Updates 707
parallelisierte Modelle 702
Parametereffizienz 326
Parametermatrix 150
Parameterraum 124
Parameterserver 707
Parametervektor 117, 121, 146, 150
partial_fit() 227
partielle Ableitung 124
partielle Ableitungen erster Ordnung (Jacobians) 361

partielle Ableitungen zweiter Ordnung (Hessians) 361
PCA (Principal Component Analysis)
Anomalie- und Novelty-Erkennung 276
Pearson 60
Peephole-Verbindungen 522
Performance Scheduling 364
permutation() 54
Perzentil 51
Perzeptron-Konvergenztheorem 288
Perzeptrons 286
im Vergleich zu logistischer Regression 289
trainieren 288
PG-Algorithmen 622, 627
pip 45
Pipeline-Konstruktor 72
Pipelines 40, 428
Pipelines zur Transformation 72
Policy 613
Policy-Gradienten 615
siehe auch PG-Algorithmen
Policy-Raum 614
polynomielle Merkmale, hinzufügen 159, 160
polynomielle Regression 116, 131
Lernkurven bei 133, 137
polynomieller Kernel 160, 172
Pooling Layer 460
Pooling-Kernel 460
Positional Encodings 560
Post-Training-Quantisierung 686
Power Scheduling 363
Prädiktoren 10, 66
predict() 66
PReLU (parametrisierte Leaky ReLU) 338
primales Problem 170
priorisiertes Experience Replay (PER) 642
Projektion 217
Protocol Buffer (Protobufs) 429
Proximal Policy Optimization (PPO) 665
Pruning 184, 772
p-Wert 184
Python
isolierte Umgebung in 45
Notebooks in 46
pickle 78
pip 45

Q

Q-Learning-Algorithmus 632
aproximatives Q-Learning 634
Deep-Q-Learning 635

quadratische Programme (QP) [169](#)
Qualitätsmaße [41](#)
Konfusionsmatrix [94](#)
Kreuzvalidierung [92](#)
Relevanz und Sensitivität [96](#)
ROC-(Receiver-Operating-Characteristic-)Kurve [100](#)
quantisierungsbewusstes Training [688](#)
Queries pro Sekunde [667](#)
Queues [387, 792](#)
Q-Werte [629](#)
Q-Wert-Iterationsalgorithmus [630](#)

R

radiale Basisfunktion (RBF) [161](#)
Ragged-Tensoren [387, 788](#)
Rainbow-Agent [644](#)
Random Forests [77, 103, 177, 188, 191, 199](#)
 Extra-Trees [200](#)
 Wichtigkeit von Merkmalen [200](#)
randomisierte Leaky ReLU (RReLU) [338](#)
randomisierte PCA [227](#)
rechtsschief [52](#)
Recognition Network [573](#)
Reconstruction Pre-Image [230](#)
Region Proposal Network [495](#)
Regression [10](#)
 Entscheidungsbäume [185](#)
 versus Klassifikation [111](#)
Regressionsmodelle, lineare [74](#)
Regressionsprobleme
 Regressions-MLPs [294](#)
 Regressions-MLPs mit der Sequential API [309](#)
reguläre Ausdrücke [541](#)
Regularisierung [29, 32, 137, 144](#)
 Drop-out [368](#)
 Early Stopping [143](#)
 Elastic Net [142](#)
 Entscheidungsbäume [183](#)
 ℓ_1 - und ℓ_2 -Regularisierung [367](#)
 Lasso-Regression [139](#)
 nach Tikhonov [137](#)
 Ridge-Regression [137](#)
 Shrinkage [207](#)
REINFORCE-Algorithmen [622](#)
Reinforcement Learning (RL) [15, 611](#)
 Aktionen [621](#)
 Beispiele für [612](#)
 Belohnungen, lernen zu optimieren [612](#)
 Credit-Assignment-Problem [621](#)
 Markov-Entscheidungsprozesse [627](#)

neuronale Netze als Policies 619
OpenAI Gym 615
PG-Algorithmen 622
Policy-Suche 615
Q-Learning-Algorithmus 632
Suche nach Policies 613
Temporal Difference (TD) Learning 631
TF-Agents-Bibliothek 644–663
Rekonstruktionen 574
Rekonstruktionsfehler 226
Rekonstruktionsverlust 401, 574, 591
rekurrente Autoencoder 584
rekurrente neuronale Netze (RNNs) 501
 Ein- und Ausgabesequenzen 505
 Erkundungspolicies 634
 GRU-Zelle 522
 lange Sequenzen 515–526
 Text mit Character-RNNs erzeugen 530–539
Training 506
Zeitserien vorhersagen 507–515
zustandslos und zustandsbehaftet 529, 537
rekurrente Neuronen 502
 Gedächtniszellen 504
Relevanz 95
Relevanz und Sensitivität 96
 F1-Score 96
 Relevanz-Sensitivitäts-Kurve (PR-Kurve) 101
 Wechselbeziehung zwischen Relevanz und Sensitivität 97
ReLU (Rectified Linear Unit Funktion) 293
ReLU-Aktivierungsfunktion 475
Replay Buffer 636, 650, 655
Replay Memory 636
Representation Learning 70, 438
Residual Block 398
Residual Learning 474
Residual Network (ResNet) 474
Residual Units 475
ResNet 474
ResNet-34-CNN 481
Responsibilities 264
Restfehler 205
Restricted Boltzmann Machines (RBMs) 15, 780
Reverse-Mode-Autodiff 291
rgb_array 617
Richtig-negativ-Rate (TNR) 100
Richtig-positiv-Rate (TPR) 95, 100
Ridge-Regression 137, 142
RMSProp 358
ROC-(Receiver-Operating-Characteristic-) Kurve 100
Root Mean Square Error (RMSE) 117
RReLU (randomisierte Leaky ReLU) 338

S

Sampled-Softmax-Technik 549
Sample-ineffizient 626
SavedModel-Format 669
Scaled Exponential Linear Unit (SELU), Funktion 339, 371
Scaled-Dot-Product-Attention-Schicht 562
Scheduling der Lernrate 128, 362
Scheduling, exponentielles 363
Schichten
 Adaptive Instance Normalization (AdaIN) 608
 bidirektionale rekurrente Schicht 551
 eindimensionale Convolutional Layer 524
 Masked Multi-Head-Attention-Schicht 559
 Multi-Head-Attention-Schicht 559, 562
 Scaled-Dot-Product-Attention-Schicht 562
Schreibweisen 42
Schrittweite 453
schwache Lerner 193
Schweigeverzerrung 27
Schwerpunkt 240
schwindende Gradienten *siehe* Gradienten, schwindende und explodierende
Scikit-Learn 44
 Anomalie- und Novelty-Erkennung 276
 Bagging und Pasting in 196
 CART-Algorithmus 179, 186
 Clustering-Algorithmen 260
 cross_val_score() 92
 Design 66
 Hyperparameter min_ und max_ 184
 Klassen zur SVM-Klassifikation 164
 Kreuzvalidierung 76
 lineare Regression 120
 LinearSVR, Klasse 166
 MinMaxScaler 71
 One-Hot-Vektoren 68
 PCA-Implementierung 224
 Perceptron-Klasse 289
 Pipeline-Konstruktur 72, 159
 Ridge-Regression mit 139
 SAMME 205
 SGDClassifier 91, 98, 104
 SGDRegressor 129
 sklearn.base.BaseEstimator 70, 93
 sklearn.base.clone() 92, 144
 sklearn.base.TransformerMixin 70
 sklearn.datasets.fetch_mldata() 89
 sklearn.datasets.load_iris() 147, 177, 201, 289
 sklearn.datasets.make_moons() 159, 188
 sklearn.decomposition.IncrementalPCA 227
 sklearn.decomposition.KernelPCA 228, 231

sklearn.decomposition.PCA 224
sklearn.ensemble.AdaBoostClassifier 205
sklearn.ensemble.BaggingClassifier 196
sklearn.ensemble.GradientBoostingRegressor 206
sklearn.ensemble.RandomForestClassifier 102, 194
sklearn.ensemble.RandomForestRegressor 77, 78, 199, 206
sklearn.ensemble.VotingClassifier 194
sklearn.externals.joblib 78
sklearn.linear_model.ElasticNet 142
sklearn.linear_model.Lasso 142
sklearn.linear_model.LinearRegression 22, 66, 74, 131, 132, 134
sklearn.linear_model.LogisticRegression 148, 149, 152, 194, 229
sklearn.linear_model.Perceptron 289
sklearn.linear_model.Ridge 139
sklearn.linear_model.SGDClassifier 92
sklearn.linear_model.SGDRegressor 129, 139, 142
sklearn.manifold.LocallyLinearEmbedding 231
sklearn.metrics.accuracy_score() 194, 198
sklearn.metrics.confusion_matrix() 94, 106
sklearn.metrics.f1_score() 96, 110
sklearn.metrics.mean_squared_error() 75, 82, 134, 144, 208, 231
sklearn.metrics.precision_recall_curve() 98
sklearn.metrics.precision_score() 96, 100
sklearn.metrics.recall_score() 96, 100
sklearn.metrics.roc_auc_score() 101, 102
sklearn.metrics.roc_curve() 100, 102
sklearn.model_selection.cross_val_predict() 94, 98, 102, 106, 110
sklearn.model_selection.cross_val_score() 76, 92
sklearn.model_selection.GridSearchCV 78, 87, 106, 189, 229
sklearn.model_selection.StratifiedKFold 92
sklearn.model_selection.StratifiedShuffleSplit 56
sklearn.model_selection.train_test_split() 55, 75, 134, 188, 208
sklearn.multiclass.OneVsOneClassifier 105
sklearn.neighbors.KNeighborsClassifier 110, 112
sklearn.neighbors.KNeighborsRegressor 24
sklearn.pipeline.Pipeline 72, 135, 229
sklearn.preprocessing.Imputer 65
sklearn.preprocessing.PolynomialFeatures 132, 135, 138, 159
sklearn.preprocessing.StandardScaler 72, 106, 124, 138, 156, 162
sklearn.svm.LinearSVC 157, 164, 166, 175
sklearn.svm.LinearSVR 165
sklearn.svm.SVC 160, 164, 166, 175, 194
sklearn.svm.SVR 87, 166
sklearn.tree.DecisionTreeClassifier 184, 189, 196, 199, 205
sklearn.tree.DecisionTreeRegressor 75, 177, 185, 206
sklearn.tree.export_graphviz() 178
StandardScaler 124
User Guide [XXI](#)
vollständiger SVD-Ansatz 227
Vorteile [XVIII](#)
Scikit-Optimize 324

score() 66
SE-Block 479
SE-Inception 479
selbstnormalisierend 339
selbstorganisierende Karten (SOMs) 784
selbstüberwachtes Lernen 354
Self-Attention-Mechanismen 559
SELU-Funktion (Scaled Exponential Linear Unit) 336
semantische Interpolation 594
semantische Segmentierung 251, 462, 495
Sensitivität 95, 100
Sentence Encoder 546
Sentimentanalyse
 Definition 530
 maskieren 543
 Überblick 539
 vortrainierte Embeddings wiederverwenden 545
Separable Convolution 477
SequenceExample-Protobuf (TensorFlow) 433
Sequence-to-Sequence-Modelle 514
Sequential API, Regressions-MLP 309
Sequenzen 501
 lange Sequenzen 515–526
 Zeitserien vorhersagen 507–515
SE-ResNet 479
Sets 387, 791
Shannons Informationstheorie 183
Shortcut-Verbindungen 474
Shrinkage 207
Sichern und Wiederherstellen von Modellen 316
Sigmoid-Aktivierungsfunktion (logistisch) 304
sigmoide Funktion 145
Silhouettenkoeffizient 249
Silhouettenplot 249
Simulated Annealing 127
simulierte Umwelt *siehe* OpenAI Gym
Single-Shot Learning 499
Singular Value Decomposition (SVD) 223
Singulärwertzerlegung 120
Skalarprodukt 555
Skip-Verbindungen 474
sklearn.neighbors.KNeighborsRegressor 24
Sklearn-Deep 325
Slack-Variable 168
Smoothing-Term 342
Soft Actor-Critic-Algorithmus 664
Soft Clustering 242
Soft Voting 194
Soft-Margin-Klassifikation 156
Softmax-Funktion 150, 295, 491
Softmax-Regression 149

Softplus-Aktivierungsfunktion 294
Spaltenvektoren 117
Spamfilter 3, 10
spärlich besetzte Tensoren 387
spärliche Modelle 140, 361
Spärlichkeitsverlust 587
Sparse Autoencoder 586
Sparse-Matrix 68
Sparse-Tensoren 789
Spearmint (Bibliothek) 325
Speicherbandbreite 426
spektrales Clustering 261
Spezifität 100
Spiegel-Strategie 705
Spiele *siehe* Reinforcement Learning (RL)
Spracherkennung 6
Sprachmodelle 566
 siehe auch natürliche Sprachverarbeitung (NLP)
Sprachverarbeitung (NLP) 501
spurious Patterns 778
Stacked Autoencoder 575
 Fashion-MNIST-Datensatz visualisieren 578
 Implementierung mit Keras 576
 Kopplung von Gewichten 580
 Trainieren mehrerer nacheinander 582
 unüberwachtes Vortrainieren mit 579
 Visualisieren der Rekonstruktionen 577
Stacked Denoising Autoencoder 585
Stacked Generalization *siehe* Stacking
Stacking 210
Stale Gradients 708
Standardisierung 71
StandardScaler 72
starke Lerner 193
Start-of-Sequence-(SoS-)Token 539
stationärer Punkt 765, 767
statische Signifikanz 184
Stemming 113
step() 618
sterbende ReLUs 337
Stichproben-Bias 27, 55
Stichprobenrauschen 26
Stichprobenverzerrung 26
Stimmerkennung 449
stochastische Neuronen 779
stochastische Policy 614
stochastisches Gradient Boosting 209
stochastisches Gradientenverfahren (SGD) 127, 158, 289
 Klassifikator 139
 trainieren 147
Strafen *siehe* Belohnungen, in RL

stratifizierte Stichprobe 55, 93
Streaming-Metrik 393
String-Kernels 163
String-Tensoren 387
Stützvektoren 156
Style Mixing 609
StyleGANs 571, 607
Style-Transfer 607
Subclassing API 315
Subdifferentielle 174
Subgradientenvektor 142
subsample 209
Suchmaschinen 240
Suchraum 80, 325
Summaries (TensorFlow) 318
Support Vector Machines (SVMs) 104, 155
duales Problem 170, 765
Entscheidungsfunktion und Vorhersagen 166
Kernel-SVM 171
lineare Klassifikation 155
Mechanismen von 166
nichtlineare SVM-Klassifikation 159
Online-SVMs 173
quadratische Programme (QP) 169
SVM-Regression 164
Zielfunktionen beim Trainieren 167
svd() 223
symbolische Tensoren 411, 796
symbolisches Differenzieren 772
synchrone Updates 707
Systeme zum autonomen Fahren 501

T

Talos (Bibliothek) 324
Tangens hyperbolicus (tanh-Aktivierungsfunktion) 293, 334, 503
tatsächliche Kategorie 94
teilüberwachtes Lernen Clustering-Algorithmen 240, 254
Temperatur, in der Texterzeugung 536
Temporal Difference (TD) Learning 631
Tensor-Arrays 790
TensorBoard 318
Tensoren 383
Tensoren-Arrays 387
TensorFlow
Autodiff 769
Convolutional Neural Networks und 456
Deployen auf AI-Plattformen 84
eigene Modelle und Training 379
Lernraten-Scheduling implementieren 366
Max-Pooling Layer in 462

tf.concat() 470
tf.reduce_mean() 574, 582
tf.square() 574, 582
tf.train.AdamOptimizer 574
tf.train.MomentumOptimizer 736
Vorteile XVIII
TensorFlow Extended (TFX) 444
TensorFlow Hub 382
TensorFlow Light 381
TensorFlow Model Optimization Toolkit (TF-MOT) 362
TensorFlow, Daten laden und vorverarbeiten
 Data-API 418–428
 Eingabemerkmale vorverarbeiten 434–443
 TensorFlow-Datasets-(TFDS-)Projekt 445
 TF Transform 443
 TFRecord-Format 428–434
 Überblick 417
TensorFlow, eigene Modelle und Training
 Aktivierungsfunktionen, Initialisierer, Regularisierer, Constraints 391
 Gradienten mit Autodiff berechnen 402
 Metriken 392
 Modelle 398
 Schichten 395
 sichern und laden 389
 spezielle Datenstrukturen 787–793
 Trainingsschleifen 406
 Verlustfunktion 388
 Verlustfunktionen und Metriken 400
TensorFlow, Funktionen und Graphen
 AutoGraph und Tracing 411, 795–803
 Regeln für TF Functions 412
 Übersicht 409
TensorFlow, Grundlagen
 Architektur 381
 Betriebssystemkompatibilität 381
 Bibliotheks-Ökosystem 382
 Vorteile 379
TensorFlow, Modelle deployen
 auf mehreren Devices trainieren 702–718
 auf Mobile oder Embedded Devices deployen 685–689
 mit GPUs Berechnungen beschleunigen 689–702
 TensorFlow-Modelle ausführen 668
TensorFlow, NumPy-ähnliche Operationen
 andere Datenstrukturen 387
 Tensoren und NumPy 385
 Tensoren und Operationen 383
 Typumwandlungen 385
TensorFlow.js 381
TensorFlow-Cluster 712
TensorFlow-Datasets-(TFDS-)Projekt, TF Datasets (TFDS) 445
Term-Frequency × Inverse-Document-Frequency (TF-IDF) 443

Testdatensatz 31, 53, 90
Testen und Validieren 31
 Datendiskrepanz 33
 Hyperparameter anpassen 32
 Modellauswahl 32
Texterzeugung
 Modelle einsetzen 535
 sequenzielle Datensätze aufteilen 532
 Shakespeare-Text erzeugen 536
 Trainingsdatensatz erstellen 531
 Überblick 530
 zustandsbehaftete RNNs 537
Textmerkmale 67
TF Datasets (TFDS) 418
TF Functions
 Graphen 795–803
 Regeln 412
TF Transform (tf.Transform) 443
tf.keras 297, 366, 427
tf.summary (Paket) 321
TF.Text-Bibliothek 540
TF-Agents-Bibliothek
 Collect-Fahrer 658
 Deep-Q-Netze (DQNs) 652
 DQN-Agenten 654
 Installation 645
 Replay Buffer und Beobachter 655
 Trainingsarchitektur 650
 Trainingsmetriken 657
 Trainingsschleifen 662
 Übersicht 644
 Umgebungen 645
 Umgebungsspezifikationen 646
 Umgebungswrapper 647
TFRecord-Format
 komprimierte TFRecord-Dateien 429
 Listen von Listen per SequenceExampleProtobuf 433
 Protocol Buffer (Protobufs) 429
 Überblick 428
Theano 297
theoretisches Informationskriterium 269
thermisches Gleichgewicht 779
Threshold Logic Unit (TLU) 286
Tiefenradius 470
toarray() 69
Toleranz-Hyperparameter 164
TPUs (Tensor Processing Units) 380
Trägheit 245
Train-Dev-Set 33
Trainieren von Modellen 115
 Lernkurven beim 133

- lineare Regression 115, 116
- logistische Regression 144
- polynomielle Regression 116, 131
- Überblick 115
- Training/Serving Skew 444
- Trainingsdaten 4
 - irrelevante Merkmale 28
 - minderwertige 27
 - nicht repräsentative 26
 - Overfitting 28
- Trainingsdatensatz erzeugen 531
- Underfitting 30
- unzureichende Menge 24
- Trainingsdatenpunkt 4
- Trainingsdatensatz 4, 31, 57, 64, 74
 - Kostenfunktion von 146
- Trajektorien 650, 651
- Transfer Learning 326, 484
 - siehe Wiederverwenden vortrainierter Schichten
- transform() 66, 73
- Transformationen
 - affine Transformationen 607
 - verketten 419
- Transformer 66
- Transformer, eigene 70
- Transformer-Architektur 558
- Transposed Convolutional Layer 496
- Trefferquote 95
- Truncated Backpropagation Through Time 533
- Turing-Test 529
- t-verteiltes Stochastic Neighbor Embedding (t-SNE) 234
- Typumwandlungen 385

U

- überwachtes Lernen 9
- Umwelt, im Reinforcement Learning 612, 634
- unbalancierte Datensätze 94
- Uncertainty Sampling 257
- Underfitting 30, 75, 163
- Ungleichheit als Nebenbedingung 766
- univariate Regression 41
- univariate Zeitserien 507
- Unterstichproben 460
- unüberwachtes Lernen 11
 - Algorithmen zur Dimensionsreduktion 12
 - Algorithmen zur Visualisierung 12
 - Clustering 12, 238–262
 - Erkennen von Anomalien 13
 - gaußsches Mischverteilungsmodell 262–277
 - Lernen von Assoziationsregeln 12, 14

Überblick 237
unüberwachtes Vortrainieren 352, 579
Upsampling Layer 496

V

Validierungsdatensatz 32
Value Iteration 629
value_counts() 50
Varianz
 erhalten 221
 erklärte 225
 Gleichgewicht zwischen Bias und Varianz 136
Variational Autoencoder 589
Variational Inference 274
Variational Parameters 274
Vektoren, Spaltenvektoren 117
Verallgemeinerungsfehler 31
verborgene Schichten 290
 in MLPs 290
 Neuronen pro verborgener Schicht 327
Verketten von Transformationen 419
virtuelle GPU-Devices 696
visuelle Attention 556
visueller Cortex 450
Vokabular 436
Vorhersagen 94, 166, 178
Vorhersageservice
 auf GCP AI erstellen 677–681
 verwenden 682–685
Vortraining
 Greedy Layer-Wise Pretraining 352
 vortrainierte Embeddings wiederverwenden 545
 zum Transfer Learning 484
vorverarbeitete Merkmale 52
Vorverarbeitung 253, 434–443

W

Wahrscheinlichkeiten, schätzen 145, 181
Wahrscheinlichkeitsdichtefunktion (WDF) 238, 265
Wall Time 344
Warm-up-Phase 708
WaveNet 501, 525
Weighted-Moving-Average-Modell 510
White-Box-Modelle 180
Wide & Deep 310
Wiederherstellen von Modellen 316
Wiederverwenden vortrainierter Schichten 348
 Hilfsaufgaben 353
 unüberwachtes Vortrainieren 352
Word Embeddings 438

WordTree 493

Wurzel der mittleren quadratischen Abweichung (Root Mean Square Error, RMSE) 41

X

Xavier-Initialisierung 336

Xception (Extreme Inception) 477

Y

You Only Look Once (YOLO) 492

YouTube 281

Z

Zeitseriendaten 501

Deep RNNs 510

einfache RNNs 509

grundlegende Metriken 508

Überblick 507

weitere Modelle 510

Zero Padding 452, 457

Zero-Shot Learning (ZSL) 567

Zielfunktionen beim Trainieren 167

Zielgröße 52

Zielmodell 640

zufällige Initialisierung 121, 126, 128, 334

zufällige Patches und zufällige Subräume 198

zufällige Projektionen 233

Zufallssuche 80

Zugewinn an Information 183

Zustands-Aktions-Werte 629

zustandsbehaftete Metrik 393

Zustandswert, optimaler 628

Über den Autor

Aurélien Géron arbeitet als Consultant und Dozent für Machine Learning. Als ehemaliger Mitarbeiter von Google hat er von 2013 bis 2016 das YouTube-Team zur Klassifikation von Videos geleitet. Er war Gründer und CTO von einigen Firmen: Wifirst, einem führenden Wireless ISP in Frankreich, Polyconseil, einer Consulting-Firma, die auf Telekommunikation, Medien und Strategie fokussiert war, und Kiwisoft, einer Consulting-Firma rund um Machine Learning und Data Privacy.

Davor war er als Entwickler in verschiedenen Bereichen tätig: Finanzen (JP Morgan und Société Générale), Verteidigung (Department of Defense in Kanada) und Gesundheit (Bluttransfusionen). Er hat einige technische Bücher veröffentlicht (zu C++, Wi-Fi 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 die 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 ständig 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.

Thomas Demmig ist Diplom-Physiker. Er arbeitet bei der SAP SE als Entwickler im Bereich UI Framework und hat in den letzten Jahren viele Bücher aus den Bereichen Entwickeln und IT-Management übersetzt.

Kolophon

Das Tier auf dem Cover von *Praxiseinstieg Machine Learning mit Scikit-Learn, Keras 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 lang werden, meist erreichen sie aber ein Länge zwischen 19 und 24 Zentimetern.

Kleinasiatische Feuersalamander leben in subtropischen Strauch- und Waldregionen in der Nähe von 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 (Koboldkäpflingen) zu einer 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.

Die Umschlagillustration zu diesem Buch stammt von Karen Montgomery, sie basiert auf einem Stich aus *Wood's Illustrated Natural History*. Den Umschlagentwurf haben Karen Montgomery und Michael Oréal erstellt. Auf dem Cover verwenden wir die Schriften Gilroy Semibold 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.



Rezensieren

Sie dieses Buch

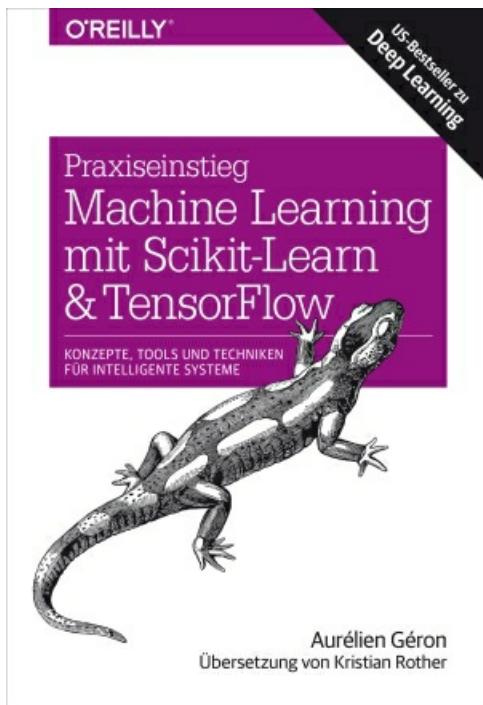
Senden

Sie uns Ihre Rezension
unter www.dpunkt.de/rez



Erhalten

Sie Ihr Wunschbuch aus
unserem Verlagsangebot



Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow

Géron, Aurélien
9783960101154
576 Seiten

[Titel jetzt kaufen und lesen](#)

Durchbrüche beim Deep Learning haben das maschinelle Lernen in den letzten Jahren eindrucksvoll vorangebracht. Inzwischen können sogar Programmierer, die kaum etwas über diese Technologie wissen, mit einfachen, effizienten Werkzeugen Machine-Learning-Programme implementieren. Dieses praxisorientierte Buch zeigt Ihnen wie. Mit konkreten Beispielen, einem Minimum an Theorie und zwei unmittelbar anwendbaren Python-Frameworks – Scikit-Learn und TensorFlow – verhilft Ihnen der Autor Aurélien Géron zu einem intuitiven Verständnis der Konzepte und Tools für das Entwickeln intelligenter Systeme. Sie lernen eine Vielzahl von Techniken kennen, beginnend mit einfacher linearer Regression bis hin zu neuronalen Netzen. Übungen zu jedem Kapitel helfen Ihnen, das Gelernte in die Praxis umzusetzen. Sie benötigen lediglich etwas Programmiererfahrung, um direkt zu starten. - Entdecken Sie Machine Learning, insbesondere neuronale Netze und das Deep Learning - Verwenden Sie Scikit-Learn, um ein Machine-Learning-Beispielprojekt vom Anfang bis zum Ende nachzuvollziehen - Erkunden Sie verschiedene trainierbare Modelle, darunter Support Vector Machines,

Entscheidungsbäume, Random Forests und Ensemble-Methoden - Nutzen Sie die Bibliothek TensorFlow, um neuronale Netze zu erstellen und zu trainieren - Lernen Sie Architekturen neuronaler Netze kennen, darunter Convolutional Nets, Recurrent Nets und Deep Reinforcement Learning - Eignen Sie sich Techniken zum Trainieren und Skalieren von neuronalen Netzen an - Wenden Sie Codebeispiele an, ohne exzessiv die Theorie von Machine Learning oder die Algorithmik durcharbeiten zu müssen

[Titel jetzt kaufen und lesen](#)

O'REILLY®



Neuronale Netze selbst programmieren

Rashid, Tariq
9783960101031
232 Seiten

[Titel jetzt kaufen und lesen](#)

Neuronale Netze sind Schlüsselemente des Deep Learning und der Künstlichen Intelligenz, die heute zu Erstaunlichem in der Lage sind. Sie sind Grundlage vieler Anwendungen im Alltag wie beispielsweise Spracherkennung, Gesichtserkennung auf Fotos oder die Umwandlung von Sprache in Text. Dennoch verstehen nur wenige, wie neuronale Netze tatsächlich funktionieren. Dieses Buch nimmt Sie mit auf eine unterhaltsame Reise, die mit ganz einfachen Ideen beginnt und Ihnen Schritt für Schritt zeigt, wie neuronale Netze arbeiten: - Zunächst lernen Sie die mathematischen Konzepte kennen, die den neuronalen Netzen zugrunde liegen. Dafür brauchen Sie keine tieferen Mathematikkenntnisse, denn alle mathematischen Ideen werden behutsam und mit vielen Illustrationen und Beispielen erläutert. Eine Kurzeinführung in die Analysis unterstützt Sie dabei. - Dann geht es in die Praxis: Nach einer Einführung in die populäre und leicht zu lernende Programmiersprache Python bauen Sie allmählich Ihr eigenes neuronales Netz mit Python auf. Sie bringen ihm bei, handgeschriebene Zahlen zu erkennen, bis es eine Performance wie ein professionell entwickeltes Netz erreicht. - Im nächsten Schritt tunen Sie die Leistung Ihres

neuronalen Netzes so weit, dass es eine Zahlenerkennung von 98 % erreicht – nur mit einfachen Ideen und simplem Code. Sie testen das Netz mit Ihrer eigenen Handschrift und werfen noch einen Blick in das mysteriöse Innere eines neuronalen Netzes. - Zum Schluss lassen Sie das neuronale Netz auf einem Raspberry Pi Zero laufen. Tariq Rashid erklärt diese schwierige Materie außergewöhnlich klar und verständlich, dadurch werden neuronale Netze für jeden Interessierten zugänglich und praktisch nachvollziehbar.

[Titel jetzt kaufen und lesen](#)



Android Tablets & Smartphones

Born, Günter
9783960103851
322 Seiten

[Titel jetzt kaufen und lesen](#)

Ohne Vorwissen Android Tablets oder Smartphones sicher bedienen

- Erfolgsautor Günter Born behandelt die typischen Fragen von Einsteigern und Senioren
- Alle Bedienungsfragen in verständlichen Schritt-für-Schritt-Anleitungen erklärt
- Komplett in Farbe, übersichtlich gestaltet und mit größerer Schrift
- Bestseller deckt jetzt Android 6 bis 9 ab und sensibilisiert für Sicherheitsfragen

Mit diesem praktischen Ratgeber finden Sie sich schnell zurecht und können Schritt für Schritt nachvollziehen, wie Sie Ihr Android-Handy oder -Tablet einrichten, wie Sie surfen, Fotos machen, Kurznachrichten und E-Mails verschicken, die Einsatzmöglichkeiten Ihres Geräts durch neue Apps erweitern und vieles mehr. Schwerpunkt ist die Bedienung von Smartphones und Tablet PCs mit den Android-Versionen 6 bis 9. Das Buch kann jedoch auch für Geräte mit älteren Android-Betriebssystemen genutzt werden, denn vieles ist hier sehr ähnlich.

[Titel jetzt kaufen und lesen](#)



Agile Spiele – kurz & gut

Bleß, Marc
9783960103196
190 Seiten

[Titel jetzt kaufen und lesen](#)

Spiele und Simulationen sind wichtige Hilfsmittel für Agile Coaches und Scrum Master und gehören in den Werkzeugkoffer eines jeden agilen Moderators. Dieses Buch beschreibt eine Auswahl von agilen Spielen, die sich in der Praxis besonders bewährt haben. Die Spiele veranschaulichen agile Prinzipien und Praktiken.

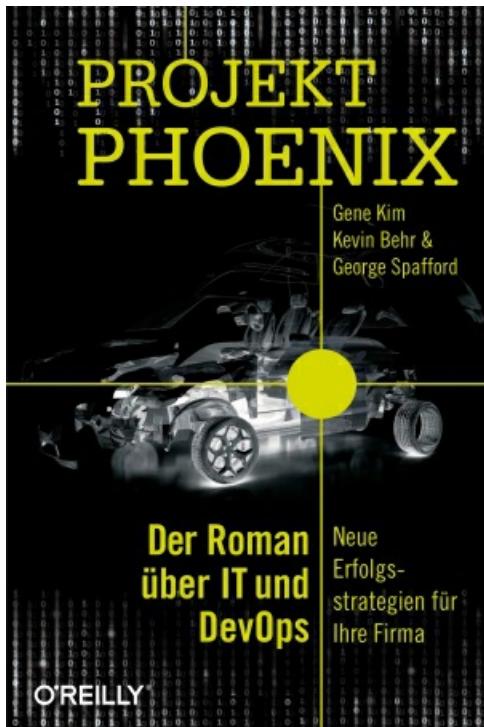
Marc Bleß und Dennis Wagner – beide seit vielen Jahren als Agile Coaches tätig – erläutern zunächst, was bei der Moderation von agilen Spielen grundsätzlich zu beachten ist und wann welches Spiel eingesetzt werden kann. Vorgestellt werden Spiele aus den Kategorien

- Vermittlung von agilen Prinzipien
- Simulation von agilen Praktiken
- Kommunikation

Beschrieben werden außerdem Spiele zur Eröffnung, zur Auflockerung und zum Abschluss von agilen Workshops und Trainings sowie Energizer für zwischendurch.

Der handliche Spiele-Werkzeugkoffer für alle, die Workshops zu agilen Methoden moderieren

[Titel jetzt kaufen und lesen](#)



Projekt Phoenix

Kim, Gene
9783960100676
352 Seiten

[Titel jetzt kaufen und lesen](#)

Bill ist IT-Manager bei Parts Unlimited. An einem Dienstagmorgen erhält er auf der Fahrt zur Arbeit einen Anruf seines CEO. Die neue IT-Initiative der Firma mit dem Codenamen Projekt Phoenix ist entscheidend für die Zukunft von Parts Unlimited, aber das Projekt hat Budget und Zeitplan massiv überzogen. Der CEO will, dass Bill direkt an ihn berichtet und das ganze Chaos in neunzig Tagen aufräumt, denn sonst wird Bills gesamte Abteilung outgesourct. Mit der Hilfe eines Vorstandsmitglieds und dessen mysteriöser Philosophie der Drei Wege wird Bill klar, dass IT-Arbeit mehr mit dem Fertigungsbereich in einer Fabrik zu tun hat als er sich je vorstellen konnte. Die Zeit drängt: Bill muss dafür sorgen, dass der Arbeitsfluss auch zwischen den Abteilungen deutlich besser läuft und das Business-Funktionalität zuverlässig bereitgestellt wird. Drei Koryphäen der DevOps-Bewegung liefern hier die rasante und unterhaltsame Story, in der sich jeder, der im IT-Bereich arbeitet, wiederfinden wird. Sie erfahren nicht nur, wie Sie Ihre eigene IT-Organisation verbessern können - nach der Lektüre dieses Buchs werden Sie IT auch nie wieder so sehen wie zuvor.

[Titel jetzt kaufen und lesen](#)