O'REILLY®

Deep Learning Kochbuch

PRAXISREZEPTE FÜR EINEN SCHNELLEN EINSTIEG

Douwe Osinga
Übersetzung von Marcus Fraaß
und Konstantin Mack

Deep Learning Kochbuch

Praxisrezepte für einen schnellen Einstieg

Douwe Osinga

Deutsche Übersetzung von
Marcus Fraaß und Konstantin Mack

O'REILLY®

Douwe Osinga

Lektorat: Alexandra Follenius
Übersetzung: Marcus Fraaß und Konstantin Mack
Fachgutachten: Kristian Rother
Korrektorat: Sibylle Feldmann, www.richtiger-text.de
Satz: III-satz, www.drei-satz.de
Herstellung: Stefanie Weidner
Umschlaggestaltung: Randy Comer, Michael Oréal, www.oreal.de
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

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

ISBN:
Print 978-3-96009-097-7
PDF 978-3-96010-264-9
ePub 978-3-96010-265-6
mobi 978-3-96010-266-3

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.
O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

1. Auflage

Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition of *Deep Learning Cookbook: Practical Recipes to Get Started Quickly*, ISBN 978-1-491-99584-6 © 2018 Douwe Osinga. 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.

5 4 3 2 1 0

Inhalt

Vorwort

Die Geschichte des Deep Learning

Die Wurzeln des aktuellen Deep-Learning-Booms reichen überraschend weit zurück – bis hinein in die 1950er-Jahre. Vage Vorstellungen von »intelligenten Maschinen« können sogar noch früher in Fiktion und Spekulation gefunden werden, aber in den 1950er- und 1960er-Jahren wurden die ersten »künstlichen neuronalen Netze« basierend auf einem extrem vereinfachten Modell biologischer Neuronen entwickelt. Unter diesen ersten Modellen stieß das Perzeptron-System von Frank Rosenblatt auf besonderes Interesse. Verbunden mit einer einfachen Kamera, konnte es lernen, verschiedene Objektarten zu unterscheiden. Obwohl die erste Version als Software auf einem IBM-Computer lief, wurden die folgenden Versionen komplett als Hardware implementiert.

Das Interesse am *mehrschichtigen Perzeptron-Modell* (MLP) setzte sich in den 1960er-Jahren fort. Dies änderte sich jedoch, als Marvin Minksy und Seymour Papert im Jahr 1969 ihr Buch *Perceptrons* (MIT Press) veröffentlichten. In diesem Buch bewiesen sie, dass lineare Perzeptron-Modelle das Verhalten einer nichtlinearen Funktion (XOR) nicht klassifizieren können. Trotz der Schwachstellen des Beweises (zu der Zeit der Publikation existierten bereits nicht-lineare Perzeptron-Modelle, und diese wurden auch von den Autoren erwähnt) läutete die Veröffentlichung des Buchs den Finanzierungseinbruch neuronaler Netze ein. Die Forschung erholte sich davon erst in den 1980er-Jahren mit dem Aufkommen einer neuen Forschergeneration.

Der steile Anstieg der Rechenleistung zusammen mit der Entwicklung der Backpropagation-Technik (seit den 1960er-Jahren in verschiedenen Formen bekannt, aber erst in den 1980er-Jahren allgemein angewandt) führte zu einem wieder aufkeimenden Interesse an neuronalen Netzen. Computer hatten nicht nur die benötigte Rechenleistung, um größere Netzwerke zu trainieren, sondern es gab nun auch Methoden, um tiefere Netze effizient zu trainieren. Die ersten Konvolutionsnetzwerke kombinierten diese Erkenntnisse mit einem Modell der Funktionsweise des Sehvermögens von Säugetieren. Dadurch entstanden erstmals Netzwerke, die

komplexe Bilder wie handgeschriebene Ziffern oder Gesichter erkennen konnten. Konvolutionsnetze erreichen dies, indem sie dasselbe »Teilnetz« auf verschiedene Stellen des Bilds anwenden und die Ergebnisse daraus zu abstrakteren Merkmalen zusammenfassen. In Kapitel 12 schauen wir uns diese Funktionsweise genauer an.

In den 1990er- und frühen 2000er-Jahren ging das Interesse an neuronalen Netzen wieder zurück, da »verständlichere« Modelle wie *Support Vector Machines* (SVMs) und Entscheidungsbäume an Beliebtheit gewannen. SVMs erwiesen sich für viele Datenquellen der damaligen Zeit als hervorragende Klassifikatoren, besonders in Verbindung mit von Menschen entwickelten Datenmerkmalen. In der Bildverarbeitung wurde die »Entwicklung von Merkmalen« populär. Dabei werden Merkmalsdetektoren erstellt, die kleine Elemente in einem Bild erkennen, die dann von Hand zu etwas Größerem vereint werden, das komplexere Formen erkennt. Später stellte sich heraus, dass Deep-Learning-Netze sehr ähnliche Merkmale erkennen und diese auch auf sehr ähnliche Weise kombinieren. In Kapitel 12 erforschen wir einige der inneren Abläufe dieser Modelle und visualisieren dabei das, was sie lernen.

Mit dem Aufkommen der Allzweckprogrammierung auf Grafikprozessoren (GPUs) in den späten 2000er-Jahren konnten neuronale Netzwerkarchitekturen große Fortschritte gegenüber anderen Modellen erzielen. GPUs enthalten Tausende kleine Prozessoren, die Milliarden von Operationen pro Sekunde parallel ausführen können. Ursprünglich für Computerspiele entwickelt, um komplexe 3-D-Szenen in Echtzeit darstellen zu können, stellte sich heraus, dass dieselbe Hardware verwendet werden kann, um neuronale Netze parallel zu trainieren und so Geschwindigkeitsverbesserungen um den Faktor 10 oder höher zu erreichen.

Außerdem ermöglichte das Internet nun den Zugriff auf riesige Trainingsdatensätze. Forscher, die zuvor Klassifikatoren mit Tausenden von Bildern trainierten, hatten jetzt Zugang zu Bildern im zwei- bis dreistelligen Millionenbereich. Verbunden mit größeren Netzwerkarchitekturen hatten neuronale Netze nun beste Erfolgsaussichten. Diese Dominanz hat sich in den folgenden Jahren dank verbesserter Techniken fortgesetzt und auch auf andere Anwendungsgebiete als die Bilderkennung ausgeweitet, etwa auf Übersetzung, Spracherkennung und Bilderzeugung.

Warum genau jetzt?

Während der rapide Rechenleistungsanstieg und bessere Techniken zu einer Interessenzunahme an neuronalen Netzen führten, sahen wir gleichzeitig große Fortschritte bei deren *Benutzerfreundlichkeit*. Insbesondere Deep-Learning-Frameworks wie TensorFlow, Theano und Torch ermöglichen auch Laien, komplexe neuronale Netze zu verwenden, um ihre eigenen Aufgaben im Bereich des Machine Learnings zu lösen. Dies hat eine Aufgabe, die früher nur mit monate- oder sogar jahrelanger Programmiererfahrung und Anstrengung bewältigt werden konnte (GPU-Kernel effizient zu programmieren, ist extrem schwierig!), zu etwas gemacht, das jeder an

einem Nachmittag (oder in ein paar Tagen) erledigen kann. Durch die gestiegene Benutzerfreundlichkeit ist auch die Zahl der Forscher, die an Deep-Learning-Aufgaben arbeiten können, sprunghaft gestiegen. Frameworks wie Keras ermöglichen durch ihr noch größeres Abstraktionsniveau, dass jeder, der über ordentliche Python-Kenntnisse und gewisse Hilfsmittel verfügt, interessante eigene Experimente durchführen kann, wie auch dieses Buch zeigt.

Ein zweites wichtiges Argument für »Warum genau jetzt?« ist, dass große Datensätze nun für jedermann erreichbar geworden sind. Klar, Facebook und Google haben immer noch einen Vorteil durch ihren Zugang zu Milliarden von Bildern, Kommentaren und sonstigen Dingen, aber mittlerweile kann man Datensätze mit Millionen von Elementen aus einer Vielzahl an Quellen beziehen. In Kapitel 1 sehen wir verschiedene Möglichkeiten, um an Datensätze zu gelangen. Über das gesamte Buch hinweg zeigt der Beispielcode jedes Kapitels normalerweise im ersten Rezept, wie man an den benötigten Trainingsdatensatz herankommt.

Mittlerweile haben auch private Unternehmen begonnen, riesige Datenmengen zu produzieren und zu sammeln, wodurch der gesamte Bereich des Deep Learnings plötzlich wirtschaftlich sehr interessant geworden ist. Ein Modell, das zwischen Hunden und Katzen unterscheiden kann, ist ja schön und gut, aber ein Modell, das unter Berücksichtigung historischer Verkaufsdaten den Umsatz um 15 % erhöht, kann den Unterschied zwischen Profit und Bankrott für ein Unternehmen ausmachen.

Voraussetzungen

Heutzutage gibt es eine große Auswahl an Plattformen, Technologien und Programmiersprachen für Deep Learning. In diesem Buch sind alle Codebeispiele in Python verfasst, und viele basieren auf dem ausgezeichneten Framework Keras. Die Codebeispiele sind auf GitHub als Python-Notebooks verfügbar. Zum Verständnis ist es hilfreich, über Grundlagenkenntnisse in den folgenden Gebieten zu verfügen:

Python
: Python 3 wird bevorzugt, aber Python 2.7 sollte auch funktionieren. Wir verwenden eine Vielzahl an Hilfsbibliotheken, die alle einfach per `pip` installiert werden können. Der Code ist im Allgemeinen recht einfach gehalten, sodass selbst Anfänger in der Lage sein sollten, dem Geschehen zu folgen.

Keras
: Die Schwerstarbeit im Machine Learning wird fast vollständig von Keras übernommen. Keras ist eine abstrakte Schnittstelle für die Deep-Learning-Frameworks TensorFlow und Theano. Keras ermöglicht es, neuronale Netze auf sehr gut lesbare Weise zu definieren. Der gesamte Code wurde mit TensorFlow getestet, sollte aber auch mit Theano funktionieren.

NumPy, SciPy, scikit-learn
: Diese umfangreichen und nützlichen Bibliotheken werden in vielen Rezepten nebenbei verwendet. Meistens sollte aus dem Kontext heraus klar sein, was passiert. Aber kurz in der entsprechenden Dokumentation nachzulesen, kann nicht schaden.

Jupyter Notebook
: Notebooks bieten eine tolle Möglichkeit, Code zu präsentieren. Sie ermöglichen eine Mischung aus Code, Ausgabe des Codes und Kommentaren – und dabei alles schön im Browser einsehbar.

Jedes Kapitel hat ein oder mehrere Notebooks, die den dazugehörigen Code enthalten. Der gedruckte Code im Buch lässt oft Details wie Importe aus, daher ist es sinnvoll, den Code von Git herunterzuladen und ein lokales Notebook zu starten. Mit folgendem Code können Sie die Notebooks herunterladen und in das neue Verzeichnis gehen:

```
git clone https://github.com/DOsinga/deep_learning_cookbook.git
cd deep_learning_cookbook
```

So können Sie eine virtuelle Umgebung für das Projekt erstellen:

```
python3 -m venv venv3
source venv3/bin/activate
```

und die benötigten Bibliotheken installieren:

```
pip install -r requirements.txt
```

Wenn Sie eine GPU besitzen und diese verwenden möchten, müssen Sie `tensorflow` deinstallieren und dafür `tensorflow-gpu` installieren, was Sie mit folgendem Befehl erreichen können:

```
pip uninstall tensorflow
pip install tensorflow-gpu
```

Außerdem benötigen Sie dafür noch eine kompatible GPU-Bibliothekskonfiguration, was bei der Einrichtung unter Umständen eine ziemlich lästige Angelegenheit sein kann.

Jetzt können Sie den IPython-Notebook-Server wie folgt aufrufen:

```
jupyter notebook
```

Hat alles funktioniert, sollte automatisch ein Webbrowser mit einer Übersicht der Notebooks geöffnet werden. Sie können gern mit dem Code herumexperimentieren; verwenden Sie einfach Git, um Ihre Änderungen rückgängig zu machen, falls Sie zur Ausgangssituation zurückkehren möchten:

```
git checkout <notebook_to_reset>.ipynb
```

Der erste Abschnitt jedes Kapitels erwähnt die für das jeweilige Kapitel relevanten Notebooks. Außerdem sind die Notebooks gemäß den Kapiteln nummeriert, es

sollte also einfach sein, sich zurechtzufinden. In dem Ordner mit den Notebooks sollten Sie noch drei weitere Verzeichnisse finden:

Data
: Dieser Ordner enthält die für die verschiedenen Notebooks benötigten Daten – meist Beispiele öffentlicher Datensätze oder Dinge, bei denen es zu umständlich wäre, sie selbst zu erstellen.

Generated
: Dieser Ordner wird zum Speichern von Zwischenergebnissen verwendet.

Zoo
: Dieser Ordner enthält einen Unterordner pro Kapitel mit den gespeicherten Modellen des jeweiligen Kapitels. Haben Sie keine Zeit, die Modelle selbst zu trainieren, können Sie sie trotzdem ausführen, indem Sie sie von hier laden.

Wegweiser durch dieses Buch

Kapitel 1 liefert detaillierte Informationen darüber, wie neuronale Netzwerke funktionieren, woher man Daten beziehen und wie man diese vorverarbeiten kann, um dem Netzwerk die Verarbeitung zu erleichtern. In Kapitel 2 dreht sich alles um Fehler und wie man sie beheben kann. Neuronale Netze sind in der Regel schwer zu debuggen, daher sind die Tipps und Tricks zur Fehlerbehebung aus diesem Kapitel sehr hilfreich, vor allem später für die eher projektorientierten Rezepte im Rest des Buchs. Falls Sie ungeduldig sind, können Sie dieses Kapitel auch zunächst überspringen und dann später darauf zurückkommen, wenn Sie irgendwo feststecken.

Die Kapitel 3 bis 15 handeln von den verschiedenen Medien, beginnend mit Textverarbeitung, gefolgt von Bildverarbeitung und schließlich Klangverarbeitung in Kapitel 15. Jedes Kapitel beschreibt ein Projekt, das in mehrere Rezepte aufgeteilt ist. Üblicherweise beginnt jedes Kapitel mit einem Rezept zur Datenakquise, gefolgt von einigen Rezepten, die auf das Projektziel ausgerichtet sind, und einem Rezept zur Datenvisualisierung.

Kapitel 16 dreht sich darum, wie Modelle in der Produktion eingesetzt werden können. In Python-Notebooks herumzuexperimentieren, ist super, aber letztendlich wollen wir unsere Ergebnisse mit echten Nutzern teilen und unsere Modelle auf Servern oder mobilen Endgeräten laufen lassen. In diesem Kapitel werden die verschiedenen Möglichkeiten dafür erläutert.

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.

`Feste Zeichenbreite`
: Wird für Programmlistings und Programmelemente wie Namen von Variablen, Funktionen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter verwendet.

`Feste Zeichenbreite, kursiv`
: Kennzeichnet Text, den der Nutzer je nach Kontext durch entsprechende Werte ersetzen muss.

Dieses Zeichen steht für einen Tipp oder eine Empfehlung.

Dieses Zeichen steht für einen allgemeinen Hinweis.

Verwenden von Codebeispielen

Jedes Kapitel enthält ein oder mehrere Python-Notebooks, die die jeweiligen Codebeispiele aus den Kapiteln enthalten. Sie können die Kapitel lesen, ohne den Code auszuführen, aber es macht mehr Spaß, beim Lesen nebenher mit den Notebooks zu arbeiten. Sie können den Code unter *https://github.com/DOsinga/deep_learning_cookbook* finden.

Führen Sie die folgenden Befehle in einer Shell aus, um den Beispielcode für die Rezepte zum Laufen zu bringen:

```
git clone https://github.com/DOsinga/deep_learning_cookbook.git
cd deep_learning_cookbook
python3 -m venv venv3
source venv3/bin/activate
pip install -r requirements.txt
jupyter notebook
```

Dieses Buch ist dazu da, Ihnen bei Ihren Aufgaben zu helfen. Der gesamte Code in den begleitenden Notebooks steht unter der großzügigen Apache-Lizenz 2.0.

Wir freuen uns über Zitate, verlangen diese aber nicht. Ein Zitat enthält Titel, Autor, Verlag und ISBN, beispielsweise: »*Deep Learning Kochbuch* von Douwe Osinga (O'Reilly). Copyright 2018 Douwe Osinga, 978-1-491-99584-6«.

Danksagungen

Von Wissenschaftlern, die ihre Ideen auf *https://arxiv.org* veröffentlichen, über Programmierer, die diese Ideen in Code umsetzen und den Code auf GitHub stellen, bis hin zu öffentlichen und privaten Institutionen, die Datensätze für jedermann

veröffentlichen – die Welt des Machine Learnings ist voll von Menschen und Organisationen, die Neueinsteiger willkommen heißen und diesen den Einstieg so leicht wie möglich machen. Open Data, Open Source, Open-Access-Publishing – ohne diese Kultur des Teilens gäbe es dieses Buch nicht.

Was für die in diesem Buch behandelten Ideen gilt, gilt umso mehr für den Code in diesem Buch. Ein Machine-Learning-Modell von Grund auf selbst zu schreiben, ist schwierig, daher basieren fast alle Modelle in den Notebooks auf Code von anderen Quellen. Dies ist der beste Lösungsansatz – finden Sie ein Modell, das etwas mit Ihrer Aufgabe Vergleichbares erledigt, und ändern Sie es Schritt für Schritt, während Sie bei jedem Schritt überprüfen, ob das Modell noch funktioniert.

Ein besonderer Dank gilt meinem Freund und Koautor dieses Buchs, Russell Power. Abgesehen davon, dass er beim Schreiben von Vorwort, Kapitel 6 und Kapitel 7 geholfen hat, war er maßgeblich an der Überprüfung der fachlichen Korrektheit sowohl des Buchs als auch des begleitenden Codes beteiligt. Darüber hinaus war er von unschätzbarem Wert als Diskussionspartner zu vielen Ideen, von denen es einige in das Buch geschafft haben.

Außerdem möchte ich meiner lieben Frau danken, die stets als erste Verteidigungslinie beim Korrekturlesen der neuen Kapitel diente. Sie besitzt die außergewöhnliche Fähigkeit, Fehler in einem Text zu erkennen, der weder in ihrer Muttersprache verfasst ist noch von einem Thema handelt, in dem sie Expertin ist.

Die Datei *requirements.in* enthält alle Open-Source-Pakete, die in diesem Buch verwendet werden. Ein herzliches Dankeschön geht an alle, die an diesen Projekten mitwirken. Dies gilt in besonderer Weise für Keras, da fast der gesamte Code auf diesem Framework basiert und sich oft an dessen Beispielen orientiert.

Ideen und Beispielcode dieser Pakete, aber auch viele Blogbeiträge haben zu diesem Buch beigetragen, insbesondere:

Kapitel 2, Fehlerbehebung
: Dieses Kapitel greift auf Ideen aus dem Blogbeitrag »37 Reasons Why Your Neural Network Is Not Working« (*http://bit.ly/2IDxljz*) von Slav Ivanov zurück.

Kapitel 3, Die Ähnlichkeit von Texten mithilfe von Worteinbettungen berechnen
: Der Dank hier gilt Google für das Veröffentlichen des Word2vec-Modells.

  Das Gensim-Projekt von Radim Řehůřek trägt viel zu diesem Kapitel bei, und Teile des Codes basieren auf Beispielen aus diesem großartigen Projekt.

Kapitel 5, Text im Stil eines Beispieltexts generieren
: Dieses Kapitel ist stark an den Blogbeitrag »The Unreasonable Effectiveness of Recurrent Neural Networks« (*http://karpathy.github.io/2015/05/21/rnn-effectiveness/*) von Andrej Karpathy angelehnt. Dieser Blogbeitrag hat mein Interesse an neuronalen Netzen wieder geweckt.

  Die Visualisierung ist von Motoki Wus Blogbeitrag »Visualizations of Recurrent Neural Networks« (*http://bit.ly/2s8uAvg*) inspiriert.

Kapitel 6, Übereinstimmende Fragen

Dieses Kapitel steht gewissermaßen in Verbindung mit dem Quora-Fragenpaare-Wettbewerb auf Kaggle (*https://www.kaggle.com/c/quora-question-pairs*).

Kapitel 8, Sequenz-zu-Sequenz-Mapping

Der Beispielcode stammt aus den Beispielen von Keras, wird aber auf einem etwas anderen Datensatz angewendet.

Kapitel 11, Mehrere Bildinhalte erkennen

Dieses Kapitel basiert auf dem Projekt *keras_frcnn* (*https://github.com/yhenon/keras-frcnn*) von Yann Henon.

Kapitel 12, Mit Bildstilen arbeiten

Dieses Kapitel ist an »How Convolutional Neural Networks See the World« (*http://bit.ly/2s4ORCf*) und natürlich Googles DeepDream (*https://github.com/google/deepdream/blob/master/dream.ipynb*) angelehnt.

Kapitel 13, Bilder mit Autoencodern erzeugen

Der Code und die Ideen basieren auf dem Blogbeitrag über Conditional Variational Autoencoder (*http://nnormandin.com/science/2017/07/01/cvae.html*) von Nicholas Normandin.

Kapitel 14, Piktogramme mithilfe von neuronalen Netzwerken erzeugen

Der Trainingscode für den Autoencoder auf Grundlage von Keras basiert auf dem Projekt DCGAN-Keras (*https://github.com/ctmakro/DCGAN-Keras/blob/master/lets_gan.py*) von Qin Yongliang.

Kapitel 15, Musik und Deep Learning

Dieses Kapitel wurde durch das Projekt *gtzan.keras* (*https://github.com/Hguimaraes/gtzan.keras*) von Heitor Guimarães inspiriert.

KAPITEL 1

Werkzeuge und Techniken

In diesem Kapitel werfen wir einen Blick auf die gängigen Werkzeuge und Techniken beim Deep Learning. Dieses Kapitel eignet sich gut, um sich einen Überblick über die Grundlagen zu verschaffen und sie bei Bedarf aufzufrischen.

Zu Beginn schauen wir uns verschiedene Arten neuronaler Netze an, die Ihnen im Verlauf dieses Buchs begegnen werden. Die meisten Rezepte, die wir Ihnen später vorstellen, sind sehr praxisorientiert und behandeln daher nur flüchtig, wie neuronale Netze aufgebaut sind.

Dann besprechen wir, woher wir unsere Daten bekommen. Technologie-Riesen wie Facebook und Google haben Zugang zu gigantischen Datenmengen für ihre Deep-Learning-Forschung, aber auch für uns gibt es genügend zugängliche Daten im Internet, um interessante Projekte durchzuführen. Die Rezepte in diesem Buch verwenden Daten aus einer Vielzahl unterschiedlicher Quellen.

Im nächsten Abschnitt geht es um die Vorverarbeitung von Daten – ein sehr wichtiger Bereich, der allerdings oft übersehen wird. Selbst wenn Sie die richtigen Einstellungen für das neuronale Netz vorgenommen haben und zusätzlich eine gute Datenqualität vorliegt, müssen Sie trotzdem sicherstellen, dass Sie dem Netzwerk Ihre Daten in einer Form zuführen, die für das Netzwerk so geeignet wie möglich ist. Denn Sie wollen es dem Netzwerk so einfach wie möglich machen, die Dinge zu lernen, die es lernen soll, ohne dass andere, irrelevante Informationen in den Daten das Netzwerk dabei stören.

1.1 Arten neuronaler Netze

In diesem Kapitel und auch über das gesamte Buch hinweg sprechen wir über *Netzwerke* und *Modelle*. Ein Netzwerk (oder auch neuronales Netz) besteht aus einer gewissen Anzahl verbundener *Schichten*. Man speist Daten auf der einen Seite ein, und die verarbeiteten Daten kommen auf der anderen Seite heraus. Jede Schicht vollzieht dabei eine mathematische Operation an den durchfließenden Daten und besitzt eine Reihe veränderlicher Parameter, die das genaue Verhalten der Schicht

bestimmen. In diesem Zusammenhang beziehen sich *Daten* auf einen *Tensor*, einen Vektor mit mehreren Dimensionen (üblicherweise zwei oder drei).

Eine vollständige Erläuterung der verschiedenen Arten von Schichten und der Mathematik hinter deren Operationen würde den Rahmen dieses Buchs sprengen. Die einfachste Art von Schicht, die vollständig verbundene Schicht, nimmt ihre Eingabe als Matrix, multipliziert diese mit einer weiteren Matrix, den sogenannten *Gewichten*, und addiert eine dritte Matrix, den sogenannten *Bias*. Auf jede Schicht folgt eine *Aktivierungsfunktion*, eine mathematische Funktion, die die Ausgabe einer Schicht auf die Eingabe der nächsten Schicht abbildet. Zum Beispiel gibt es eine einfache Aktivierungsfunktion namens ReLU, die alle positiven Werte weitergibt, aber alle negativen Werte auf null setzt.

Genau genommen bezieht sich der Begriff *Netzwerk* auf die Architektur, die Art und Weise, wie die verschiedenen Schichten miteinander verbunden sind, während ein *Modell* das Netzwerk und zusätzlich alle Variablen beschreibt, die das Verhalten des Modells bestimmen. Das Trainieren eines Modells verändert diese Variablen, um die Vorhersagen besser an die erwarteten Ausgaben anzupassen. In der Praxis werden diese beiden Begriffe jedoch oft synonym verwendet.

Die Begriffe »Deep Learning« und »neuronale Netze« umfassen tatsächlich eine Vielzahl von Modellen. Die meisten dieser Netzwerke haben gewisse Gemeinsamkeiten (beispielsweise verwenden beinahe alle Klassifikationsnetzwerke eine bestimmte *Verlustfunktion*). Obwohl die Anwendungsbereiche dieser Modelle sehr vielfältig sind, lassen sich fast alle Modelle in gewisse Kategorien unterteilen. Manche Modelle verwenden Elemente mehrerer Kategorien: Beispielsweise haben viele Netzwerke für Bildklassifikationen am Ende einen Bereich mit vollständig verbundenen Schichten, um die eigentliche Klassifikation durchzuführen.

Vollständig verbundene neuronale Netze

Vollständig verbundene neuronale Netze waren die erste Art von Netzwerken, die erforscht wurden, und beherrschten das Forschungsinteresse bis in die späten 1980er-Jahre. In einem vollständig verbundenen Netz wird jeder Ausgabewert als gewichtete Summe aller Eingabewerte berechnet. Aus diesem Verhalten ergibt sich der Begriff »vollständig verbunden«: Jeder Ausgabewert ist mit jedem Eingabewert verbunden. Als Formel lässt sich das wie folgt schreiben:

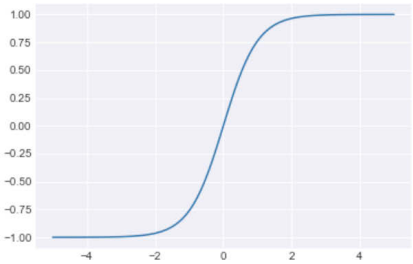
$$y_i = \Sigma_j W_{ij} x_j$$

Der Kürze halber stellen die meisten Forschungsartikel ein vollständig verbundenes Netz in Matrixschreibweise dar. Dafür multiplizieren wir den Eingabevektor mit der Gewichtsmatrix W, um den Ausgabevektor zu erhalten:

$$y = Wx$$

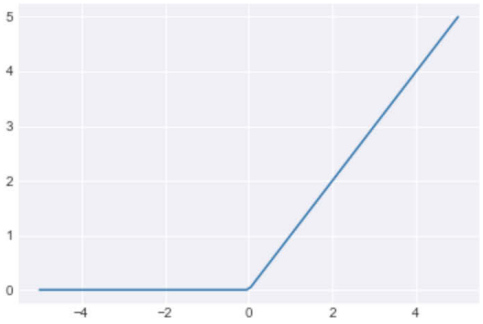
Da die Matrixmultiplikation eine lineare Rechenoperation ist, wäre ein Netzwerk, das ausschließlich Matrixmultiplikationen enthält, auf das Erlernen linearer Zusammenhänge beschränkt. Um unser Netzwerk jedoch aussagekräftiger zu machen, folgt auf die Matrixmultiplikation eine nicht-lineare Aktivierungsfunktion. Jede differenzierbare Funktion kann als Aktivierungsfunktion dienen, aber es gibt einige, die man sehr häufig antrifft. Der Tangens hyperbolicus, oder *tanh*, war bis vor Kurzem die am häufigsten verwendete Art von Aktivierungsfunktion und kann noch immer in einigen Modellen gefunden werden:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



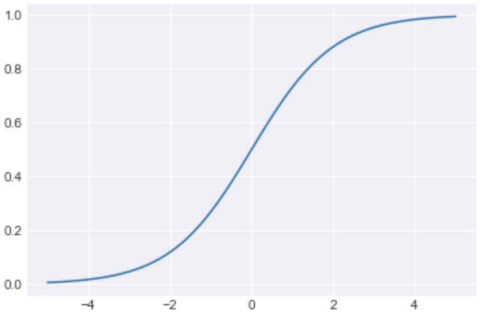
Das Problem der tanh-Funktion besteht darin, dass sie bei Eingabewerten, die weit von null entfernt sind, sehr »flach« ist. Das wiederum führt zu einem sehr geringen Gradienten, wodurch das Netzwerk sehr lange zum Lernen braucht. Seit Neuestem haben andere Aktivierungsfunktionen an Beliebtheit gewonnen. Eine der am häufigsten verwendeten Aktivierungsfunktionen ist die sogenannte Rectified Linear Unit, *ReLU* genannt:

$$relu(x) = \begin{cases} 0 & \text{if x} < 0 \\ x & \text{if x} \geq 0 \end{cases}$$



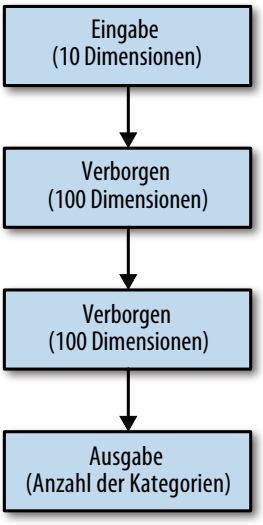
Außerdem verwenden viele neuronale Netze eine *sigmoide* Aktivierungsfunktion in der letzten Schicht des Netzwerks. Diese Art von Funktion gibt immer einen Wert zwischen 0 und 1 aus, wodurch die Ausgabewerte als Wahrscheinlichkeiten behandelt werden können:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$



Eine *Schicht* des Netzwerks entspricht einer Matrixmultiplikation gefolgt von einer Aktivierungsfunktion. Netzwerke können zum Teil aus über 100 Schichten bestehen, wobei die Anzahl der vollständig verbundenen Schichten in der Regel auf eine Handvoll beschränkt ist. Beim Lösen von Klassifikationsaufgaben (»Welche Katzenrasse ist auf diesem Bild zu sehen?«) wird die letzte Schicht als *Klassifikationsschicht* bezeichnet. Dabei hat diese stets so viele Ausgabewerte, wie es Kategorien zur Auswahl gibt.

Die Schichten in der Mitte des Netzwerks werden *verborgene Schichten* genannt, und die einzelnen Neuronen (Einheiten) einer verborgenen Schicht werden manchmal als *verborgene Neuronen* bezeichnet. Der Begriff »verborgen« rührt daher, dass diese Werte nicht direkt von außen als Ein- oder Ausgabewerte unseres Modells sichtbar sind. Die Anzahl der Ausgabewerte in diesen Schichten hängt ganz vom Modell ab:

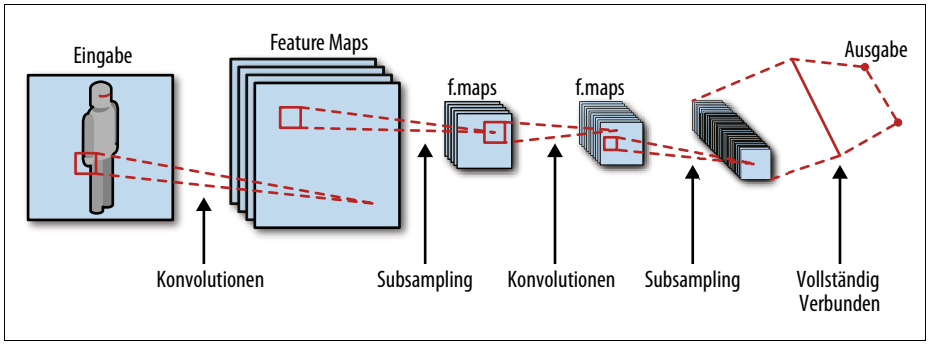


Es existieren zwar ein paar Faustregeln dazu, wie man Anzahl und Größe der verborgenen Schichten wählen sollte, jedoch gibt es generell keine bessere Verfahrensweise für die Auswahl der optimalen Konfiguration, als auszuprobieren.

Konvolutionsnetze

Zu Beginn wurde in der Forschung versucht, eine Vielzahl von Problemen mit vollständig verbundenen neuronalen Netzen zu lösen. Aber wenn wir mit Bildern als Eingabe arbeiten, sind vollständig verbundene Netzwerke nicht die beste Wahl. Bilder sind häufig sehr groß: Ein einzelnes Bild mit 256 × 256 Pixeln (eine gängige Auflösung für Klassifikationsaufgaben) hat 256 × 256 × 3 Eingabewerte (drei Farben für jeden Pixel). Wenn nun dieses Modell eine einzige verborgene Schicht mit 1.000 verborgenen Neuronen hat, besitzt diese Schicht allein fast 200 Millionen Parameter (trainierbare Werte)! Da Modelle zur Bilderkennung etliche Schichten benötigen, um bei der Klassifikation gut zu funktionieren, würden wir Milliarden von Parametern erhalten, wenn wir nur vollständig verbundene Schichten verwendeten.

Bei so vielen Parametern wäre es fast unmöglich, ein Overfitting unseres Modells zu vermeiden (Overfitting wird im nächsten Kapitel ausführlich behandelt; es beschreibt die Situation, in der ein Netzwerk lediglich die Trainingsdaten auswendig lernt, anstatt sinnvoll zu verallgemeinern). *Konvolutionsnetze* (*Convolutional Neural Networks*, CNNs) ermöglichen uns, Bildklassifikatoren mit übermenschlicher Performance mit weitaus weniger Parametern zu trainieren. CNNs erreichen das, indem sie den Sehprozess von Tieren und Menschen nachahmen:



Die Grundfunktion in einem CNN ist die *Konvolution* (Faltung). Anstatt eine Funktion auf das gesamte Eingabebild anzuwenden, scannt eine Konvolution schrittweise jeweils einen kleinen Teil des Bilds. Nach jedem solchen Scan wird ein *Kernel* angewendet (typischerweise eine Matrixmultiplikation gefolgt von einer Aktivierungsfunktion, genau wie in einem vollständig verbundenen Netzwerk). Einzelne Kernel werden auch oft als *Filter* bezeichnet. Nachdem wir den Kernel auf das gesamte Bild angewendet haben, bekommen wir als Ergebnis ein neues,

möglicherweise kleineres Bild zurück. Eine gängige Größe für Filter ist (3, 3). Würden wir nun beispielsweise 32 dieser Filter auf unser Eingabebild anwenden, bekämen wir 3 * 3 * 3 (Bildfarben) * 32 = 864 Parameter – deutlich weniger als bei einem vollständig verbundenen Netz!

Subsampling

Durch den Konvolutionsschritt konnten wir zwar die Anzahl der Parameter deutlich verringern, aber wir haben dadurch nun ein anderes Problem geschaffen. Jede Netzwerkschicht kann lediglich ein 3 × 3-Fenster des Bilds auf einmal »betrachten«, wie sollen wir in diesem Fall dann überhaupt Objekte erkennen können, die sich über das gesamte Bild erstrecken? Hier hilft das sogenannte *Subsampling*, das Konvolutionsnetze üblicherweise verwenden, um die Größe des Bilds im Verlauf des Netzwerks zu verringern. Zwei geläufige Subsampling-Methoden sind:

Strided Konvolution
: In einer strided Konvolution überspringen wir einfach ein oder mehrere Pixel beim Verschieben des Konvolutionsfilters über das Bild. Dadurch erhalten wir ein kleineres Ausgabebild. Wenn zum Beispiel unser Eingabebild 256 × 256 Pixel groß ist und wir beim Verschieben des Filters immer einen Pixel überspringen, hat unser Ausgabebild eine Größe von 128 × 128 (der Einfachheit halber ignorieren wir hier die Frage nach dem Auffüllen der Randumgebung des Bilds, auch Padding genannt). Diese Art von strided Downsampling findet man häufig in generativen Netzen (siehe den nächsten Abschnitt »Adversarial Networks und Autoencoder« auf Seite 9).

Pooling
: Anstatt Pixel bei der Konvolution zu überspringen, verwenden viele Netzwerke *Pooling-Schichten*, um die Anzahl der Eingabewerte zu verringern. Eine Pooling-Schicht ist gewissermaßen eine weitere Form der Konvolution, aber statt die Eingabewerte mit einer Matrix zu multiplizieren, wird hier ein Pooling-Operator verwendet, typischerweise der *max*- oder *average*-Operator. Max-Pooling ermittelt jeweils den größten Wert von jedem *Farbkanal* aus dem aktuell bearbeiteten Bildfenster. Average-Pooling hingegen nimmt den Durchschnittswert des Bildfensters. (Man kann sich darunter auch ein einfaches Verunschärfen des Eingabebilds vorstellen.)

Subsampling kann also als Methode verstanden werden, um das Abstraktionsniveau des Netzwerks zu erhöhen. Auf der untersten Ebene erkennen unsere Konvolutionen kleine, lokale Merkmale. Von diesen, nicht sehr tiefen Merkmalen gibt es viele. Mit jedem Pooling-Schritt erhöhen wir nun das Abstraktionsniveau. Die Anzahl der Merkmale sinkt, aber die Tiefe jedes Merkmals steigt. Diesen Prozess führen wir so lange fort, bis wir am Ende nur noch sehr wenige Merkmale mit sehr hohem Abstraktionsniveau haben. Diese können wir dann für die Vorhersage verwenden.

Vorhersage

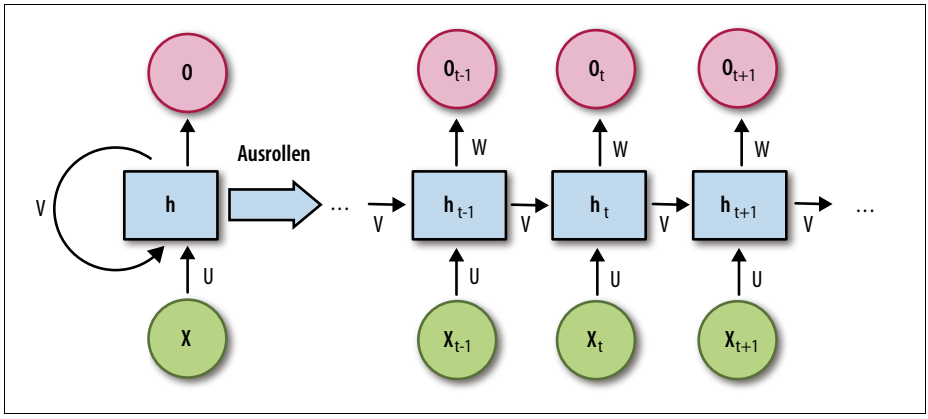
Nach einer Reihe von Konvolutions- und Pooling-Schichten folgen bei CNNs am Ende des Netzwerks ein oder zwei vollständig verbundene Schichten, um die Vorhersage zu erzeugen.

Rekurrente neuronale Netze

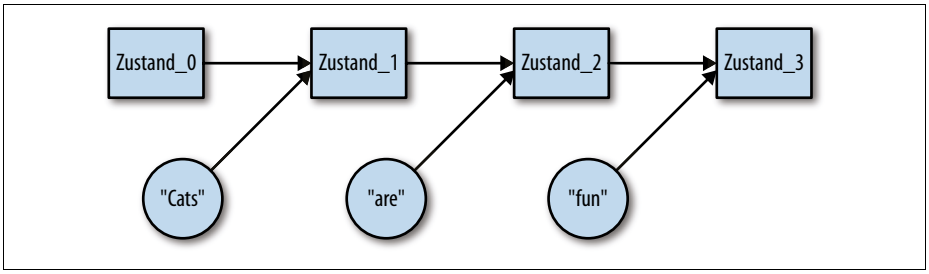
Rekurrente neuronale Netze (RNNs) sind konzeptionell den CNNs sehr ähnlich, aber strukturell sehr unterschiedlich. RNNs werden häufig bei sequenziellen Eingabedaten verwendet. Diese Art Eingabedaten findet sich in der Regel bei der Text- oder Sprachverarbeitung. Anstatt einen einzelnen Datensatz komplett zu verarbeiten (wie wir es mit einem CNN für ein Bild machen würden), können wir bei sequenziellen Aufgaben jeweils nur einen Teil der Aufgabe bearbeiten. Möchten wir zum Beispiel ein Netzwerk erstellen, das Textstücke im Stil von Shakespeare verfasst, würden wir als Eingabedaten natürlich die existierenden Stücke von Shakespeare nehmen:

```
Lear. Attend the lords of France and Burgundy, Gloucester.
Glou. I shall, my liege.
```

Das Netzwerk soll nun lernen, das nächste Wort des Textstücks für uns vorherzusagen. Dazu muss es sich den bisherigen Text »merken«. Rekurrente neuronale Netze geben uns genau dafür einen Mechanismus an die Hand. RNNs erlauben uns auch, Modelle zu bauen, die von selbst in der Lage sind, mit Eingabedaten unterschiedlicher Länge umzugehen (z. B. Sätze oder Teile eines Texts). Die einfachste Form eines RNN sieht wie folgt aus:



Aus konzeptioneller Sicht kann man sich dieses RNN wie ein sehr tiefes, vollständig verbundenes Netzwerk vorstellen, das wir »ausgerollt« haben. In diesem Modell benötigt jede Schicht des Netzwerks jetzt zwei Eingabewerte statt wie sonst einen:



Sie erinnern sich, dass wir anfangs in unserem vollständig verbundenen Netz folgender Matrixmultiplikation begegnet sind:

$$y = Wx$$

Der zweite Eingabewert lässt sich dieser Operation am einfachsten hinzufügen, indem man ihn mit dem verborgenen Zustand verknüpft:

$$hidden_i = W\{hidden_{i-1}|x\}$$

wobei das »|« hier für verknüpfen steht. Genau wie bei dem vollständig verbundenen Netz können wir nun das Ergebnis der Matrixmultiplikation in die Aktivierungsfunktion einsetzen, um den neuen Zustand zu berechnen:

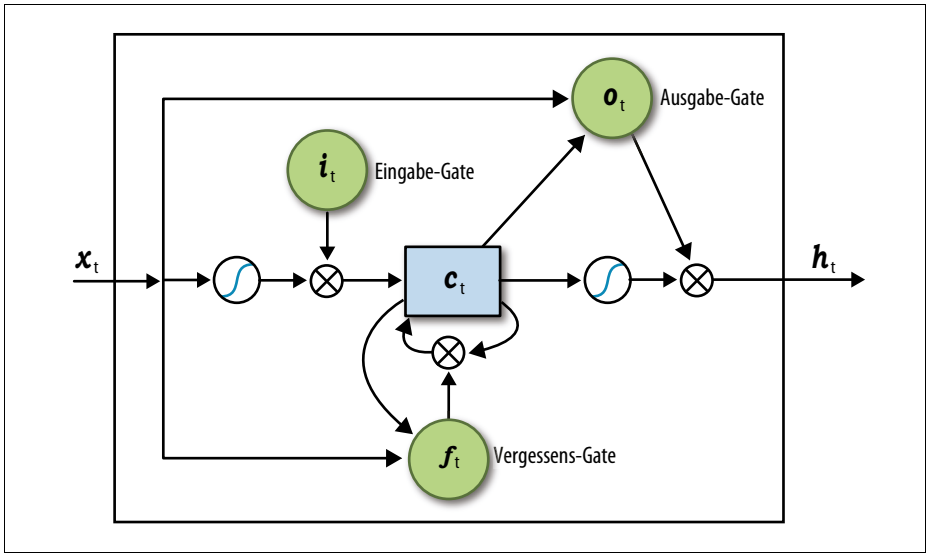
$$hidden_i = f\left(W\{hidden_{i-1}|x\}\right)$$

Mithilfe dieser Veranschaulichung unseres RNN können wir ebenfalls leicht nachvollziehen, wie es trainiert wird: Wir behandeln das RNN schlicht wie ein ausgerolltes, vollständig verbundenes Netzwerk und trainieren es ganz normal. Dieser Vorgang wird in der Literatur auch als *Backpropagation Through Time* (BPTT) bezeichnet. Bei sehr langen Eingabedaten ist es üblich, diese in kleinere Stücke aufzuteilen und jedes Stück unabhängig voneinander zum Trainieren zu verwenden. Diese Methode funktioniert zwar nicht bei jeder Aufgabenstellung, aber sie ist im Allgemeinen sicher und wird oft verwendet.

Schwindende Gradienten und LSTMs

Unser einfaches RNN neigt leider dazu, bei längeren Eingabesequenzen schlechter als erwünscht abzuschneiden. Der Grund dafür ist, dass es aufgrund seiner Struktur anfällig für das Problem »schwindender Gradienten« ist. Schwindende Gradienten resultieren aus der Tatsache, dass unser ausgerolltes Netzwerk sehr tief ist. Jedes Mal, wenn wir eine Aktivierungsfunktion durchlaufen, kann es passieren, dass als Ergebnis ein sehr kleiner Gradient weitergegeben wird (beispielsweise hat die ReLU-Aktivierungsfunktion den Gradienten null für jeden Eingabewert < 0). Sobald dies bei einem Neuron passiert, kann über dieses Neuron kein weiteres Training des Netzwerks weitergeleitet werden. Das führt zu einem immer geringer werdenden Trainingssignal, je weiter wir zurückgehen. Als Ergebnis lernt das Netzwerk nur extrem langsam oder sogar überhaupt nicht mehr.

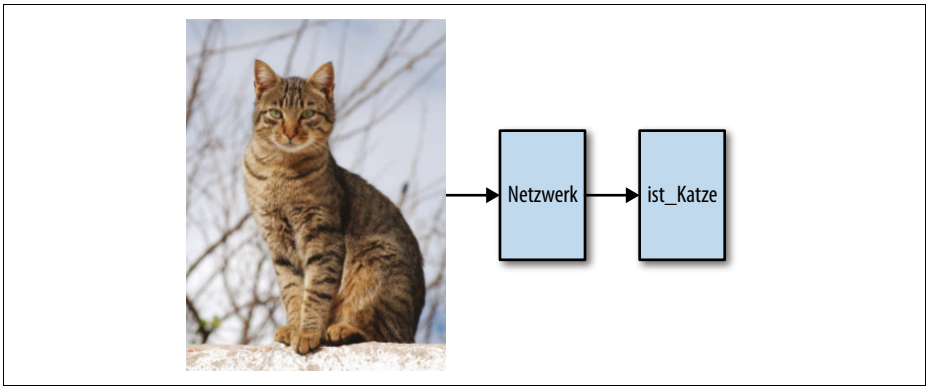
Um dieses Problem zu überwinden, wurde eine alternative Architektur für RNNs entwickelt. Das Ausrollen unseres Zustands über die Zeit hinweg wird dabei als Grundmodell beibehalten. Aber statt einer einfachen Matrixmultiplikation gefolgt von einer Aktivierungsfunktion wird nun ein etwas komplexerer Mechanismus verwendet, um den Zustand durch das Netzwerk weiterzugeben (Quelle: Wikipedia (*https://bit.ly/2HJL86P*)):



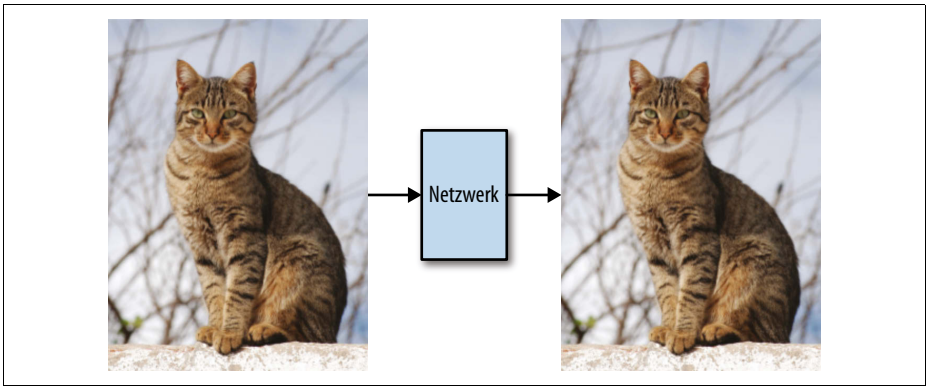
Das sogenannte *Long Short-Term Memory Network* (LSTM) ersetzt die einfache Matrixmultiplikation durch vier Matrixmultiplikationen und führt die Idee der *Gates* ein, die mit einem Vektor multipliziert werden. Der entscheidende Faktor, der es einem LSTM ermöglicht, effektiver zu lernen als ein einfaches RNN, ist, dass es in einem LSTM immer einen Weg von der Endausgabe (der finalen Vorhersage) zu jeder Schicht des Netzwerks gibt, die die Gradienten erhält. Die einzelnen Details dazu, wie genau ein LSTM dies erreicht, würden den Rahmen dieses Kapitels sprengen, es gibt jedoch online einige exzellente Tutorials (*http://colah.github.io/posts/2015-08-Understanding-LSTMs/*).

Adversarial Networks und Autoencoder

Für Adversarial Networks und Autoencoder benötigen wir keine neuen strukturellen Komponenten. Stattdessen verwenden sie die für die jeweilige Aufgabe am besten geeignete Struktur: Beispielsweise benutzt ein Adversarial Network oder ein Autoencoder beim Arbeiten mit Bildern Konvolutionen. Sie unterscheiden sich jedoch darin, wie sie trainiert werden. Die meisten neuronalen Netze werden trainiert, um auf Grundlage der Eingabedaten (ein Bild) eine Vorhersage zu treffen (ist das eine Katze?):



Autoencoder werden jedoch darauf trainiert, das Eingabebild wieder auszugeben:



Wozu könnte das gut sein? Wenn die verborgenen Schichten in der Mitte unseres Netzwerks eine Darstellung unseres Eingabebilds mit (deutlich) weniger Informationen als das Originalbild enthalten, aus der das Originalbild aber rekonstruiert werden kann, erzeugt das eine Art Komprimierung: Wir können ein Bild nehmen und es allein mit den Werten der verborgenen Schichten darstellen. Man kann sich das so vorstellen, dass wir das Originalbild mithilfe unseres Netzwerks in einen abstrakten Raum projizieren. Jeder Punkt in diesem Raum kann dann wieder zurück in ein Bild umgewandelt werden.

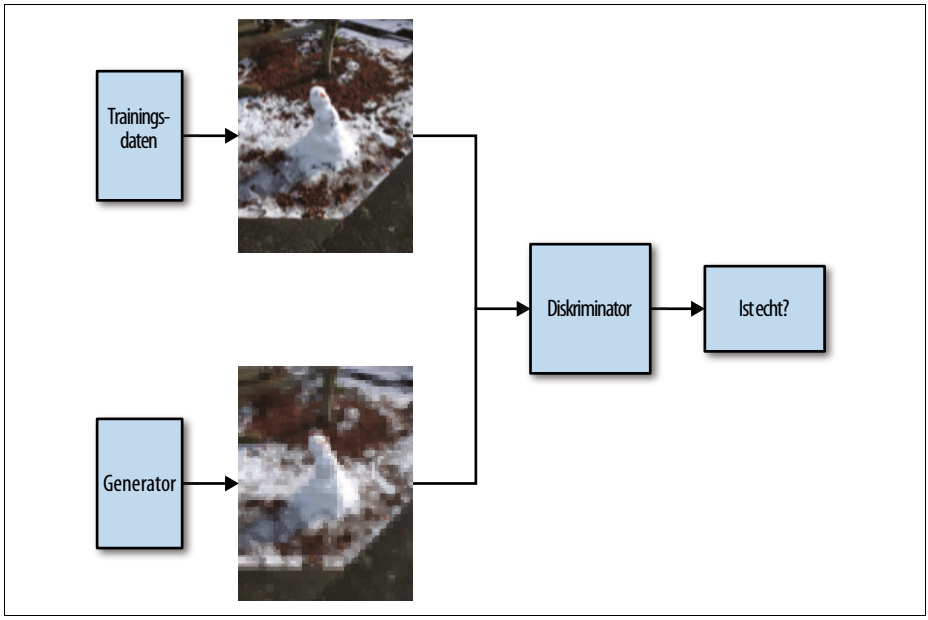
Autoencoder wurden erfolgreich auf kleine Bilder angewendet, aber der Trainingsmechanismus ist nicht ohne Weiteres auf größere Aufgaben anwendbar. Der abstrakte Raum in der Mitte, aus dem die Bilder rekonstruiert werden, ist in der Praxis nicht »dicht« genug. Daher führen viele Punkte aus diesem Raum nicht zu in sich stimmigen Bildern.

Wir sehen ein Beispielnetzwerk für einen Autoencoder in Kapitel 13.

Adversarial Networks sind ein neueres Modell, das tatsächlich realistische Bilder erzeugen kann. Dazu teilt das Netzwerk die Aufgabe in zwei Teile auf: ein Generator-

netz und ein Diskriminatornetz. Das Generatornetzwerk erzeugt ein Bild (oder Textstück). Das Diskriminatornetz versucht, festzustellen, ob das Eingabebild »echt« ist oder ob es vom Generatornetzwerk stammt.

Beim Trainieren des Adversarial Network trainieren wir die beiden Teilnetzwerke gleichzeitig:



Wir nehmen einige Bilder aus unserem Generatornetzwerk und schicken sie durch unser Diskriminatornetz. Das Generatornetzwerk wird für die Erzeugung von Bildern belohnt, die das Diskriminatornetz täuschen, also für Bilder, die als echt eingestuft werden. Das Diskriminatornetz muss versuchen, das jeweilige Bild richtig zu erkennen (es kann also nicht einfach immer sagen, das Bild sei eine Fälschung). Dadurch, dass die beiden Netze miteinander konkurrieren, kann dieses Verfahren zu einem generativen Netz führen, das qualitativ hochwertige und natürliche Bilder erzeugt. Kapitel 14 zeigt, wie wir *Generative Adversarial Networks* (GANs) verwenden können, um Icons zu erzeugen.

Fazit

Es gibt eine Vielzahl an Möglichkeiten, ein neuronales Netz zu konstruieren, wobei die Wahl natürlich hauptsächlich vom Zweck des Netzwerks abhängt. Die Entwicklung neuer Netzwerktypen ist ein fester Bestandteil der aktuellen Forschung. Selbst die Implementierung eines in einem Forschungspapier beschriebenen Netzes ist meist schwierig. In der Praxis ist es oft am einfachsten, ein Beispiel zu finden, das in etwa in dieselbe Richtung geht wie die eigene Idee, um es dann darauf aufbauend Stück für Stück zu ändern, bis es wirklich das tut, was man will.