O'REILLY®

Deutsche Ausgabe

Natural Language Processing mit PyTorch

Intelligente Sprachanwendungen mit Deep Learning erstellen

Delip Rao, Brian McMahan
Übersetzung von Frank Langenau

Natural Language Processing mit PyTorch

Intelligente Sprachanwendungen mit Deep Learning erstellen

Delip Rao und Brian McMahan

Deutsche Übersetzung von Frank Langenau

O'REILLY®

Delip Rao, Brian McMahan

Lektorat: Alexandra Follenius
Übersetzung: Frank Langenau
Korrektorat: Claudia Lötschert, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Karen Montgomery, 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 978-3-96009-118-9
PDF 978-3-96010-324-0
ePub 978-3-96010-325-7
mobi 978-3-96010-326-4

1. Auflage 2020

Wieblinger Weg 17
69123 Heidelberg

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.

Authorized German translation of the English edition of *Natural Language Processing with PyTorch: Build Intelligent Language Applications Using Deep Learning*, ISBN 978-1-492-97823-8 © 2019 Delip Rao and Brian McMahan. 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.

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.

5 4 3 2 1 0

Inhalt

Vorwort

Dieses Buch zielt darauf ab, Einsteiger in die Verarbeitung natürlicher Sprache (Natural Language Processing, NLP) und Deep Learning an einen Probiertisch zu bringen, auf dem die wichtigsten Themen in beiden Bereichen präsentiert werden. Beide Themenbereiche wachsen exponentiell. Da dieses Buch sowohl in Deep Learning als auch in NLP mit dem Schwerpunkt auf Implementierung einführt, nimmt es eine wichtige Mittelstellung ein. Als wir das Buch geschrieben haben, mussten wir die schwierige und manchmal unbequeme Entscheidung treffen, welches Material wir weglassen sollten. Wir hoffen, dass das Buch insbesondere Anfängern solide Grundkenntnisse vermittelt und einen Einblick in das gibt, was möglich ist. Maschinelles Lernen und speziell Deep Learning sind eher empirische, auf viel Erfahrung beruhende Disziplinen als intellektuelle Wissenschaften. Die großzügigen geschlossenen Beispiele in jedem Kapitel laden Sie ein, an dieser Erfahrung teilzuhaben.

Als wir mit der Arbeit am Buch begannen, ging es los mit PyTorch 0.2. Die Beispiele wurden mit jedem PyTorch-Update von 0.2 bis 0.4 überarbeitet. PyTorch 1.2 ist bereits veröffentlicht, wenn dieses Buch in deutscher Übersetzung erscheint. Die Codebeispiele im Buch sind PyTorch 0.4-konform und funktionieren auch mit PyTorch 1.2.

Ein paar Worte zum Stil des Buchs. An den meisten Stellen haben wir bewusst auf Mathematik verzichtet, nicht weil die Mathematik von Deep Learning besonders schwierig wäre (ist sie nicht), sondern weil sie in vielen Situationen vom Hauptziel dieses Buchs – nämlich den Einsteiger zu befähigen – ablenken würde. In gleicher Weise haben wir sowohl im Code als auch im Text Ausführlichkeit gegenüber Prägnanz bevorzugt. Fortgeschrittene Leser und erfahrene Programmierer sehen wahrscheinlich Möglichkeiten, den Code zu straffen, doch wir wollten so deutlich wie möglich sein, um ein möglichst breites Publikum zu erreichen.

Konventionen in diesem Buch

In diesem Buch werden folgende typografische Konventionen verwendet:

Kursiv

Kennzeichnet neue Begriffe, URLs, E-Mail-Adressen, Dateien und Dateierweiterungen.

`Schreibmaschinenschrift`

Wird in Programmlistings verwendet und im Fließtext für Programmelemente wie zum Beispiel Variablen- oder Funktionsnamen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter.

`Schreibmaschinenschrift fett`

Kennzeichnet Befehle oder andere Texte, die vom Benutzer buchstäblich eingegeben werden sollen.

`Schreibmaschinenschrift kursiv`

Zeigt Text, der ersetzt werden soll durch Werte, die der Benutzer bereitstellt, oder Werte, die sich aus dem Kontext ergeben.

Dieses Element kennzeichnet einen Tipp oder Vorschlag.

Dieses Element kennzeichnet einen allgemeinen Hinweis.

Dieses Element kennzeichnet eine Warnung oder einen Achtungshinweis.

Codebeispiele verwenden

Ergänzungsmaterial (Codebeispiele, Übungen usw.) steht zum Download unter *https://nlproc.info/PyTorchNLPBook/repo/* bereit.

Dieses Buch soll Ihnen bei Ihrer täglichen Arbeit helfen. Falls Beispielcode zum Buch angeboten wird, dürfen Sie ihn im Allgemeinen in Ihren Programmen und für Dokumentationen verwenden. Sie müssen uns nicht um Erlaubnis bitten, außer wenn Sie einen erheblichen Teil des Codes kopieren. Wenn Sie zum Beispiel ein Programm schreiben, das einige Codeblöcke aus diesem Buch verwendet, benötigen Sie keine Erlaubnis. Sollten Sie aber eine CD-ROM mit den Beispielen von

O'Reilly-Büchern verkaufen oder verteilen, ist eine Erlaubnis erforderlich. Wenn Sie eine Frage beantworten und dabei dieses Buch oder Beispielcode zu diesem Buch zitieren, brauchen Sie wiederum keine Erlaubnis. Aber wenn Sie erhebliche Teile des Beispielcodes aus diesem Buch in die Dokumentation Ihres Produkts einfließen lassen, ist eine Erlaubnis einzuholen.

Wir schätzen eine Quellenangabe, verlangen sie aber nicht. Eine Quellenangabe umfasst in der Regel Titel, Autor, Verlag und ISBN. Zum Beispiel: »*Natural Language Processing mit PyTorch* von Delip Rao und Brian McMahan (O'Reilly). Copyright 2020, ISBN 978-3-96009-118-9.«

Wenn Sie der Meinung sind, dass Sie die Codebeispiele in einer Weise verwenden, die über die oben erteilte Erlaubnis hinausgeht, kontaktieren Sie uns bitte unter *kommentar@oreilly.de*.

Danksagungen

Dieses Buch hat eine Entwicklung durchgemacht, wobei jede Version des Buchs anders als die vorherige Version aussah. An jeder Version waren verschiedene Personen (und sogar verschiedene Frameworks für Deep Learning) beteiligt.

Die Autoren möchten Goku Mohandas für sein anfängliches Mitwirken am Buch danken. Goku hat viel Energie in das Projekt gesteckt, bevor er es aus beruflichen Gründen leider verlassen musste. Gokus Enthusiasmus für PyTorch und seine positive Einstellung sind unübertroffen, und die Autoren haben ihn schmerzlich vermisst. Wir erwarten noch große Dinge von ihm!

Das Buch wäre nicht in bester technischer Form, wenn es nicht das freundliche und dennoch hochqualitative Feedback unserer technischen Gutachter, Liling Tan und Debasish Gosh, gegeben hätte. Liling hat seine Kompetenz bei der Entwicklung von Produkten mit modernem NLP beigetragen, während das wertvolle Feedback von Debasish aus der Sicht des Entwicklers stammt. Dankbar sind wir auch für die Unterstützung von Alfredo Canziani, Soumith Chintala und den vielen anderen tollen Menschen in den Entwicklerforen von PyTorch. Darüber hinaus haben wir von den täglichen, fruchtbaren NLP-Gesprächen unter der Twitter-Gemeinde *#nlproc* profitiert. Viele Erkenntnisse dieses Buchs sind sowohl dieser Community als auch unserer persönlichen Praxis zuzuschreiben.

Keinesfalls möchten wir versäumen, Jeff Bleiel für seine ausgezeichnete Unterstützung als unser Redakteur zu danken. Ohne seine Führung hatte dieses Buch nicht das Licht der Welt erblickt. Die Lektorate von Bob Russell und die Produktionsunterstützung durch Nan Barber haben dieses Manuskript von einem Rohentwurf zu einem druckfähigen Buch gemacht. Außerdem möchten wir Shannon Cutt für ihre Hilfe in den frühen Phasen des Buchs danken.

Ein Großteil des Materials im Buch entstand aus dem zweitägigen NLP-Training, das die Autoren auf den AI- und Strata-Konferenzen von O'Reilly abgehalten haben.

Wir möchten Ben Lorica, Jason Perdue und Sophia DeMartini dafür danken, dass sie mit uns an der Gestaltung der Trainings zusammengearbeitet haben.

Delip ist dankbar dafür, Brian McMahan als Co-Autor gewonnen zu haben. Brian tat alles, um die Entwicklung des Buchs zu unterstützen. Für beide waren es wertvolle Erfahrungen, sie haben die Freude und die Schmerzen an der Entwicklung geteilt! Delip möchte auch Ben Lorica bei O'Reilly dafür danken, dass er darauf beharrt hat, ein Buch über NLP zu schreiben.

Brian möchte Sara Manuel für ihre unendliche Unterstützung danken und Delip Rao dafür, dass er der Motor war, der dieses Buch zur Vollendung gebracht hat. Ohne seine nicht nachlassende Beharrlichkeit und Charakterstärke wäre dieses Buch nicht möglich gewesen.

KAPITEL 1

Einführung

Vertraute Namen wie Echo (Alexa), Siri und Google Translate haben mindestens eine Sache gemeinsam. Es sind alles Anwendungen, die von der *Verarbeitung natürlicher Sprache* (Natural Language Processing, NLP) abgeleitet sind, einem der zwei Hauptthemen dieses Buchs. NLP bezieht sich auf eine Reihe von Techniken, die statistische Methoden anwenden, und zwar mit oder ohne Erkenntnisse aus der Linguistik, um Text zu verstehen und damit praktische Aufgaben lösen zu können. Dieses »Verstehen« von Text wird hauptsächlich durch Transformieren von Texten in brauchbare rechentechnische *Darstellungen* abgeleitet, und zwar in diskrete oder stetige kombinatorische Strukturen wie zum Beispiel Vektoren oder Tensoren, Graphen und Bäume.

Das Lernen von Darstellungen, die für eine Aufgabe geeignet sind, aus Daten (in diesem Fall Text) ist Gegenstand des *maschinellen Lernens*. Die Anwendung maschinellen Lernens auf Textdaten kann auf eine mehr als 30-jährige Geschichte zurückblicken, doch in den letzten 10 Jahren[1] hat sich eine Reihe von Techniken des maschinellen Lernens, das sogenannte *Deep Learning*, ständig weiterentwickelt und erweist sich zunehmend als sehr effektiv für verschiedene Aufgaben der künstlichen Intelligenz (Artificial Intelligence, AI) in NLP, Sprache und Computervision. Deep Learning ist ein weiteres Hauptthema, das wir hier behandeln; somit ist dieses Buch eine Studie zu NLP und Deep Learning.

Literaturhinweise sind am Ende jedes Kapitels in diesem Buch aufgelistet.

Einfach ausgedrückt ermöglicht es Deep Learning, Darstellungen von Daten mithilfe einer Abstraktion – den sogenannten *Berechnungsgraphen* – und Techniken

1 Während die Geschichte der neuronalen Netze und NLP lang und umfangreich ist, wird Collobert und Weston (2008) oft die Pionierarbeit zugeschrieben, die Anwendung von Deep Learning auf NLP mit modernen Verfahren übernommen zu haben.

der numerischen Optimierung zu lernen. Der Erfolg von Deep Learning und von Berechnungsgraphen ist so groß, dass maßgebende Unternehmen wie Google, Facebook und Amazon Implementierungen von Berechnungsgraphen-Frameworks und Bibliotheken, die darauf aufbauen, veröffentlicht haben, um Forscher und Techniker darauf aufmerksam zu machen. In diesem Buch betrachten wir mit PyTorch ein zunehmend beliebter werdendes Python-basiertes Berechnungsgraphen-Framework, um Algorithmen für Deep Learning zu implementieren. In diesem Kapitel erläutern wir, was Berechnungsgraphen sind und warum wir PyTorch als Framework gewählt haben.

Das Gebiet des maschinellen Lernens und Deep Learning ist riesengroß. In diesem Kapitel und im größeren Teil dieses Buchs betrachten wir das sogenannte *überwachte Lernen*; das heißt Lernen mit benannten Trainingsbeispielen. Wir erklären das Paradigma des überwachten Lernens, das zur Grundlage für das Buch wird. Wenn Ihnen viele dieser Begriffe bislang nicht vertraut sind, dann sind Sie hier an der richtigen Stelle. Dieses Kapitel informiert nicht nur, sondern dringt auch tiefer in diese Konzepte ein. Das Gleiche gilt auch für spätere Kapitel. Wenn Sie bereits mit der Terminologie und den hier erwähnten Konzepten vertraut sind, sollten Sie trotzdem dabeibleiben, und zwar aus zwei Gründen: um ein gemeinsames Vokabular für den Rest des Buchs zu erarbeiten und alle Lücken zu füllen, die für das Verstehen zukünftiger Kapitel erforderlich sind.

Dieses Kapitel hat sich zum Ziel gesetzt, dass Sie

- ein klares Verständnis für das Paradigma des überwachten Lernens bekommen, die Terminologie verstehen und ein konzeptionelles Framework entwickeln, um Lernaufgaben für zukünftige Kapitel anzugehen.
- lernen, wie Eingaben für die Lernaufgaben zu codieren sind.
- verstehen, was Berechnungsgraphen sind.
- die Grundlagen von PyTorch meistern.

Los geht's!

Das Paradigma des überwachten Lernens

Überwachung im maschinellen Lernen, oder *überwachtes Lernen*, bezieht sich auf Fälle, in denen die Grundwahrheit für die *Ziele* (das, was vorhergesagt wird) für die *Beobachtungen* verfügbar ist. So ist zum Beispiel in der Dokumentklassifizierung das Ziel eine kategorische Bezeichnung[2], und die Beobachtung ist ein Dokument. In der maschinellen Übersetzung ist die Beobachtung ein Satz in der einen Sprache und das Ziel ein Satz in einer anderen Sprache. Mit diesem Verständnis von den Eingabedaten veranschaulicht Abbildung 1-1 das Paradigma des überwachten Lernens.

2 Eine kategorische Variable ist eine Variable, die einen festen Satz von Werten annimmt; zum Beispiel `{TRUE, FALSE}`, `{VERB, NOUN, ADJECTIVE ...}` usw.

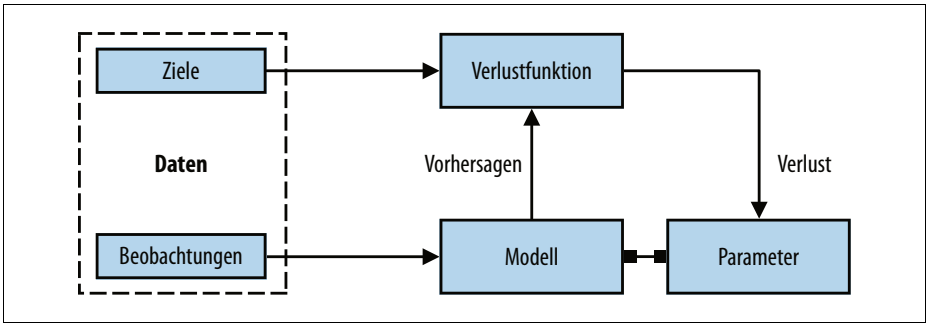

Abbildung 1-1: Das Paradigma des überwachten Lernens, ein konzeptionelles Framework für das Lernen von benannten Eingabedaten

Entsprechend der Darstellung in Abbildung 1-1 lässt sich das Paradigma des überwachten Lernens in sechs Hauptkonzepte gliedern:

Beobachtungen
: Beobachtungen sind Elemente, über die wir etwas vorhersagen möchten. Wir kennzeichnen Beobachtungen mit x. Manchmal verweisen wir auf die Beobachtungen auch als *Eingaben*.

Ziele
: Ziele sind Benennungen, die einer Beobachtung entsprechen. Normalerweise sind es die Dinge, die vorhergesagt werden. Entsprechend den Standardnotationen im maschinellen Lernen/Deep Learning verweisen wir darauf mit y. Für diese Benennung wird manchmal auch der Begriff *Grundwahrheit* verwendet.

Modell
: Ein Modell ist ein mathematischer Ausdruck oder eine Funktion, der/die eine Beobachtung x übernimmt und den Wert ihres Ziellabels vorhersagt.

Parameter
: Manchmal auch als Gewichte bezeichnet, parametrisieren sie das Modell. Es hat sich eingebürgert, die Notation w (für engl. weights – Gewichte) oder $\hat{w}$ zu verwenden.

Vorhersagen
: Vorhersagen – auch *Schätzungen* genannt – sind die Werte der Ziele, die das Modell anhand der Beobachtungen erraten soll. Wir kennzeichnen sie mit einer »Dach«-Notation. Die Vorhersage eines Ziels y wird also mit $\hat{y}$ bezeichnet.

Verlustfunktion
: Eine Verlustfunktion ist eine Funktion, die vergleicht, wie weit eine Vorhersage von ihrem Ziel für Beobachtungen in den Trainingsdaten entfernt ist. Für ein Ziel und seine Vorhersage weist die Verlustfunktion einen skalaren realen Wert zu, den sogenannten *Verlust*. Je geringer der Verlustwert ist, desto besser sagt das Modell das Ziel voraus. Die Verlustfunktion bezeichnen wir mit L.

Obwohl man das Thema nicht unbedingt mathematisch-formal angehen muss, um in der Modellierung von NLP/Deep Learning produktiv zu sein oder dieses Buch zu schreiben, kommen wir auf das Paradigma des überwachten Lernens zurück, um die Leser, die neu auf diesem Gebiet sind, mit der Standardterminologie bekannt zu machen. Somit können sie sich mit der Notation und dem Schreibstil vertraut machen, wie er in Forschungsbeiträgen zum Beispiel auf arXiv anzutreffen ist.

Betrachten wir ein Dataset $D = \{X_i, y_i\}_{i=1}^{n}$ mit n Beispielen. Für dieses Dataset möchten wir eine Funktion (ein Modell) f lernen, das durch Gewichte w parametrisiert ist. Wir treffen also eine Annahme über die Struktur von f, und anhand dieser Struktur werden die gelernten Werte des Gewichts w das Modell vollständig charakterisieren. Für eine gegebene Eingabe X sagt das Modell $\hat{y}$ als Ziel voraus:

$$\hat{y} = f(X,w)$$

Training mit (stochastischem) Gradientenabstieg

Überwachtes Lernen verfolgt das Ziel, Werte der Parameter auszuwählen, die die Verlustfunktion für ein gegebenes Dataset minimieren. Mit anderen Worten ist dies äquivalent dem Suchen von Wurzeln einer Gleichung. Wir wissen, dass *Gradientenabstieg* eine gebräuchliche Technik ist, um die Wurzeln einer Gleichung zu ermitteln. Wie bereits erwähnt, raten wir beim herkömmlichen Gradientenabstieg bestimmte Anfangswerte für die Wurzeln (Parameter) und aktualisieren die Parameter iterativ, bis die Zielfunktion (Verlustfunktion) einen Wert ergibt, der unterhalb eines akzeptablen Schwellenwerts (auch als Konvergenzkriterium bezeichnet) liegt.

Für große Datasets ist die Implementierung des herkömmlichen Gradientenabstiegs über dem gesamten Dataset normalerweise unmöglich infolge Speicherbeschränkungen und sehr langsam aufgrund des Berechnungsaufwands. Stattdessen wird gewöhnlich eine Approximation für den Gradientenabstieg bemüht, der sogenannte *Stochastische Gradientenanstieg* (SGD). Im stochastischen Fall wird ein Datenpunkt oder eine Teilmenge der Datenpunkte zufällig ausgewählt, und der Gradient wird für diese Teilmenge berechnet. Wird ein einzelner Datenpunkt verwendet, nennt man diesen Ansatz reines SGD, und wenn eine Teilmenge von (mehreren) Datenpunkten verwendet wird, sprechen wir von einem *Mini-Batch-SGD*. Oft lässt man die Wörter »rein« und »Mini-Batch« weg, wenn der verwendete Ansatz klar aus dem Kontext hervorgeht. In der Praxis verwendet man reinen SGD nur selten, weil dieses Verfahren aufgrund von verrauschten Aktualisierungen in einer sehr langsamen Konvergenz resultiert. Es gibt verschiedene Varianten des allgemeinen SGD-Algorithmus, die alle auf schnellere Konvergenz abzielen. In späteren Kapiteln untersuchen wir einige dieser Varianten in Verbindung damit, wie die Gradienten beim Aktualisieren der Parameter verwendet werden. Dieser Vorgang der iterativen Parameteraktualisierung wird *Backpropagation* genannt. Jeder Schritt (auch als Epoche bezeichnet) der Backpropagation besteht aus einem *Forward-Pass* (Vorwärtspass) und einem *Backward-Pass* (Rückwärtspass). Der Vorwärtspass aktualisiert die Parameter mit dem Gradienten des Verlusts.

Beim überwachten Lernen kennen wir für die Trainingsbeispiele das wahre Ziel y für eine Beobachtung. Der Verlust für diese Instanz wird demnach $L(y, \hat{y})$ sein. Überwachtes Lernen wird dann zu einem Prozess, die optimalen Parameter/Gewichte w zu ermitteln, die den kumulativen Verlust für alle n Beispiele minimieren.

Beachten Sie, dass hier bis jetzt nichts spezifisch für Deep Learning oder neuronale Netze ist.[3] Die Richtungen der Pfeile in Abbildung 1-1 zeigen den »Fluss« der Daten beim Trainieren des Systems an. Mehr über das Training und zum Konzept des »Flusses« kommt im Abschnitt »Berechnungsgraphen« auf Seite 10 zur Sprache, doch zuerst werfen wir einen Blick darauf, wie wir unsere Eingaben und Ziele in NLP-Problemen numerisch darstellen können, damit wir Modelle trainieren und Ausgaben vorhersagen können.

Beobachtung und Zielcodierung

Wir müssen die Beobachtungen (Text) numerisch darstellen, um sie in Verbindung mit Algorithmen des maschinellen Lernens verwenden zu können. In Abbildung 1-2 ist dies grafisch dargestellt.

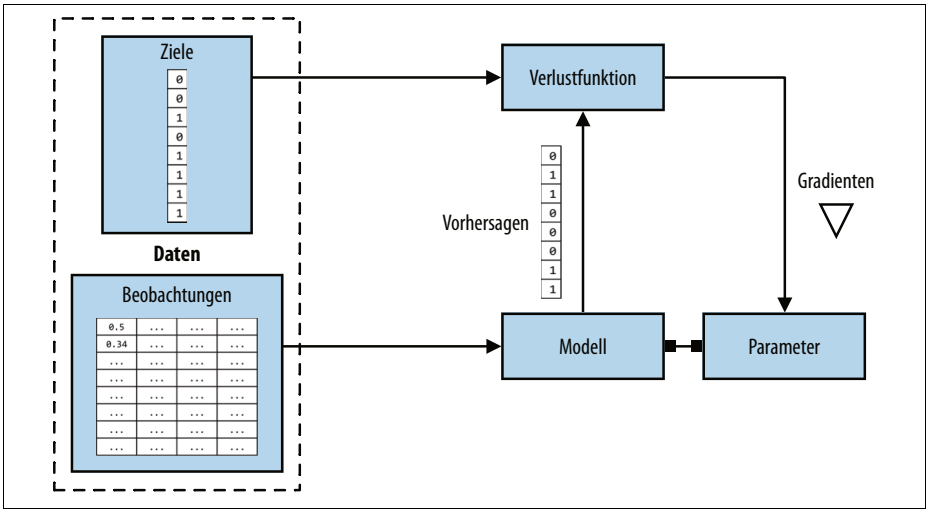

Abbildung 1-2: Beobachtung und Zielcodierung: Die Ziele und Beobachtungen von Abbildung 1-1 werden numerisch als Vektoren oder Tensoren dargestellt. Dies zusammen bezeichnet man als »Eingabecodierung«.

Text lässt sich in einfacher Weise als numerischer Vektor darstellen. Es gibt unzählige Arten, diese Zuordnung/Darstellung durchzuführen. Tatsächlich ist ein großer

3 Deep Learning unterscheidet sich von herkömmlichen neuronalen Netzen, wie sie in der Literatur vor 2006 diskutiert wurden, in dem Sinne, dass dieser Begriff auf ein wachsendes Repertoire von Techniken verweist, die eine höhere Zuverlässigkeit ermöglicht haben, indem mehr Schichten in das Netz hinzugefügt wurden. Weshalb dies wichtig ist, untersuchen wir in den Kapiteln 3 und 4.

Teil dieses Buchs dem Lernen solcher Darstellungen gewidmet. Allerdings beginnen wir mit einigen einfachen zahlenmäßigen Darstellungen, die auf Heuristiken basieren. Trotz ihrer Einfachheit sind sie an sich schon unglaublich leistungsfähig und können als Ausgangspunkt für das Lernen umfangreicherer Darstellungen dienen. Alle diese zahlenmäßigen Darstellungen beginnen mit einem Vektor fester Dimension.

1-aus-n-Darstellung

Wie der Name schon andeutet, beginnt die 1-aus-n-Darstellung (engl. One-Hot-Representation) mit einem Vektor aus Nullen und setzt den korrespondierenden Eintrag im Vektor auf 1, wenn das Wort im Satz oder Dokument enthalten ist. Betrachten Sie die folgenden beiden Sätze:

```
Time flies like an arrow.
Fruit flies like a banana.
```

Zerlegt man die Sätze in Token, wobei die Satzzeichen ignoriert werden, und behandelt jedes Wort als kleingeschrieben, bekommt man ein Vokabular der Größe 8: {`time`, `fruit`, `flies`, `like`, `a`, `an`, `arrow`, `banana`}. Somit können wir jedes Wort in einem achtdimensionalen 1-aus-n-Vektor darstellen. In diesem Buch verwenden wir 1_w, in der Bedeutung einer 1-aus-n-Darstellung für ein Token/Wort w.

Die »zusammengeklappte« (engl. collapsed) oder kompakte 1-aus-n-Darstellung für eine Phrase, einen Satz oder ein Dokument ist einfach ein logisches OR der 1-aus-n-Darstellungen seiner einzelnen Wörter. Mit der in Abbildung 1-3 gezeigten Codierung wird die 1-aus-n-Darstellung für die Phrase »like a banana« eine 3x8-Matrix sein, wobei die Spalten die achtdimensionalen 1-aus-n-Vektoren sind. Üblich ist auch eine »zusammengeklappte« oder binäre Codierung, bei der der Text/die Phrase durch einen Vektor von der Länge des Vokabulars dargestellt wird, wobei Nullen und Einsen die Ab- bzw. Anwesenheit eines Worts kennzeichnen. Die binäre Codierung für »like a banana« würde dann lauten: [0, 0, 0, 1, 1, 0, 0, 1].

Wenn Sie an dieser Stelle erschrocken sind, dass wir die beiden verschiedenen Bedeutungen (oder Sinnzusammenhänge) von »flies« zusammengefasst haben, Gratulation, aufmerksamer Leser! Die Sprache ist voll von Mehrdeutigkeiten, doch auch wenn wir furchtbar vereinfachende Annahmen treffen, können wir nützliche Lösungen aufbauen. Es ist möglich, sinnspezifische Darstellungen zu lernen, doch so weit sind wir jetzt noch nicht.

Obwohl wir in diesem Buch kaum etwas anderes als 1-aus-n-Darstellungen für die Eingaben verwenden, führen wir nun die Darstellungen *Term-Frequency* (TF) und *Term-Frequency-Inverse-Document-Frequency* (TF-IDF)[4] ein. Das geschieht auf-

4 TF (Term-Frequency) – Vorkommenshäufigkeit von Termen; IDF (Inverse-Document-Frequency) – inverse Dokumenthäufigkeit (siehe Wikipedia unter *https://de.wikipedia.org/wiki/Tf-idf-Maß*)

grund ihrer Beliebtheit in NLP, aus historischen Gründen und der Vollständigkeit wegen. Diese Darstellungen können im Information Retrieval auf eine lange Geschichte verweisen und werden selbst heute noch in NLP-Produktionssystemen aktiv eingesetzt.

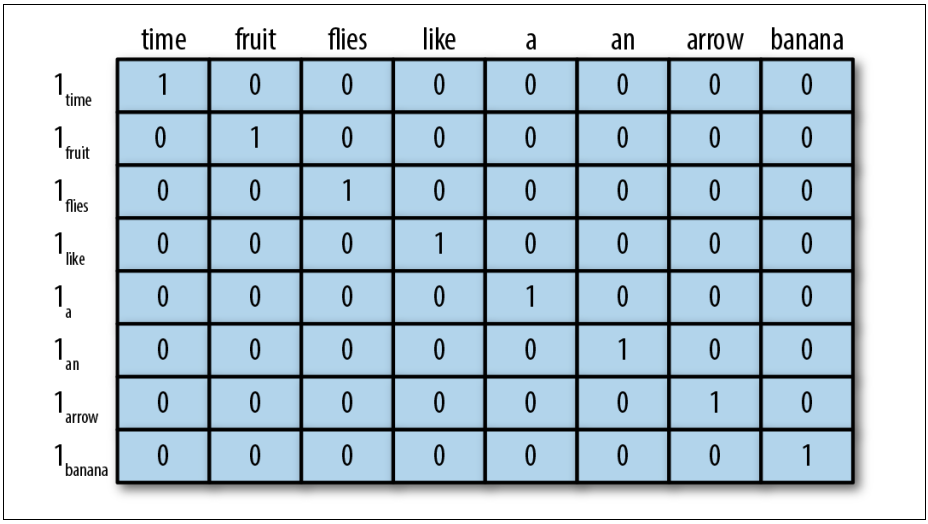

Abbildung 1-3: 1-aus-n-Darstellung für die Codierung der Sätze »Time flies like an arrow« und »Fruit flies like a banana«.

TF-Darstellung

Die TF-Darstellung einer Phrase, eines Satzes oder eines Dokuments ist einfach die Summe der 1-aus-n-Darstellungen seiner/ihrer einzelnen Wörter. Um mit unseren absurden Beispielen fortzufahren, wobei wir die oben erwähnte 1-aus-n-Codierung verwenden, hat der Satz »Fruit flies like time flies a fruit« die folgende TF-Darstellung: `[1, 2, 2, 1, 1, 0, 0, 0]`. Beachten Sie, dass jeder Eintrag angibt, wie oft das entsprechende Wort im Satz (Korpus) erscheint. Die Vorkommenshäufigkeit eines Worts *w* bezeichnen wir mit *TF(w)*.

Beispiel 1-1: Eine »kompakte« 1-aus-n- oder binäre Darstellung mit scikit-learn generieren

```
from sklearn.feature_extraction.text import CountVectorizer
import seaborn as sns

corpus = ['Time flies like an arrow.',
          'Fruit flies like a banana.']
one_hot_vectorizer = CountVectorizer(binary=True)
one_hot = one_hot_vectorizer.fit_transform(corpus).toarray()
sns.heatmap(one_hot, annot=True,
            cbar=False, xticklabels=vocab,
            yticklabels=['Sentence 1','Sentence 2'])
```

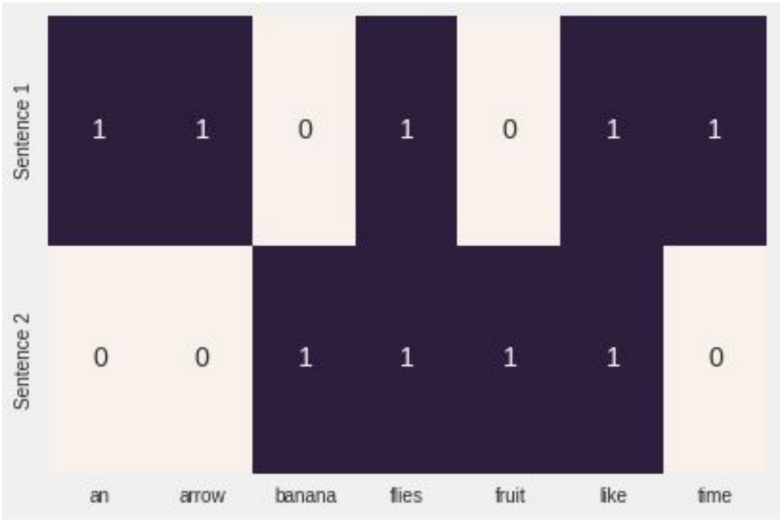

Abbildung 1-4: Die kompakte 1-aus-n-Darstellung, die mit Beispiel 1-1 generiert wird

TF-IDF-Darstellung

Betrachten Sie eine Sammlung von Patentdokumenten. Man würde erwarten, dass die meisten von ihnen Wörter wie *claim*, *system*, *method*, *procedure* usw. enthalten, die oftmals mehrfach wiederholt werden. Die TF-Darstellung gewichtet Wörter proportional zu ihrer Häufigkeit. Allerdings tragen allgemeine Wörter wie zum Beispiel »claim« nichts zum Verständnis eines speziellen Patents bei. Wenn umgekehrt ein seltenes Wort (wie etwa »Tetrafluoroethylene«) weniger häufig auftritt, aber sehr wahrscheinlich auf das Wesen des Patentdokuments hinweist, würden wir ihm in unserer Darstellung ein größeres Gewicht zuweisen. Die Inverse-Document-Frequency (IDF) ist eine Heuristik, um genau dies zu tun.

Die IDF-Darstellung bestraft häufige Token und belohnt seltene Token in der Vektordarstellung. Die *IDF(w)* eines Token *w* ist definiert mit Bezug auf einen Korpus als:

$$IDF(w) = \log \frac{N}{n_w}$$

wobei n_w die Anzahl der Dokumente ist, die das Wort *w* enthalten, und *N* die Gesamtanzahl der Dokumente angibt. Der TF-IDF-Score ist einfach das Produkt *(TF(w) * IDF(w)*. Erstens ist anzumerken, wenn ein sehr allgemeines Wort in allen Dokumenten auftritt (das heißt $n_w = N$), dann ist *IDF(w)* = 0 und der TF-IDF-Score ist ebenfalls 0, wodurch dieser Term vollkommen bestraft wird. Zweitens: Wenn ein Term sehr selten auftritt, vielleicht in nur einem Dokument, nimmt die IDF den maximal möglichen Wert log *N* an. Abbildung 1-2 zeigt, wie man eine TF-IDF-Darstellung einer Liste von englischen Sätzen mithilfe von `scikit-learn` generiert.

Beispiel 1-2: Eine TF-IDF-Darstellung mithilfe von scikit-learn generieren

```
from sklearn.feature_extraction.text import TfidfVectorizer
import seaborn as sns

tfidf_vectorizer = TfidfVectorizer()
tfidf = tfidf_vectorizer.fit_transform(corpus).toarray()
sns.heatmap(tfidf, annot=True, cbar=False, xticklabels=vocab,
            yticklabels= ['Sentence 1', 'Sentence 2'])
```

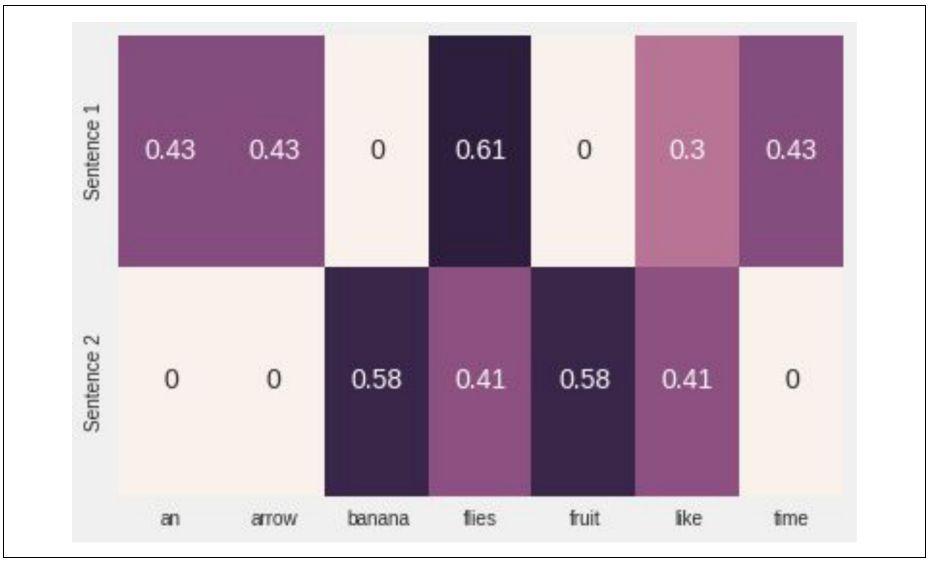

Abbildung 1-5: Die TF-IDF-Darstellung, die von Beispiel 1-2 generiert wurde

Im Deep Learning sieht man selten Eingaben, die mithilfe von heuristischen Darstellungen wie TF-IDF codiert wurden, weil das Ziel darin besteht, eine Darstellung zu lernen. Oftmals beginnen wir mit einer 1-aus-n-Codierung mithilfe ganzzahliger Indizes und einer speziellen »eingebetteten Suche«-Schicht, um Eingaben für das neuronale Netz zu konstruieren. In späteren Kapiteln stellen wir mehrere Beispiele vor, die so arbeiten.

Zielcodierung

Wie im Abschnitt »Das Paradigma des überwachten Lernens« auf Seite 2 angemerkt, kann die genaue Natur der Zielvariablen von der zu lösenden NLP-Aufgabe abhängen. So ist zum Beispiel bei maschineller Übersetzung, Zusammenfassung und Beantwortung von Fragen das Ziel ebenfalls Text und wird mit den Konzepten codiert, wie sie weiter vorn bei 1-aus-n-Codierung beschrieben wurden. Viele NLP-Aufgaben verwenden tatsächlich kategorische Benennungen, worin das Modell eine aus einem festen Satz von Benennungen vorhersagen muss. Eine gängige Art, dies zu codieren, ist die Verwendung eines eindeutigen Indexes pro Benennung,

doch diese einfache Darstellung kann problematisch werden, wenn die Anzahl der Ausgabebenennungen einfach zu groß ist. Ein Beispiel hierfür ist das Problem der *Sprachmodellierung*, bei der die Aufgabe darin besteht, das nächste Wort vorherzusagen, und zwar anhand der Wörter, die in der Vergangenheit gesehen wurden. Der Benennungsraum ist das gesamte Vokabular einer Sprache, das leicht zu mehreren Hunderttausenden Wörtern anwachsen kann und auch Sonderzeichen, Namen usw. umfasst. Auf dieses Problem kommen wir in späteren Kapiteln zurück und zeigen, wie es sich lösen lässt.

Bei manchen NLP-Problemen ist ein numerischer Wert aus einem gegebenen Text vorherzusagen. Nehmen wir beispielsweise ein englisches Essay an, in dem wir eine numerische Güte oder einen Verständlichkeits-Score zuweisen müssen. In einem Auszug aus einer Restaurant-Kritik müssen wir vielleicht eine numerische Sternbewertung auf die erste Nachkommastelle vorhersagen. Anhand der Tweets eines Benutzers sind wir vielleicht gefordert, die Altersgruppe des Benutzers vorherzusagen. Es gibt verschiedene Methoden, numerische Ziele zu codieren, doch einfach die Ziele in kategorischen »Fächern« zu platzieren – zum Beispiel »0–18«, »19–25«, »25–30« usw. – und dies als ordinales Klassifizierungsproblem zu betrachten, ist ein vernünftiger Ansatz.[5] Die Teilung der Fächer kann einheitlich oder nicht einheitlich und datengesteuert sein. Obwohl es den Rahmen dieses Buchs sprengen würde, auf dieses Thema im Detail einzugehen, wenden wir unsere Aufmerksamkeit kurz diesen Fragen zu, weil die Zielcodierung in solchen Fällen die Performance drastisch beeinflusst. Wir empfehlen Ihnen, sich Dougherty et al. (1995) anzusehen und den dort enthaltenen Referenzen zu folgen.

Berechnungsgraphen

Abbildung 1-1 hat das Paradigma des überwachten Lernens (Trainings) zusammengefasst als Datenflussarchitektur, bei der die Eingaben durch das Modell (einen mathematischen Ausdruck) transformiert werden, um Vorhersagen zu erhalten, und durch die Verlustfunktion (ein weiterer mathematischer Ausdruck), um ein Rückmeldesignal bereitzustellen, um die Parameter des Modells anzupassen. Dieser Datenfluss lässt sich komfortabel implementieren mithilfe der Datenstruktur *Berechnungsgraph*.[6] Im technischen Sinne ist ein Berechnungsgraph eine Abstraktion, die mathematische Ausdrücke modelliert. Im Kontext von Deep Learning führen die Implementierung des Berechnungsgraphen (wie zum Beispiel Theano, TensorFlow und PyTorch) zusätzliche Verwaltungsarbeiten aus, um automatische

5 Eine »ordinale« Klassifizierung ist ein Mehrklassenklassifizierungsproblem, bei dem es eine partielle Reihenfolge zwischen den Benennungen gibt. In unserem Altersbeispiel kommt die Kategorie »0–18« vor der Kategorie »19–25« usw.

6 Seppo Linnainmaa (*http://bit.ly/2Rnmdao*) hat die Idee der automatischen Differentiation auf Berechnungsgraphen erstmals bereits im Rahmen seiner Masterarbeit von 1970 eingeführt! Varianten davon sind zur Grundlage für moderne Deep-Learning-Frameworks wie Theano, TensorFlow und PyTorch geworden.

Differentiation zu implementieren, wie sie benötigt wird, um Gradienten von Parametern während des Trainings im Paradigma des überwachten Lernens zu erhalten. Wir untersuchen dies weiter im Abschnitt »Grundlagen von PyTorch« unten.

Inferenz (oder Vorhersage) ist einfach eine Ausdrucksevaluierung (ein Vorwärtsfluss in einem Berechnungsgraphen). Sehen wir uns an, wie der Berechnungsgraph Ausdrücke modelliert. Nehmen wir folgenden Ausdruck:

$$y = wx + b$$

Dieser Ausdruck lässt sich in Form von zwei Teilausdrücken $z = wx$ und $y = z + b$ schreiben. Den ursprünglichen Ausdruck können wir dann mit einem gerichteten azyklischen Graphen (DAG – Directed Acyclic Graph) darstellen, in dem die Knoten die mathematischen Operationen wie Multiplikation und Addition sind. Die Eingaben für die Operationen sind die eingehenden Kanten zu den Knoten, und die Ausgabe jeder Operation ist die ausgehende Kante. Somit lässt sich der Berechnungsgraph für den Ausdruck $y = wx + b$ wie in Abbildung 1-6 veranschaulichen. Im folgenden Abschnitt sehen wir uns an, wie PyTorch uns erlaubt, unkompliziert Berechnungsgraphen zu erzeugen und wie wir damit die Gradienten berechnen können, ohne uns selbst mit irgendwelchen Verwaltungsarbeiten herumschlagen zu müssen.

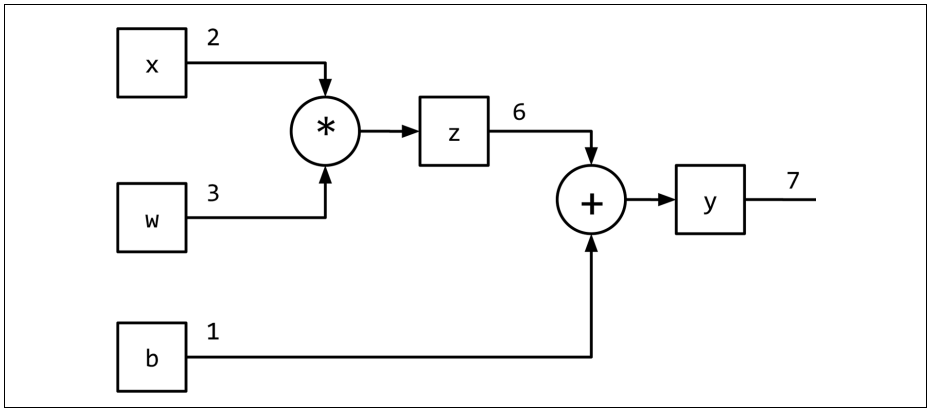

Abbildung 1-6: Den Ausdruck $y = wx + b$ mit einem Berechnungsgraphen darstellen

Grundlagen von PyTorch

In diesem Buch verwenden wir ausgiebig PyTorch für die Implementierung unserer Deep-Learning-Modelle. PyTorch ist ein von der Community getriebenes Open-Source-Framework für Deep Learning. Im Unterschied zu Theano, Caffe und TensorFlow implementiert PyTorch eine bandbasierte Methode zur *automatischen Differentiation* (*http://bit.ly/2Jrntq1*), mit der wir Berechnungsgraphen dynamisch definieren und ausführen können. Dies ist äußerst hilfreich beim Debuggen und auch für das Konstruieren ausgefeilter Modelle mit minimalem Aufwand.