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.

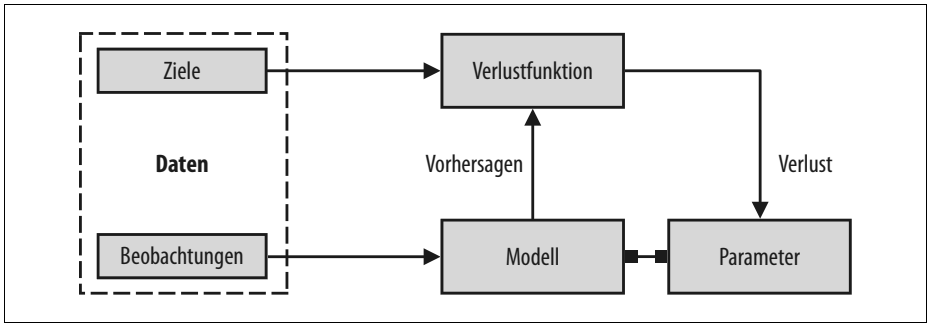

Abbildung 1-1: Das Paradigma des überwachten Lernens, ein konzeptionelles Framework für das Lernen von benannten Eingabedaten

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.

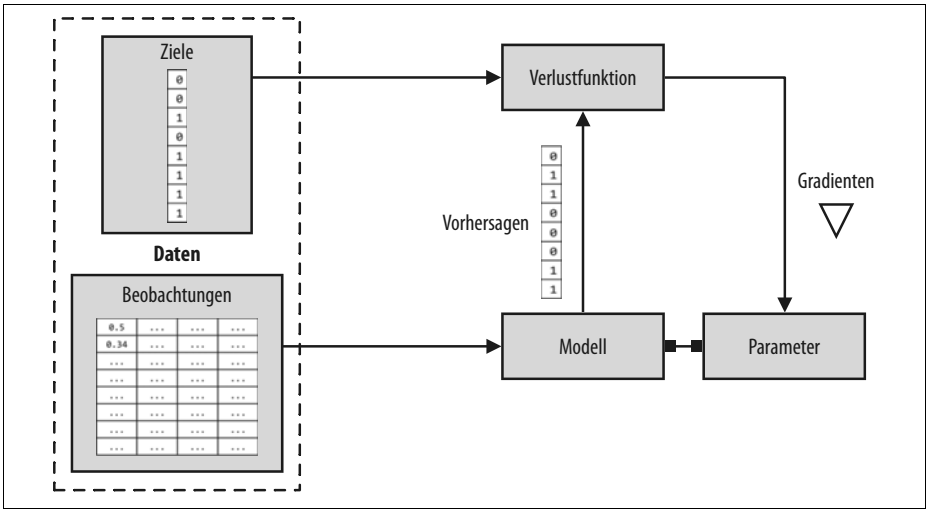

Abbildung 1-2: Beobachtung und Zielcodierung: Die Ziele und Beobachtungen von Abbildung 1-1 werden numerisch als Vektoren oder Tensoren dargestellt. Dies zusammen bezeichnet man als »Eingabecodierung«.

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.

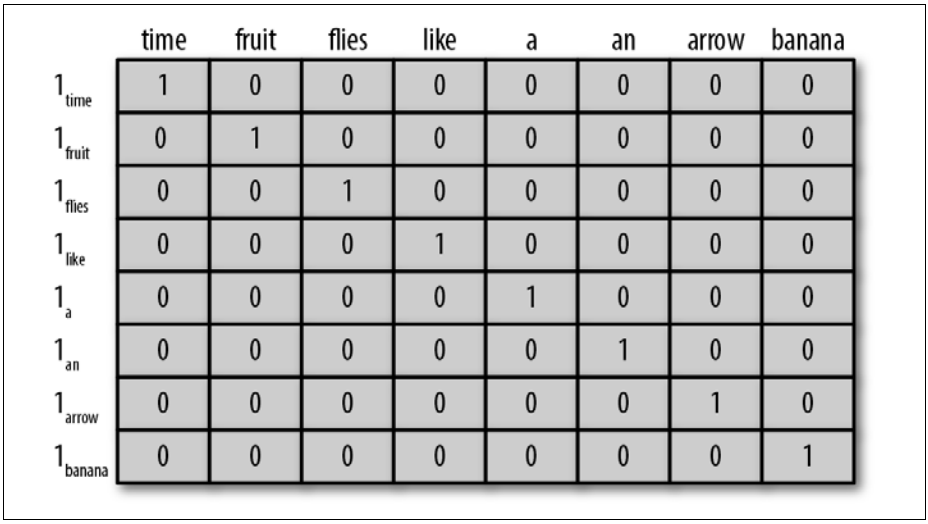

Abbildung 1-3: 1-aus-n-Darstellung für die Codierung der Sätze »Time flies like an arrow« und »Fruit flies like a banana«.

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'])
```

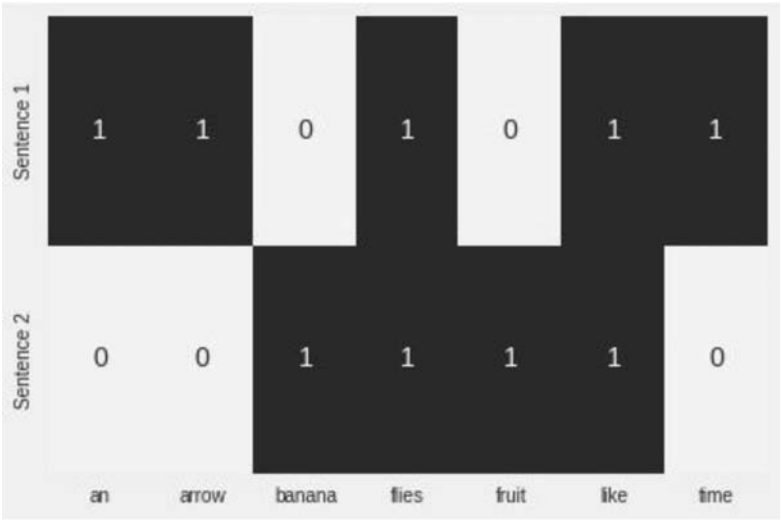

Abbildung 1-4: Die kompakte 1-aus-n-Darstellung, die mit Beispiel 1-1 generiert wird

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'])
```

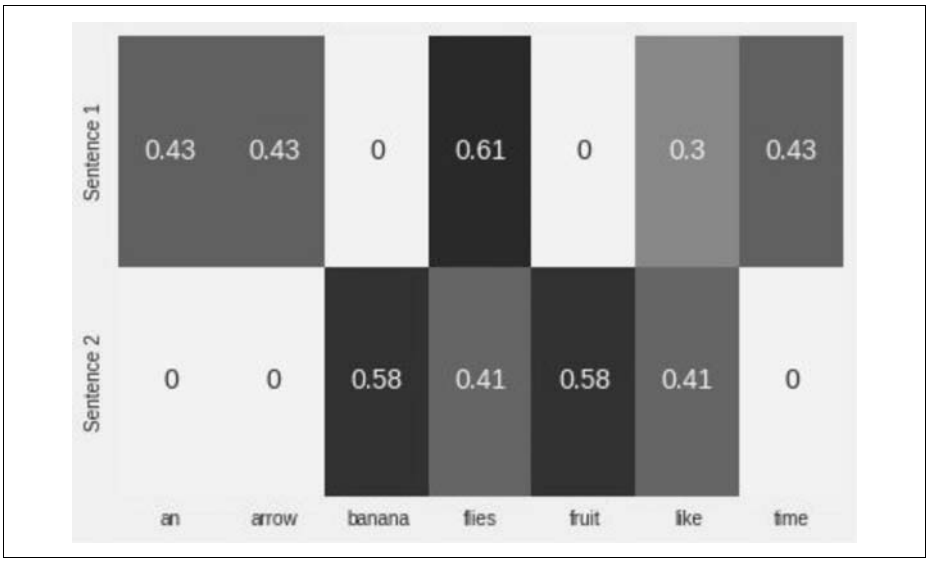

Abbildung 1-5: Die TF-IDF-Darstellung, die von Beispiel 1-2 generiert wurde

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.

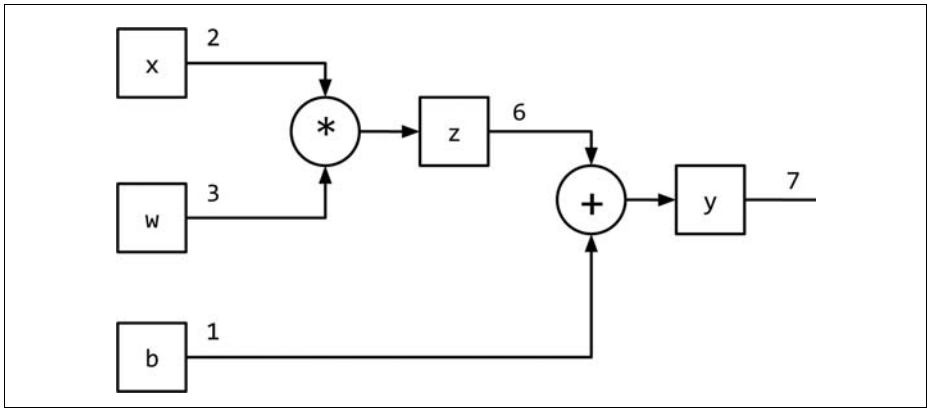

Abbildung 1-6: Den Ausdruck $y = wx + b$ mit einem Berechnungsgraphen darstellen

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.

Dynamisch und statisch berechnete Graphen

In statischen Frameworks wie Theano, Caffe und TensorFlow ist es erforderlich, den Berechnungsgraphen zuerst zu deklarieren, zu kompilieren und dann auszuführen.[7] Obwohl dies zu äußerst effizienten Implementierungen führt (die in Produktions- und mobilen Umgebungen nützlich sind), können sie sich während der Forschung und Entwicklung als ziemlich umständlich erweisen. Moderne Frameworks wie Chainer, DyNet und PyTorch implementieren dynamische Berechnungsgraphen, um einen flexibleren, imperativen Entwicklungsstil zu ermöglichen, ohne dass die Modelle vor jeder Ausführung kompiliert werden müssen. Dynamische Berechnungsgraphen sind besonders nützlich bei der Modellierung von NLP-Aufgaben, für die jede Eingabe potenziell in einer anderen Graphenstruktur resultieren könnte.

PyTorch ist eine optimierte Bibliothek zur Tensor-Manipulation, die ein ganzes Feld von Paketen für Deep Learning bietet. Im Mittelpunkt der Bibliothek steht der *Tensor* – ein mathematisches Objekt, das bestimmte mehrdimensionale Daten speichert. Ein Tensor der Ordnung null ist einfach eine Zahl oder ein *Skalar*. Ein Tensor erster Ordnung ist ein Array von Zahlen oder ein *Vektor*. Ein Tensor zweiter Ordnung ist ein Array von Vektoren oder eine *Matrix*. Demzufolge lässt sich ein Tensor verallgemeinern als *n*-dimensionales Array von Skalaren, wie Abbildung 1-7 zeigt.

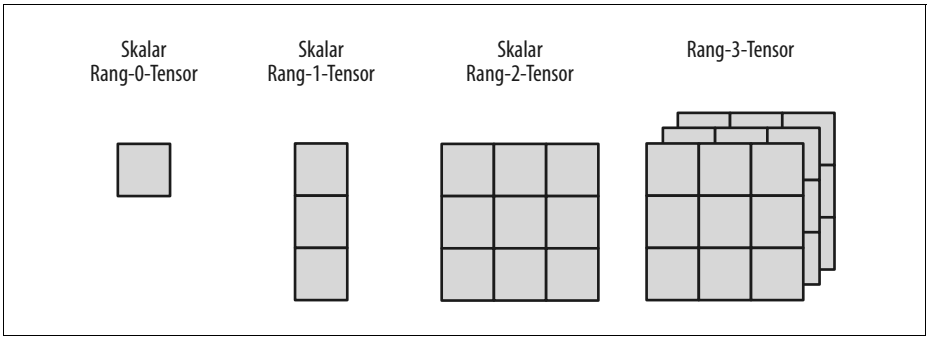

Abbildung 1-7: Tensoren als Verallgemeinerung mehrdimensionaler Arrays

In diesem Abschnitt unternehmen Sie Ihre ersten Schritte mit PyTorch, um Sie vertraut zu machen mit verschiedenen PyTorch-Operationen. Dazu gehören unter anderem:

7 Seit Version 1.7 besitzt TensorFlow einen »eager mode«, in dem es unnötig ist, den Graphen vor der Ausführung zu kompilieren. Dennoch sind statische Graphen immer noch die Hauptstütze von TensorFlow.

- Tensoren erstellen
- Indizieren, Slicen und Verknüpfen mit Tensoren
- Gradienten mit Tensoren berechnen
- CUDA-Tensoren mit GPUs verwenden

Zum jetzigen Zeitpunkt sollten Sie über ein einsatzbereites Python 3.5+ Notebook verfügen und PyTorch installiert haben, wie es der nächste Abschnitt beschreibt, damit Sie die angegebenen Beispiele immer nachvollziehen können.[8]

Und wir empfehlen, dass Sie jeweils auch die Übungen weiter hinten in den Kapiteln durcharbeiten.

PyTorch installieren

Um PyTorch auf Ihren Computern zu installieren, bestimmen Sie zunächst unter *pytorch.org* Ihre Systemumgebung. Wählen Sie Ihr Betriebssystem aus und dann den Paketmanager (wir empfehlen Conda oder Pip), gefolgt von der verwendeten Python-Version (wir empfehlen 3.5+). Anhand dieser Angaben wird der Befehl erzeugt, mit dem Sie PyTorch installieren. Bei Redaktionsschluss der Übersetzung lautet er für die Conda-Umgebung zum Beispiel wie folgt:

```
conda install pytorch torchvision -c pytorch
```

Wenn Sie über einen CUDA-fähigen Grafikprozessor (Graphics Processor Unit, GPU) verfügen, können Sie auch die entsprechende Version von CUDA wählen. Zusätzliche Einzelheiten finden Sie in den Installationsanweisungen auf *pytorch.org*.

Tensoren erstellen

Wir definieren zunächst eine Hilfsfunktion `describe(x)`, die verschiedene Eigenschaften eines Tensors *x* zusammenfasst, beispielsweise den Typ, die Dimensionen und den Inhalt des Tensors:

```
Input[0]

def describe(x):
    print("Type: {}".format(x.type()))
    print("Shape/size: {}".format(x.shape))
    print("Values: \n{}".format(x))
```

PyTorch erlaubt es uns mit dem Paket `torch`, Tensoren auf viele verschiedene Arten zu erstellen. So lässt sich zum Beispiel ein zufälliger Tensor erzeugen, indem man seine Dimensionen angibt, wie es Beispiel 1-3 zeigt.

8 Den Code für diesen Abschnitt finden Sie unter */chapters/chapter_1/PyTorch_Basics.ipynb* im GitHub-Repository für dieses Buch. (*https://nlproc.info/PyTorchNLPBook/repo/*)

Beispiel 1-3: Einen Tensor in PyTorch mit torch.Tensor erstellen

```
Input[0]:

import torch
describe(torch.Tensor(2, 3))

Output[0]:

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 3.2018e-05, 4.5747e-41, 2.5058e+25],
        [ 3.0813e-41, 4.4842e-44, 0.0000e+00]])
```

Wir können auch einen Tensor erzeugen, indem wir ihn zufällig mit Werten aus einer Gleichverteilung über dem Intervall `[0, 1)` oder der Standardnormalverteilung[9] wie in Beispiel 1-4 initialisieren. Auch zufällig initialisierte Tensoren wie etwa von der Gleichverteilung sind wichtig, wie Kapitel 3 und 4 noch erläutern werden.

Beispiel 1-4: Einen zufällig initialisierten Tensor erzeugen

```
Input[0]

import torch
describe(torch.rand(2, 3))   # gleichverteilt zufällig
describe(torch.randn(2, 3))  # normalverteilt zufällig

Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.0242, 0.6630, 0.9787],
        [ 0.1037,   0.3920, 0.6084]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[-0.1330, -2.9222, -1.3649],
        [ 2.3648, 1.1561, 1.5042]])
```

Wir können auch Tensoren erzeugen, die mit dem gleichen Skalar gefüllt sind. Um einen Tensor mit Nullen oder Einsen zu erzeugen, stehen vordefinierte Funktionen bereit, und für das Füllen der Tensoren mit spezifischen Werten können wir die Methode `fill_()` heranziehen. Jede PyTorch-Methode mit einem Unterstrich (`_`) verweist auf eine In-place-Operation; das heißt, sie modifiziert den Inhalt an Ort und Stelle, ohne ein neues Objekt zu erzeugen, wie Beispiel 1-5 zeigt.

9 Die Standardnormalverteilung ist eine Normalverteilung mit dem Mittelwert = 0 und der Varianz = 1.

Beispiel 1-5: Einen gefüllten Tensor erzeugen

```
Input[0]

import torch
describe(torch.zeros(2, 3))
x = torch.ones(2, 3)
describe(x)
x.fill_(5)
describe(x)

Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0., 0., 0.],
        [ 0., 0., 0.]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1., 1., 1.],
        [ 1., 1., 1.]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 5., 5., 5.],
        [ 5., 5., 5.]])
```

Beispiel 1-6 demonstriert, wie wir einen Tensor auch deklarativ mithilfe von Python-Listen erzeugen können.

Beispiel 1-6: Einen Tensor aus Listen erzeugen und initialisieren

```
Input[0]

x = torch.Tensor([[1, 2, 3],
                  [4, 5, 6]])
describe(x)

Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1., 2., 3.],
        [ 4., 5., 6.]])
```

Die Werte können entweder von einer Liste kommen, wie im Beispiel, oder von einem NumPy-Array. Und natürlich ist es immer möglich, ebenso von einem PyTorch-Tensor zu einem NumPy-Array überzugehen. Beachten Sie, dass der Tensor den Typ `DoubleTensor` hat anstelle des standardmäßigen `FloatTensor` (siehe den

nächsten Abschnitt). Dieser entspricht dem Datentyp `float64` der NumPy-Zufallsmatrix, wie in Beispiel 1-7 dargestellt.

Beispiel 1-7: Einen Tensor von NumPy erzeugen und initialisieren

```
Input[0]

import torch
import numpy as np
npy = np.random.rand(2, 3)
describe(torch.from_numpy(npy))

Output[0]

Type: torch.DoubleTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.8360, 0.8836, 0.0545],
        [ 0.6928, 0.2333, 0.7984]], dtype=torch.float64)
```

Die Möglichkeit, zwischen NumPy-Arrays und PyTorch-Tensoren zu konvertieren, wird wichtig, wenn Sie mit Legacy-Bibliotheken arbeiten, die NumPy-formatierte numerische Werte verwenden.

Typ und Größe von Tensoren

Jeder Tensor besitzt einen zugeordneten Typ und eine Größe. Wenn Sie den Konstruktor `torch.Tensor` verwenden, hat der Tensor den Standardtyp `torch.FloatTensor`. Allerdings können Sie einen Tensor in einen anderen Typ konvertieren (`float`, `long`, `double` usw.), indem Sie ihn während der Initialisierung spezifizieren oder später mit einer der Methoden zur Typumwandlung ändern. Der Initialisierungstyp lässt sich nach zwei Verfahren festlegen: entweder, indem Sie direkt den Konstruktor des jeweiligen Tensor-Typs aufrufen, oder die spezielle Methode `torch.tensor()` verwenden und den `dtype` übergeben, wie Beispiel 1-8 zeigt.

Beispiel 1-8: Tensor-Eigenschaften

```
Input[0]

x = torch.FloatTensor([[1, 2, 3],
                       [4, 5, 6]])
describe(x)

Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1., 2., 3.],
        [ 4., 5., 6.]])
```

```
Input[1]

x = x.long()
describe(x)

Output[1]

Type: torch.LongTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1, 2, 3],
        [ 4, 5, 6]])

Input[2]

x = torch.tensor([[1, 2, 3],
                  [4, 5, 6]], dtype=torch.int64)
describe(x)

Output[2]

Type: torch.LongTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1, 2, 3],
        [ 4, 5, 6]])

Input[3]

x = x.float()
describe(x)

Output[3]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1., 2., 3.],
        [ 4., 5., 6.]])
```

Mit der Eigenschaft `shape` und der Methode `size()` eines Tensor-Objekts greifen wir auf die Abmessungen seiner Dimensionen zu. Die beiden Arten des Zugriffs auf diese Abmessungen sind nahezu synonym. Die Form des Tensors zu inspizieren, ist unverzichtbar beim Debuggen von PyTorch-Code.

Tensor-Operationen

Nachdem Sie Ihre Tensoren erstellt haben, können Sie auf ihnen operieren, wie Sie es mit herkömmlichen Typen in Programmiersprachen tun würden, wie zum Beispiel `+`, `-`, `*` und `/`. Anstelle der Operatoren können Sie auch Funktionen verwenden, die den symbolischen Operatoren entsprechen, wie es Beispiel 1-9 für `.add()` zeigt.

Beispiel 1-9: Tensor-Operationen: Addition

```
Input[0]

import torch
x = torch.randn(2, 3)
describe(x)

Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.0461, 0.4024, -1.0115],
        [ 0.2167, -0.6123, 0.5036]])

Input[1]

describe(torch.add(x, x))

Output[1]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.0923, 0.8048, -2.0231],
        [ 0.4335, -1.2245, 1.0072]])

Input[2]

describe(x + x)

Output[2]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.0923, 0.8048, -2.0231],
        [ 0.4335, -1.2245, 1.0072]])
```

Es gibt auch Operationen, die Sie auf eine bestimmte Dimension eines Tensors anwenden können. Wie Sie vielleicht schon bemerkt haben, stellen wir die Zeilen für einen zweidimensionalen Tensor als Dimension `0` und die Spalten als Dimension `1` dar, wie es Beispiel 1-10 veranschaulicht.

Beispiel 1-10: Dimensionen-basierte Tensor-Operationen

```
Input[0]

import torch
x = torch.arange(6)
describe(x)
```

```
Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([6])
Values:
tensor([ 0., 1., 2., 3., 4., 5.])

Input[1]

x = x.view(2, 3)
describe(x)

Output[1]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0., 1., 2.],
        [ 3., 4., 5.]])

Input[2]

describe(torch.sum(x, dim=0))

Output[2]

Type: torch.FloatTensor
Shape/size: torch.Size([3])
Values:
tensor([ 3., 5., 7.])

Input[3]

describe(torch.sum(x, dim=1))

Output[3]

Type: torch.FloatTensor
Shape/size: torch.Size([2])
Values:
tensor([ 3., 12.])

Input[4]

describe(torch.transpose(x, 0, 1))

Output[4]

Type: torch.FloatTensor
Shape/size: torch.Size([3, 2])
Values:
tensor([[ 0., 3.],
        [ 1., 4.],
        [ 2., 5.]])
```

Oftmals müssen wir komplexere Operationen ausführen, die aus einer Kombination von Indizieren, Slicing, Verknüpfen und Mutationen bestehen. Wie NumPy und andere numerische Bibliotheken bringt PyTorch vordefinierte Funktionen mit, die derartige Tensor-Manipulationen stark vereinfachen.

Indizieren, Slicing und Verknüpfen

Als NumPy-Anwender dürfte Ihnen das Indizierungs- und Slicing-Schema wie in Beispiel 1-11 gezeigt sehr vertraut vorkommen.

Beispiel 1-11: Einen Tensor slicen und indizieren

```
Input[0]

import torch
x = torch.arange(6).view(2, 3)
describe(x)

Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0., 1., 2.],
        [ 3., 4., 5.]])

Input[1]

describe(x[:1, :2])

Output[1]

Type: torch.FloatTensor
Shape/size: torch.Size([1, 2])
Values:
tensor([[ 0., 1.]])

Input[2]

describe(x[0, 1])

Output[2]

Type: torch.FloatTensor
Shape/size: torch.Size([])
Values:
1.0
```

Beispiel 1-12 demonstriert, dass PyTorch auch über Funktionen für komplexe Indizierungs- und Slicing-Operationen verfügt, bei denen Sie daran interessiert sind, auf nicht aneinandergrenzende Elemente eines Tensors effizient zuzugreifen.

Beispiel 1-12: Komplexes Indizieren: Indizieren nicht aneinandergrenzender Elemente eines Tensors

```
Input[0]

indices = torch.LongTensor([0, 2])
describe(torch.index_select(x, dim=1, index=indices))

Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 0., 2.],
        [ 3., 5.]])

Input[1]

indices = torch.LongTensor([0, 0])
describe(torch.index_select(x, dim=0, index=indices))


Output[1]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0., 1., 2.],
        [ 0., 1., 2.]])

Input[2]

row_indices = torch.arange(2).long()
col_indices = torch.LongTensor([0, 1])
describe(x[row_indices, col_indices])

Output[2]

Type: torch.FloatTensor
Shape/size: torch.Size([2])
Values:
tensor([ 0., 4.])
```

Der Typ der Indizes muss `LongTensor` sein – eine Forderung für das Indizieren mit PyTorch-Funktionen. Es ist auch möglich, Tensoren mithilfe vordefinierter Verkettungsfunktionen zu verbinden, indem man die Tensoren und die Dimension angibt, wie Beispiel 1-13 zeigt.

Beispiel 1-13: Tensoren verketten

```
Input[0]

import torch
x = torch.arange(6).view(2,3)
describe(x)
```

```
Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0., 1., 2.],
        [ 3., 4., 5.]])

Input[1]

describe(torch.cat([x, x], dim=0))

Output[1]

Type: torch.FloatTensor
Shape/size: torch.Size([4, 3])
Values:
tensor([[ 0., 1., 2.],
        [ 3., 4., 5.],
        [ 0., 1., 2.],
        [ 3., 4., 5.]])

Input[2]

describe(torch.cat([x, x], dim=1))

Output[2]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 6])
Values:
tensor([[ 0., 1., 2., 0., 1., 2.],
        [ 3., 4., 5., 3., 4., 5.]])

Input[3]

describe(torch.stack([x, x]))

Output[3]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 2, 3])
Values:
tensor([[[ 0., 1., 2.],
        [ 3., 4., 5.]],

        [[ 0., 1., 2.],
        [ 3., 4., 5.]]])
```

PyTorch implementiert zudem hocheffiziente Operationen der linearen Algebra auf Tensoren, beispielsweise Multiplikation, Inverse und Spur, wie Beispiel 1-14 zeigt.

Beispiel 1-14: Lineare Algebra auf Tensoren: Multiplikation

```
Input[0]

import torch
x1 = torch.arange(6).view(2, 3)
describe(x1)
```

```
Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0., 1., 2.],
        [ 3., 4., 5.]])

Input[1]

x2 = torch.ones(3, 2)
x2[:, 1] += 1 describe(x2)

Output[1]

Type: torch.FloatTensor
Shape/size: torch.Size([3, 2])
Values:
tensor([[ 1., 2.],
        [ 1., 2.],
        [ 1., 2.]])

Input[2]

describe(torch.mm(x1, x2))

Output[2]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 3.,6.],
        [ 12., 24.]])
```

Bislang haben wir uns nur Verfahren angesehen, um konstante PyTorch-Tensor-Objekte zu erstellen und zu manipulieren. Genauso wie eine Programmiersprache (wie zum Beispiel Python) mithilfe von Variablen Datenelemente kapselt und zusätzliche Informationen über diese Daten verwaltet (wie etwa die Speicheradresse, an der das Datenelement abgelegt ist), behandeln PyTorch-Tensoren die erforderliche Buchhaltung für den Aufbau von Berechnungsgraphen beim maschinellen Lernen, indem sie einfach ein boolesches Flag zur Instanziierungszeit aktivieren.

Tensoren und Berechnungsgraphen

Die PyTorch-Klasse `tensor` kapselt die Daten (den Tensor selbst) und einen Bereich von Operationen, wie zum Beispiel algebraische Operationen, Indizieren und Umformen. Wenn aber, wie in Beispiel 1-15 gezeigt, das boolesche Flag `requires_grad` für einen Tensor auf `True` gesetzt ist, werden Buchhaltungsoperationen aktiviert, die den Gradienten am Tensor sowie die Gradientenfunktion verfolgen können, die beide erforderlich sind, um das im Abschnitt »Das Paradigma des überwachten Lernens« auf Seite 2 beschriebene gradientenbasierte Lernen zu unterstützen.

Beispiel 1-15: Tensoren für die Buchhaltung von Gradienten

```
Input[0]

import torch
x = torch.ones(2, 2, requires_grad=True)
describe(x)
print(x.grad is None)

Output[0]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 1., 1.],
        [ 1., 1.]])
True

Input[1]

y = (x + 2) * (x + 5) + 3
describe(y)
print(x.grad is None)

Output[1]

Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 21., 21.],
        [ 21., 21.]])
True

Input[2]

z = y.mean()
describe(z)
z.backward()
print(x.grad is None)

Output[2]

Type: torch.FloatTensor
Shape/size: torch.Size([])
Values:
21.0
False
```

Wenn Sie einen Tensor mit `requires_grad=True` erzeugen, fordern Sie PyTorch auf, Buchhaltungsinformationen zu verwalten, die Gradienten berechnen. Erstens verfolgt PyTorch die Werte des Vorwärtspasses. Am Ende der Berechnungen wird dann ein einzelner Skalar verwendet, um einen Rückwärtspass zu berechnen. Initiiert wird der Rückwärtspass mit der Methode `backward()` auf einem Tensor, der sich aus der Bewertung einer Verlustfunktion ergibt. Der Rückwärtspass berechnet einen Gradientenwert für ein Tensor-Objekt, das am Vorwärtspass beteiligt war.

Im Allgemeinen ist der Gradient ein Wert, der den Anstieg der Ausgabe einer Funktion bezogen auf die Eingabe der Funktion darstellt. Im Rahmen von Berechnungsgraphen existieren Gradienten für jeden Parameter im Modell, und man

kann sie sich vorstellen als den Beitrag des Parameters zum Fehlersignal. In PyTorch können Sie auf die Gradienten für die Knoten im Berechnungsgraphen mit der Membervariablen `grad` zugreifen. Optimierer verwenden die Variable `grad`, um die Werte der Parameter zu aktualisieren.

CUDA-Tensoren

Bis jetzt haben wir unsere Tensoren im CPU-Arbeitsspeicher alloziert. Für Operationen der linearen Algebra kann es aber sinnvoll sein, eine GPU zu nutzen, sofern Sie über eine verfügen. Um eine GPU zu verwenden, müssen Sie zuerst den Tensor im Speicher der GPU allozieren. Der Zugriff auf die GPUs erfolgt über eine spezialisierte API. Die sogenannte CUDA API stammt von NVIDIA und lässt sich auch nur für NVIDIA GPUs verwenden.[10] PyTorch bietet CUDA-Tensor-Objekte, die sich von den normalen Tensoren, die an die CPU gebunden sind, nicht unterscheiden lassen, außer in der Art und Weise, wie sie intern alloziert werden.

Mit PyTorch ist es sehr einfach, diese CUDA-Tensoren zu erzeugen, den Tensor von der CPU auf die GPU zu übertragen und dabei seinen zugrunde liegenden Typ zu bewahren. Die bevorzugte Methode in PyTorch ist es, geräte-agnostisch zu sein und Code zu schreiben, der sowohl auf der GPU als auch auf der CPU funktioniert. In Beispiel 1-16 testen wir zunächst mit `torch.cuda.is_available()`, ob eine GPU verfügbar ist, und rufen den Gerätenamen mit `torch.device()` ab. Alle zukünftigen Tensoren werden dann instanziiert und mit der Methode `to(device)` auf das Zielgerät verschoben.

Beispiel 1-16: CUDA-Tensoren erzeugen

```
Input[0]

import torch
print (torch.cuda.is_available())

Output[0]

True

Input[1]

# Bevorzugte Methode: Geräteagnostische Tensor-Instanziierung
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print (device)

Output[1]

cuda
```

10 Das heißt, dass Sie bei GPUs eines anderen Herstellers – sagen wir AMD oder ARM – momentan außen vor bleiben (wobei Sie natürlich PyTorch trotzdem im CPU-Modus verwenden können). Allerdings kann sich das in Zukunft ändern (*http://bit.ly/2JpSafj*).

```
Input[2]

x = torch.rand(3, 3).to(device)
describe(x)

Output[2]

Type: torch.cuda.FloatTensor
Shape/size: torch.Size([3, 3])
Values:
tensor([[ 0.9149, 0.3993, 0.1100],
        [ 0.2541, 0.4333, 0.4451],
        [ 0.4966, 0.7865, 0.6604]], device='cuda:0')
```

Um sowohl auf CUDA- als auch auf Nicht-CUDA-Objekten zu operieren, müssen wir sicherstellen, dass sie sich auf demselben Gerät befinden. Andernfalls schlagen die Berechnungen fehl, wie Beispiel 1-17 zeigt. Diese Situation entsteht zum Beispiel, wenn Überwachungsmetriken berechnet werden, die nicht Teil des Berechnungsgraphen sind. Wenn Sie auf zwei Tensor-Objekten operieren, müssen Sie gewährleisten, dass sich beide auf demselben Gerät befinden.

Beispiel 1-17: CUDA-Tensoren mit CPU-gebundenen Tensoren mischen

```
Input[0]

y = torch.rand(3, 3)
x + y

Output[0]

---------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
      1 y = torch.rand(3, 3)
----> 2 x + y

RuntimeError: Expected object of type
torch.cuda.FloatTensor but found type torch.FloatTensor for argument #3 'other'

Input[1]

cpu_device = torch.device("cpu")
y = y.to(cpu_device) x = x.to(cpu_device) x + y

Output[1]

tensor([[ 0.7159, 1.0685, 1.3509],
        [ 0.3912, 0.2838, 1.3202],
        [ 0.2967, 0.0420, 0.6559]])
```

Denken Sie daran, dass es teuer ist, die Daten zwischen GPU und CPU hin und her zu schaufeln. Deshalb ist es üblich, viele der parallelisierbaren Berechnungen auf der GPU abzuwickeln und dann lediglich das letzte Ergebnis an die CPU zurück zu übertragen. Dadurch können Sie die GPUs vollständig nutzen. Haben Sie mehrere für CUDA sichtbare Geräte (das heißt mehrere GPUs), empfiehlt es sich, die

Umgebungsvariable `CUDA_VISIBLE_DEVICES` zu verwenden, wenn Sie das Programm ausführen:

```
CUDA_VISIBLE_DEVICES=0,1,2,3 python main.py
```

In diesem Buch gehen wir nicht weiter auf Parallelität und Multi-GPU-Training ein, doch sind diese Techniken wichtig beim Skalieren von Experimenten und manchmal sogar, um große Modelle zu trainieren. Sehen Sie sich am besten die Dokumentation und die Diskussionsforen zu PyTorch an (*http://bit.ly/2PqdsPF*), um zusätzliche Hilfe und Unterstützung zu diesem Thema zu erhalten.

Übungen

Ein Thema lässt sich am besten beherrschen, wenn man Probleme selbstständig löst. Hier finden Sie einige Aufwärmübungen. Viele der Probleme erfordern es, dass Sie die offizielle Dokumentation (*http://bit.ly/2F6PSU8*) durchgehen und nach hilfreichen Funktionen suchen.

1. Erzeugen Sie einen zweidimensionalen Tensor und fügen Sie dann eine Dimension der Größe `1` bei Dimension `0` ein.
2. Entfernen Sie die zusätzliche Dimension, die Sie dem Tensor in Übung 1 hinzugefügt haben.
3. Erzeugen Sie einen zufälligen Tensor der Form `5x3` im Intervall `[3, 7)`.
4. Erzeugen Sie einen Tensor mit Werten aus einer Normalverteilung (Mittelwert=0; Standardabweichung=1).
5. Rufen Sie die Indizes aller Elemente ungleich null im Tensor `torch.Tensor([1, 1, 1, 0, 1])` ab.
6. Erzeugen Sie einen zufälligen Tensor der Größe (3, 1) und stapeln Sie dann horizontal vier Kopien aneinander.
7. Geben Sie das Batch-Matrix-Matrix-Produkt zweier dreidimensionaler Matrizen (`a=torch.rand(3,4,5)`, `b=torch.rand(3,5,4)`) zurück.
8. Geben Sie das Batch-Matrix-Matrix-Produkt einer dreidimensionalen Matrix und einer zweidimensionalen Matrix (`a=torch.rand(3,4,5)`, `b=torch.rand(5,4)`) zurück.

Lösungen

```
1. a = torch.rand(3, 3)
   a.unsqueeze(0)
2. a.squeeze(0)
3. 3 + torch.rand(5, 3) * (7 - 3)
4. a = torch.rand(3, 3)
   a.normal_()
```

```
5. a = torch.Tensor([1, 1, 1, 0, 1])
   torch.nonzero(a)
6. a = torch.rand(3, 1)
   a.expand(3, 4)
7. a = torch.rand(3, 4, 5)
   b = torch.rand(3, 5, 4)
   torch.bmm(a, b)
8. a = torch.rand(3, 4, 5)
   b = torch.rand(5, 4)
   torch.bmm(a, b.unsqueeze(0).expand(a.size(0), *b.size()))
```

Zusammenfassung

Dieses Kapitel hat die Hauptthemen dieses Buchs – Verarbeitung natürlicher Sprache oder NLP (von engl. Natural Language Processing) und Deep Learning – eingeführt und ein detailliertes Verständnis für das Paradigma des überwachten Lernens entwickelt. Nunmehr sollten Sie die verschiedenen relevanten Begriffe wie Beobachtungen, Modelle, Parameter, Vorhersagen, Verlustfunktionen, Darstellungen, Lernen/Training und Inferenz verinnerlicht haben oder zumindest wissen, dass es sie gibt. Außerdem haben Sie gesehen, wie man Eingaben (Beobachtungen und Ziele) für Lernaufgaben mittels 1-aus-n-Codierung codiert, und wir haben auch zahlenmäßige Darstellungen wie TF und TF-IDF untersucht. Bei unserer Tour durch PyTorch haben wir zuerst untersucht, was Berechnungsgraphen sind, dann statische und dynamische Berechnungsgraphen gegenübergestellt und einen Ausflug in die Operationen der Tensor-Manipulationen von PyTorch unternommen. Kapitel 2 gibt einen Überblick über herkömmliche NLP. Diese beiden Kapitel sollten die notwendigen Grundlagen schaffen, wenn das Thema des Buchs für Sie relativ neu ist, und Sie auf die übrigen Kapitel vorbereiten.

Literaturhinweise

[1] Offizielle API-Dokumentation von PyTorch (*http://bit.ly/2RjBxVw*).

[2] Dougherty, James, Ron Kohavi und Mehran Sahami: *Supervised and Unsupervised Discretization of Continuous Features*. In: Proceedings of the 12th International Conference on Machine Learning, 1995.

[3] Collobert, Ronan und Jason Weston: *A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning*. In: Proceedings of the 25th International Conference on Machine Learning, 2008.