nicht nur, wie Transformer-Modelle funktionieren, sondern wir sehen uns auch an, wie sich vortrainierte Modelle, die über den Transformer-Hub *Hugging Face* vertrieben werden, sich für verschiedene Aufgaben nachtrainieren lassen. Das Buch schließt im **zwölften Kapitel** mit einer Zusammenfassung einiger grundlegender Konzepte und einer Diskussion der Stärken und Schwächen neuronaler Netze.

Wie Sie mit den Codebeispielen arbeiten können

Die meisten Kapitel bestehen aus einem Mix aus Einführungen in Konzepte maschinellen Lernens und aus einem praxisorientierten Teil, in denen diese Konzepte mit Python umgesetzt werden. Wenn Sie die Beispiele auf Ihrem Rechner laufen lassen möchten, müssen Sie die Voraussetzungen dafür schaffen.

Wir verwenden Python in der *Version 3.7.6*, eingebettet in eine virtuelle Umgebung, die über die Data Science-Plattform *Anaconda* verwaltet wird. Anaconda bietet unter anderem den Vorteil, dass es bei der Installation externer Pakete sicherstellt, dass die Pakete untereinander harmonieren. Da wir einige Pakete installieren müssen und da diese Pakete in ihrer Arbeit wiederum auf weitere Unterpakete angewiesen sind, ist das ziemlich hilfreich.

Um also arbeitsfähig zu sein, benötigen Sie die folgenden Bibliotheken in Ihrer virtuellen Umgebung:

```
matplotlib, Version=3.4.2
nltk, Version=3.6.2
numpy, Version=1.19.2
pandas, Version=1.0.3
scikit-learn, Version=0.22.1
tensorflow, Version=2.3.0
transformers, Version=4.10.2
```

Bis auf *Transformers* (das zum Zeitpunkt der Drucklegung mit pip installiert werden muss) sind alle Bibliotheken über Anaconda verwaltbar (installieren mit *conda install*). Sie können den Code vermutlich auch mit anderen, aktuelleren Versionen von Python und den genannten Bibliotheken ausführen. Allerdings ist es möglich, dass Sie dann an der einen oder anderen Stelle auf Warnungen, Fehlermeldungen oder auf andere Probleme stoßen, die wir nicht vorhersehen können.

Die Codebeispiele im Buch sind über *GitHub* als *Jupyter Notebooks* verfügbar. Sie können den Code entweder im Internet unter *https://github.com/tplusone/hanser_deep_nlp* abrufen und ansehen oder, wenn Sie die Software *Git* installiert haben, das gesamte Repository inklusive Codebeispielen und Beispieldaten über das Terminal mit dem folgenden Befehl auf Ihren Rechner ziehen:

```
git clone https://github.com/tplusone/hanser_deep_nlp.git
```

2 Textdaten verarbeiten und vorverarbeiten

Wenn es um die Verarbeitung numerischer und textbasierter Informationen geht, hat Python gegenüber anderen Programmiersprachen ein paar entscheidende Vorteile. Schon die Standardbibliothek bietet eine Vielzahl einfacher Funktionen und Klassen, um Textdaten in Form zu bringen, zu transformieren und zu strukturieren. So ist es ein Leichtes, einmal tokenisierte Texte in Arrays zu verwalten, Auszüge zu extrahieren oder mithilfe von Schleifen oder List Comprehensions Transformationen durchzuführen. Auch Casting-Operationen, die Arrays in Sets, Tuples oder Dictionarys verwandeln, funktionieren in den meisten Fällen vorhersehbar und problemlos. So lassen sich Texte für das Anlernen in Form bringen.

Um einen maschinellen Lernalgorithmus zu trainieren, brauchen wir aber andere Datenklassen. Statistische Verfahren operieren mit mathematischen Funktionen und mögen daher keine Überraschungen, wenn es um Datentypen und die Struktur der Datenbehälter geht. Da eines der Grundprinzipien in Python aber die Abkehr von Typsicherheit zugunsten von Duck-Typing ist, benötigen wir eine Alternative, die dieses Anforderungsprofil erfüllt. Die Bibliothek *NumPy* (Numerical Python) ist dabei die erste Wahl. NumPy ist zum Glück auf die Datenklassen der Standardbibliothek abgestimmt, sodass Castings in beide Richtungen reibungslos funktionieren.

Im Folgenden sehen wir uns einige ausgewählte Techniken, Klassen und Bibliotheken an, die für die Vorverarbeitung von Texten zur Analyse mit Lernalgorithmen von Bedeutung sind. Dabei handelt es sich um keine auch nur annähernd vollständige Abhandlung der verschiedenen Möglichkeiten. Wir werfen aber einen Blick auf die Verfahren, die in den nachfolgenden Kapiteln immer wieder verwendet werden und die wir dort nicht noch einmal im Detail vorstellen. Neben der Tokenisierung und numerischen Encodierung von Wörtern geht es um die Repräsentation von Wörtern als One-Hot-Sets und natürlich um die Arbeit mit der Bibliothek NumPy.

2.1 Grundlegende Techniken der Verarbeitung von Textdaten

Texte liegen nach dem Einlesen in Python normalerweise als Strings vor. Eine Besonderheit der String-Klasse in Python ist, dass sie sich wie ein Iterable verhält. Die einzelnen Buchstaben werden als Characters auf Indexpositionen abgespeichert. Man kann deshalb über

einen String sowohl iterieren als auch über den Aufruf einer Indexposition bzw. über Slicing einzelne oder mehrere Buchstaben herauslösen:

```
1.  text = 'Die Sonne steht hoch am Himmel.'
2.  text[5], text[5:10]
```

Ausgabe:

```
('o', 'onne ')
```

Da in vielen statistischen Anwendungen Wörter oder Token als kleinste Analyseeinheiten fungieren, müssen wir Texte fast immer auf dieser Ebene zerlegen. Dieser Vorgang nennt sich *Tokenisierung*. Wir könnten uns mit einem Regex zwar einen eigenen Tokenizer zusammenbauen, einfacher geht es allerdings mit einem getesteten Produkt, das Wörter und Satzzeichen an verschiedenen Positionen erkennt und extrahiert. Die *word_tokenize*-Funktion aus dem NLTK-Modul erledigt genau diese Arbeit und gibt bei Übergabe eines Strings eine Liste der enthaltenen Wörter inklusive Satzzeichen zurück:

```
3.  from nltk.tokenize import word_tokenize
4.
5.  text = 'Die Sonne steht hoch am Himmel.'
6.  word_tokenize(text)
```

Ausgabe:

```
['Die', 'Sonne', 'steht', 'hoch', 'am', 'Himmel', '.']
```

In längeren Texten wiederholen sich die Wörter in der Regel mehrfach. Wenn wir von jedem dieser Wörter jeweils nur ein Exemplar behalten möchten, können wir die Liste in ein Set umwandeln. Eine angenehme Nebenwirkung dieses Vorgangs ist, dass automatisch alle Duplikate eliminiert werden:

```
7.  words = ['die', 'Sonne', 'Sonne', 'scheint', 'scheint']
8.  set(words)
```

Ausgabe:

```
{'Sonne', 'die', 'scheint'}
```

Wie man Wörter in numerische Form bringt, um damit einen Machine Learning-Algorithmus zu füttern, sehen wir uns im nächsten Abschnitt genauer an. Manchmal brauchen wir allerdings keine spezielle Encodierung, sondern lediglich eine numerische Repräsentation der Wörter aus den Trainingsdaten (zum Beispiel, wenn wir eine Zielvariable vorbereiten). In diesem Fall bietet es sich an, zur Encodierung ein Dictionary zu verwenden. Es enthält für jedes Wort (*Key*) eine Ganzzahl (*Value*). Damit können wir die einzelnen Wörter aus einer Textsequenz nachschlagen und encodieren. In einem zweiten (zur Decodierung vorgesehenen) Dictionary sind die Keys und Values vertauscht: Bei Übergabe einer Ganzzahl erhalten wir das zugeordnete Wort zurück.

Die Ordnung der Wörter und Ganzzahlen in den Dictionarys können wir letztlich willkürlich festlegen, da es für den Lernalgorithmus unerheblich ist, welche Ganzzahlen für welche Wörter stehen. Wichtig ist nur, dass jedem Wort je eine eigene Ganzzahl zugeordnet ist. Allerdings bietet es sich natürlich an, die Reihenfolge der Zahlen mit der alphabetischen Ordnung der Wörter in Einklang zu bringen:

```
9.  words = ['die', 'Sonne', 'Sonne', 'scheint', 'scheint']
10. words = list(set(words))
11. words = [ word.lower() for word in words ]
12. words.sort()
13. word_index = dict([ (word, idx) for idx, word in enumerate(words) ])
14. index_word = dict([(idx, word) for word, idx in word_index.items()])
15. word_index, index_word
```

Ausgabe:

```
({'die': 0, 'scheint': 1, 'sonne': 2},
 {0: 'die', 1: 'scheint', 2: 'sonne'})
```

Code-Hinweise:

In *Zeile 10* produzieren wir aus den tokenisierten Wörtern des Textes zunächst ein Set, um die Duplikate zu entfernen. Dieses Set casten wir danach (in der gleichen Zeile) wieder als Liste, weil wir nur Listen sortieren können. In *Zeile 11* wandeln wir die Buchstaben der einzelnen Wörter innerhalb der Liste unter Verwendung einer List Comprehension in Kleinbuchstaben um. Danach (*Zeile 12*) sortieren wir die Liste alphabetisch und aufsteigend mit der sort-Methode.

In *Zeile 13* produzieren wir das erste Dictionary in einem Aufwasch. Die enumerate-Funktion gibt uns bei der Iteration über die Liste der Wörter automatisch Integer-Werte für jedes Element (Wort) zurück. Die Werte beginnen bei 0 und werden bei jedem Iterationsschritt um den Wert 1 inkrementiert. Diese Zahlen verwenden wir, um die Encodierung der Wörter festzulegen. Da die Liste sortiert ist, erhalten wir eine alphabetisch informierte Encodierung. Wie man sieht, arbeiten wir die Wörter mit einer List-Comprehension ab und erzeugen deswegen das Dictionary erst nach Fertigstellung der Liste. Damit das Casting mit *dict* funktioniert, platzieren wir als Elemente der Liste einfach das Wort (*word*) und die zugehörige Ganzzahl (*idx*) in ein Tuple. Bei der Umwandlung in ein Dictionary liest Python das Tuple automatisch als Key-Value-Paar aus.

Bei der Erzeugung des Index-to-Word-Dictionarys in *Zeile 14* gehen wir auf Nummer sicher. Wir lesen das bereits vorhandene Word-to-Index-Dictionary unter Verwendung der items-Methode aus, die sowohl Keys als auch Values zurückgibt. In der List-Comprehension vertauschen wir dann einfach die Positionen der beiden Werte im Tuple und erzeugen daraus ein neues Dictionary.

Mit dem so geschaffenen Dictionary (*word_index*) können wir jetzt weiterarbeiten. Wir können damit zum Beispiel eine einfache Encodierung von Wörtern als Ganzzahlen erzeugen, die – wie wir später sehen werden – für verschiedene Aufgaben nützlich ist. Nehmen wir zum