10 Sequence-to-Sequence-Modelle

Rekurrente Netze haben wir bis jetzt nur in ihrer Funktion als einfache Klassifizierer kennengelernt. Dabei werden zwar Sequenzen als *x*-Daten analysiert, aber einzelne Werte als *y*-Daten geschätzt. Selbst die generativen Modelle, mit denen wir Textsequenzen im Stil Martin Heideggers erzeugt haben, waren so veranlagt: Längere Textsequenzen entstanden nur, weil wir das Modell in einer Schleife immer wieder einzelne Wörter schätzen ließen, die dann aneinandergereiht wurden.

Rekurrente Netze können aber weitaus mehr. Mit ihnen lassen sich problemlos Sequenzen als Ausgaben produzieren. Wenn Sie an die Funktionsweise des rekurrenten Verfahren zurückdenken, werden Sie sich erinnern, dass zu jedem Zeitschritt ein Zustand entsteht. Jeden dieser Zustände können wir theoretisch nutzen, um jeweils einzelne Wörter zu produzieren, sodass als Ausgabe eine Sequenz von Wörtern entsteht.

Leider ist die Architektur effektiver Sequence-to-Sequence-Modelle komplexer, als es sich in der Theorie anhört. Würde man zum Beispiel eine einzelne rekurrente Lernzelle verwenden, um eine Eingabe- in eine Ausgabesequenz zu transferieren, dann entspräche die Sequenzlänge der Eingabe (*x*) immer automatisch der Sequenzlänge der Ausgabe (*y*). Für Aufgaben, bei denen Ein- und Ausgaben typischerweise oder notwendigerweise unterschiedliche Längen aufweisen (zum Beispiel bei Übersetzungen oder Zusammenfassungen von Texten), sind solche Modelle ungeeignet.

Außerdem funktioniert diese Art des Mappings auch nur dann, wenn sich die Ein- und Ausgabesequenzen mehr oder weniger eins zu eins übertragen lassen. Schwierig wird es, wenn Wörter, die im Eingabesatz weiter hinten stehen, in der Ausgabe bereits weiter vorne gebraucht werden. Das ist zum Beispiel bei Übersetzungen der Fall, weil die Übertragung eines Satzes in der Regel nicht Wort für Wort geschehen kann. Oft unterscheidet sich schließlich die Stellung der Satzteile zwischen den zu übersetzenden Sprachen. Zum Beispiel wird aus dem Deutschen *„Die Kellnerin heißt Maria“* im Spanischen je nach Übersetzung zum Beispiel *„El nombre de la camarera es Maria“*. Das Wort *„heißt“* braucht man also für die Übersetzung in *„el nombre“* schon zu Beginn, in der Eingabesequenz wird es aber erst weiter hinten geliefert.

Aus diesen Gründen haben sich für Aufgaben dieser Art spezielle Modelle, sogenannte *Encoder-Decoder*-Architekturen, durchgesetzt. Kennzeichen dieser Modelle ist, dass die Analyse der Eingabesequenz von der Produktion der Ausgabesequenz entkoppelt ist. So lassen sich unabhängig von der Anzahl der Einheiten in der Eingabesequenz Ausgaben beliebiger Länge erzeugen. Außerdem wird die Ausgabe erst produziert, nachdem die Eingabesequenz komplett gelesen wurde. Das hat den Vorteil, dass die Reihenfolge der Setzung von Wörtern in der Ausgabe unabhängig von der Reihenfolge der Eingabesequenz ist. Vor der Produktion der Ausgabe sind schließlich schon alle Teile der Eingabe bekannt.

Wie solche Encoder-Decoder-Modelle aufgebaut sind und wie sie funktionieren, sehen wir uns im Folgenden genauer an. Außerdem werfen wir einen Blick auf den *Attention*-Mechanismus, der einen erheblichen Anteil an der Perfektionierung maschineller Übersetzungen hat. Danach setzen wir das Gelernte wie immer mithilfe von TensorFlow/Keras in die Tat um.

10.1 Encoder-Decoder-Modelle mit Teacher Forcing

Wenn wir uns vorstellen, wie eine Person, die zwei Sprachen beherrscht, einen Satz von der einen in die andere übersetzt, kommen wir womöglich auf die Idee, dass dieser Vorgang in zwei Schritten abläuft. Im ersten Schritt hört oder liest die Person den Satz in der Ursprungssprache. Während das Gehirn Wort für Wort verarbeitet, erzeugt es im Hintergrund eine mehr oder weniger abstrakte Vorstellung der Bedeutung des Satzes. Im zweiten Schritt wird diese abstrakte Vorstellung eingesetzt, um den Satz nach den Regeln der Zielsprache zu rekonstruieren.

Der Vorteil dieser Methode liegt auf der Hand: Die Repräsentation beinhaltet alle relevanten Informationen über die Bedeutung des Ursprungssatzes, sodass der Zielsatz faktisch aus dem Gedächtnis heraus erzeugt werden kann. Da die Repräsentation zudem abstrakt ist, spielt die formale Struktur des Quellsatzes bei der Produktion des Zielsatzes keine Rolle mehr. Das erleichtert die Übersetzung zwischen Sprachen mit sehr unterschiedlichen grammatikalischen Regeln.

Die von Cho et al. [Cho2014] vorgeschlagenen Encoder-Decoder-Architektur kann man sich als eine technische Realisation dieser Idee vorstellen. Der *Encoder* analysiert die Eingabesequenz und produziert den sogenannten *Kontextvektor* – eine abstrakte Fassung der Eingabe. Der *Decoder* verwendet den Kontextvektor des Encoders, um daraus den Zielsatz zu generieren [Ganegedara2018, S. 321].

Aber wie arbeitet nun ein Encoder-Decoder-Modell im Detail? Wie der Name nahelegt, bestehen Encoder-Decoder-Modelle aus zwei technischen Komponenten: dem Encoder und dem Decoder. Dabei handelt es sich streng genommen um zwei unterschiedliche neuronale Netze, die über die Verbindung des Kontextvektors zusammengeschaltet und zusammen trainiert werden.

Sehen wir uns zunächst den einfacheren Teil an: den *Encoder*. Er beinhaltet mindestens einen rekurrenten Layer und erzeugt aus der Eingabe – zum Beispiel einer Textsequenz, die in eine andere Sprache übersetzt werden soll – eine abstrakte Repräsentation. Diese Repräsentation (der Kontextvektor) besteht einfach aus dem finalen Zustand (h) des rekurrenten Layers nach Prozessierung der kompletten Eingabesequenz (vgl. Bild 10.1).

Der *Decoder* ist etwas komplexer organisiert. Er ist für die Produktion der Ausgabesequenz zuständig und besteht deshalb aus mindestens einem weiteren (unabhängigen) rekurrenten Layer und aus einer nachgelagerten vollständig verbundenen Schicht, die die Schätzwerte (z. B. Wörter) produziert. Der rekurrente Layer des Decoders empfängt zunächst als Eingabe den Kontextvektor aus dem Encoder. Der Kontextvektor fungiert dabei allerdings nicht als klassischer Input für die Decoder-RNN, er wird stattdessen als *initialer Zustand* in die Decoder-RNN eingepflanzt.

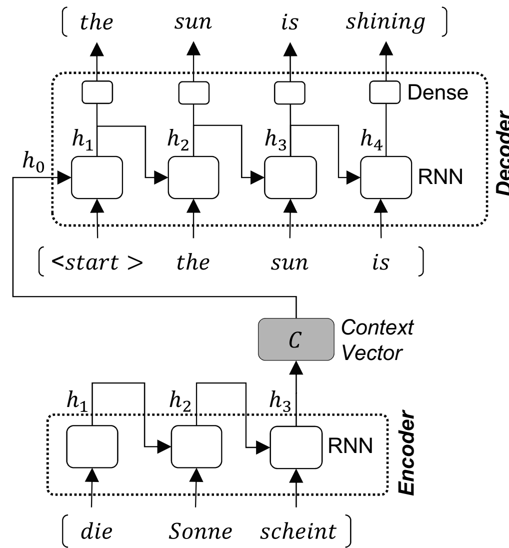


Bild 10.1 Encoder-Decoder-Modell. Der Encoder erzeugt einen Kontextvektor, der den Eingabesatz repräsentiert. Aus dem Kontextvektor und dem jeweils vorhergehenden Wort als Input schätzt der Decoder die Ausgabesequenz. Die Ausgabe des RNN des Decoders wird dazu zu jedem Zeitschritt einer vollständig verbundenen Schicht übergeben, die die Wortschätzungen mit einer Softmax-Aktivierung durchführt.
Hinweis: Die RNN-Schichten sind in der Darstellung jeweils in ihrer ausgerollten Form dargestellt.

Wie kann man sich das vorstellen? Wenn wir uns den Aufbau einer rekurrenten Schicht noch einmal ins Gedächtnis rufen, erinnern wir uns, dass der initiale Zustand der RNN (h_0) normalerweise auf 0 gesetzt wird. Das liegt daran, dass vor dem ersten Input aus der Eingabesequenz keine Informationen über den Zustand der Sequenz zur Verfügung stehen. Der Zellzustand wird ja erst mit dem ersten Input berechnet und fließt von dort aus jeweils in die Berechnung der nachfolgenden Zustände des RNN ein. Ein Decoder-RNN, der eine Eingabesequenz in eine Ausgabesequenz transformiert, verfügt jedoch über Vorinformationen in Form des Kontextvektors des Encoders. Er beinhaltet die transformierten Bedeutungen der Eingabesequenz und ist für die Produktion der Ausgabesequenz unerlässlich. Also übergeben wir vor der ersten Produktion einer Einheit (durch die Decoder-RNN) den Kontextvektor als initialen Zustand (h_0). Auf diese Weise kann der rekurrente Layer das erste Wort der Sequenz auf Grundlage der Informationen aus dem Encoder produzieren.

Wenn allerdings der Kontextvektor als Initialzustand des rekurrenten Decoder-Layers fungiert, was erhält die Decoder-RNN dann als Input? Dafür gibt es verschiedene Möglichkeiten. Die eine besteht darin, jeweils einfach Aufforderungen zur Produktion von Zielwerten zu übergeben, die darüber hinaus keine inhaltliche Bedeutung haben (z. B. 0, 1, 2, 3 ...). Das wäre zwar möglich, allerdings würden man dadurch wertvolle Informationen verschwenden. Schließlich ist die Decoder-RNN so eingestellt, dass sie bei jedem Zeitschritt eine einzelne Ausgabe (z. B. ein Wort) produziert. Es liegt also nahe, dem Decoder seinen vorhergehenden Schätzwert beim nächsten Zeitschritt wieder als Eingabe zur Verfügung zu stellen, sodass er sich daran erinnert, welches Wort bereits vorhanden ist und auf Grundlage des bereits vorhandenen Wortes Informationen über das nächste Wort ableiten kann.

Nehmen wir an, wir wollen den deutschen Satz *„die Sonne scheint“* in den englischen Satz *„the sun is shining“* übertragen. Der Decoder erhält als Eingabe also zum einen den vom Encoder produzierten Kontextvektor, der die vollständig analysierte Version des deutschen Satzes repräsentiert. Gleichzeitig erhält er als Input beim ersten Zeitschritt ein arbiträres Symbol, das nur den Anfang der Sequenz markiert, zum Beispiel <*start*> (vgl. Bild 10.1). Davon ausgehend und mit dem Kontextvektor ausgerüstet, schätzt die Decoder-RNN-Schicht zusammen mit einer nachgeschalteten vollständig verbundenen Schicht das nächste Wort bzw. das erste Wort des zu produzierenden englischen Satzes: *„the“*. Jetzt können wir mit der zeitversetzten Fütterung beginnen. Das Wort *„the“* wird nun als Eingabe verwendet und auf dieser Basis (zusammen mit dem aktualisierten Zustand aus dem vorherigen Zeitschritt) das nachfolgende Wort der Sequenz geschätzt: *„sun“*. Die Prozedur wird so lange wiederholt, bis die komplette Ausgabesequenz fertiggestellt ist.

Dieses Verfahren, bei dem man die RNN jeweils mit der vorher produzierten Ausgabe füttert, nennt sich *Teacher Forcing* [Goodfellow2016, S. 372]. Während des Anlernprozesses (solange die y-Ausgabesequenzen noch bekannt sind) wird der rekurrente Layer dabei nicht mit den geschätzten, sondern mit den tatsächlichen (zeitversetzten) Werten gefüttert. Erst wenn das Modell zur Schätzung von Zielwerten herangezogen wird, fließen die Schätzwerte des vorherigen Zeitschritts ($t-1$) jeweils als Eingaben des aktuellen Zeitschritts (t) ein.

10.2 Attention-Mechanismus

Der Vorteil des Encoder-Decoder-Modells ist zugleich dessen Nachteil: Der Kontextvektor enthält zwar eine abstrakte Fassung der gesamten Eingabesequenz. Die darin enthaltenen Informationen sind allerdings stark komprimiert. Insbesondere wenn die Eingabe- oder Ausgabesequenz aus einer langen Reihe besteht, fällt es dem Decoder zunehmend schwer, alle notwendigen Informationen für die Reproduktion aus dem Gedächtnis des Kontextvektors zu extrahieren.

Aus diesem Grund haben Bahdanau et al. [Bahdanau2015] ein Verfahren entwickelt, bei dem die Ausgabeschicht des Decoders, die für die Produktion der Sequenz zuständig ist, jeweils weitere spezifische Informationen aus dem Encoder erhält. Das Verfahren nennt sich *Attention-Mechanismus*.

Die Grundidee dieses Verfahrens ist wie bei vielen Deep Learning-Verfahren an die Spezifika der Lernaufgabe angelehnt. Wenn wir zum Beispiel bei einer Übersetzung einen Satz von der einen in die andere Sprache übertragen, können wir aufgrund abweichender Satzbauregeln zwar meistens nicht Wort für Wort übersetzen. Trotzdem gehen die Grundbedeutungen einzelner Wörter in der Regel auf spezifische andere Wörter zurück oder leiten sich im Zweifelsfall aus einigen wenigen Wörtern der Quellsprache ab. An unserem Beispiel *„Die Kellnerin heißt Maria“*, aus dem im Spanischen *„El nombre de la camarera es Maria“* wird, kann man das gut beobachten. Die Wörter *„die“*, *„Kellnerin“* und *„Maria“* können mehr oder weniger eins zu eins in *„la“*, *„camarera“* bzw. *„Maria“* übertragen werden. Bei *„heißt“* ist es schon etwas komplizierter, obwohl das Substantiv *„nombre“* den größten Bedeutungsteil

übernimmt. Da *„heißt“* nicht einfach als Verb übertragen wird, müssen in der Satzbildung die Präpositionen *„de“* und das Verb *„es“* eingeführt werden. Sehen wir aber einmal von dieser Komplikation ab, können wir immerhin drei der vier Wörter aus dem Ursprungssatz ohne Umschweife übersetzen.

Wenn wir davon ausgehen, dass solche oder ähnliche Konstellationen ziemlich häufig auftreten, dann wäre es bei der Produktion eines einzelnen Wortes im Decoder jeweils hilfreich zu wissen, auf welches Wort in der Eingabesequenz jetzt die Aufmerksamkeit liegen sollte. Genau diese Strategie verfolgt das Attention-Prinzip. Bei der Erzeugung der einzelnen Einheiten in der Ausgabe wird jetzt nicht nur der finale Zustand des Encoder-RNN zur Verfügung gestellt. Zusätzlich werden Informationen aus den einzelnen Zeitschritten, die bei der Analyse der Eingabe in der Encoder-RNN entstehen, mobilisiert. Welche Informationen dabei wichtig sind, wird im Zuge des Anlernprozesses entschieden. Wenn alles klappt, bekommt die vollständig verbundene Schicht, die den Ausgabesatz produziert, zu jedem Zeitschritt einen Zustandsvektor, der genau jene Zustandsinformationen aus der Encoder-RNN enthält, die für die Erzeugung des zu diesem Zeitschritt relevanten Wortes am interessanten sind (vgl. Bild 10.2).

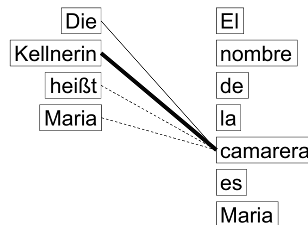


Bild 10.2 Schematische Darstellung der Arbeit des Attention-Mechanismus. Dargestellt ist die Produktion des Wortes *„camarera.“* Dafür sind aus der Eingabesequenz insbesondere Informationen aus der Verarbeitung des zweiten Inputs (*„Kellnerin“*) interessant und werden deshalb besonders stark gewichtet.
Hinweis: Die Stärke der Linien symbolisiert die Stärke des Einflusses eines Wortes aus dem Quellsatz auf die Produktion eines Wortes im Zielsatz.

Technisch betrachtet wird für jedes Zielwort (y_i) der Ausgabesequenz ein separater Kontextvektor (c_i) angelernt, der die Produktion der Ausgabe für das Wort *i* informiert. Obwohl die Intention klar ist, ist die Umsetzung des Attention-Prinzips kompliziert. Das System speist sich nämlich aus zwei unterschiedlichen Quellen:

Zum einen wie beschrieben aus den einzelnen Zellzuständen der *Encoder-RNN* (h_j), die Informationen über jede Position (j) der Eingabesequenz liefert (auch Annotationen genannt). Man muss dabei an dieser Stelle noch einmal darauf hinweisen, dass nicht der Input j, sondern der Zustand, der durch die Verarbeitung des Inputs j im RNN-Layer entsteht (h_j), gemeint ist. Man kann jedoch davon ausgehen, dass der Zustand h_j jeweils auf den aktuellen Input (j) fokussiert ist.

Zum anderen greift der Attention-Mechanismus auf die einzelnen Zellzustände der *Decoder-RNN* (s_i) zurück, die auch als Grundlage zur Schätzung der Zielvariablen im einfachen Sequence-to-Sequence-Modell herangezogen werden. Während des Trainings werden dann Gewichtungsfaktoren (a_{ij}) angelernt, die über den Einfluss des jeweiligen Zustands (h_j) innerhalb der Eingabesequenz auf das zu schätzende Wort in der Ausgabe (i) entscheiden [Bahdanau2015, S. 3].

10.3 Encoder-Decoder-Architekturen in der Praxis

Im Folgenden sehen wir uns an zwei Beispielen an, wie sich Encoder-Decoder-Modelle mit TensorFlow/Keras zusammenstellen, anlernen und zur Schätzung von Sequenzen einsetzen lassen. Im ersten Schritt stellen wir unter Verwendung der funktionalen API ein einfaches Sequence-to-Sequence-Modell basierend auf dem Encoder-Decoder-Schema von Cho et al. [Cho2014] zusammen. Danach implementieren wir in einem etwas aufwendigeren und leistungsfähigeren Modell den Attention-Mechanismus nach Bahdanau et al. [Bahdanau2015].

10.3.1 Ein einfaches Encoder-Decoder-Modell

Wir wollen im Folgenden die Aufgabe der Rechtschreibkorrektur, die wir in den vorhergehenden Kapiteln bereits mehrfach bearbeitet haben, ein letztes Mal aufgreifen, um sie mithilfe eines Sequence-to-Sequence-Modells einer Lösung zuführen. Zur Erinnerung: Die Aufgabe besteht darin, aus falsch geschriebenen Wörtern die zugrunde liegenden richtigen Wörter vorherzusagen. Dass wir dazu ein Sequence-to-Sequence-Modell verwenden, ist nicht besonders weit hergeholt: Schließlich haben wir in den bisherigen Ansätzen die Eingabesequenz (die falsch geschriebenen Wörter) Buchstabe für Buchstabe mit einem rekurrenten oder konvolutionalen Layer bearbeitet. Warum sollten wir also nicht auch die Zielvariable (die korrekten Wörter) als Sequenzen behandeln und Buchstabe für Buchstabe schätzen?

Wenn wir die korrekten Wörter als feste Einheiten definieren, wie wir das bis jetzt getan haben, brauchen wir in der Ausgabeschicht unserer Modelle jeweils genauso viele Neuronen, wie wir Wörter schätzen möchten. In unserem Beispiel haben wir nur mit einem Spieldatensatz, der aus 500 Wörtern besteht, gearbeitet. Daher sind wir auch auf keine größeren Probleme gestoßen. Bei einem großen Korpus von 100 000 oder mehr Wörtern wird dieser Ansatz jedoch deutlich ressourcenintensiver, und sowohl der Prozess des Anlernens als auch der des Schätzens verschlingt deutlich mehr Zeit und Speicherplatz. Im Gegensatz dazu ist das Sequence-to-Sequence-Modell sehr sparsam. Wenn wir statt ganzer Wörter die Sequenzen der Buchstaben schätzen, aus denen sich die Wörter zusammensetzen, benötigen wir in der Ausgabeschicht nur genauso viele Neuronen wie Buchstaben im Alphabet existieren.

10.3.1.1 Vorbereitung der Daten

Um ein Encoder-Decoder-Modell mit dem Teacher Force-Verfahren anzulernen, benötigen wir neben den *x*- und *y*-Daten, die als Input für den Encoder bzw. als Zielvariablen fungieren, eine weitere, speziell präparierte Datendatei. Dabei handelt es sich um den Input für den Decoder-Teil (vgl. Bild 10.1).

Zur Erinnerung: Der Decoder wird beim Anlernen mit dem *Teacher Force*-Verfahren nicht nur mit dem finalen Zustand des Encoders gefüttert (dem Kontextvektor), sondern zusätzlich mit der um einen Zeitschritt (Buchstaben) nach rechts verschobenen Sequenz der Buchstaben, aus denen das korrekte Wort besteht. Er erhält also zur Schätzung des aktuellen Buchstabens jeweils den vorhergehenden Buchstaben des korrekten Wortes. Da bei der Schätzung des ersten Buchstabens kein vorhergehender Buchstabe existiert, füttern wir die RNN an dieser

Stelle mit dem immer gleichen Sonderzeichen, das als Startbotschaft fungiert. Den ersten Buchstaben muss der Decoder-RNN also komplett aus dem Gedächtnis des Kontextvektors heraus schätzen.

Den Decoder-Input erstellen wir also einfach aus den *y*-Daten, indem wir die Originaldaten um eine Einheit nach rechts verschieben und an die erste Position jeweils das Tab-Zeichen ('\t') setzen, das die Aufgabe eines Startsymbols übernimmt. Die Umarbeitung erledigen wir mit einer Funktion, die wie folgt aussieht:

```
1.  def insert_start_sign(words: list, start='\t'):
2.     new_words = []
3.     for word in words:
4.        new = start + word
5.        new_words.append(new)
6.     return new_words
7.
8.  X1 = X
9.  X2 = insert_start_sign(y)
10. X1[:5], X2[:5], y[:5]
```

Ausgabe:

```
( ['jeddes', 'verlchivdene', 'zwnächst', 'gang', 'aßso'],
  ['\tjedes', '\tverschiedene', '\tzunächst', '\tging', '\talso'],
  ['jedes', 'verschiedene', 'zunächst', 'ging', 'also'] )
```

Code-Hinweise:

Wir laden für diese Aufgabe die Spelling-Daten, die wir in Kapitel 6 produziert haben. In *X* bzw. *X1* sind die fehlerhaften, in *y* die korrekten Wörter enthalten. Die neue Variable *X2* enthält die um eine Einheit nach rechts und mit dem Startsymbol „\t“ versehenen korrekten Buchstabenfolgen.

Wie wir in der Ausgabe sehen, stehen uns jetzt zum Anlernen drei Datendateien zur Verfügung: die inkorrekten Wörter (*X1*), die um eine Einheit nach rechts und mit dem Startzeichen verschobenen korrekten Wörter (*X2*) und die korrekten Wörter ohne Verschiebung (*y*).

Als Nächstes müssen wir die Wörter der drei Dateien in numerischer Form als Sequenzen darstellen. Das erledigen wir wie gewohnt, indem wir die Wörter jeweils als sequenzielle One-Hot-Arrays encodieren. Wie das genau geht, haben wir in Kapitel 8 zum Thema rekurrente Netze im Detail besprochen. Wir müssen jetzt nur darauf achten, dass die Decoder-Inputs (*X2*) und die Zielvariable (*y*) die gleiche Länge haben. Schließlich produziert die Decoder-RNN, wenn wir uns die Zustände zu jedem Zeitschritt ausgeben lassen, für jeden einzelnen Input genau eine Ausgabe, die wir zur Schätzung der Zielbuchstaben einsetzen wollen. Wir verwenden also das längste Wort aus *X2* (da diese Wörter aufgrund der Rechtsverschiebung immer länger sind als die Wörter in *y*) und bestimmen darüber die Länge der Sequenz für alle Wörter. Wenn ein Wort weniger Buchstaben umfasst als die Codiersequenz Positionen beinhaltet (was der Normalfall sein wird), füllen wir die überschüssigen Positionen mit einem weiteren Sonderzeichen (in diesem Fall verwenden wir dazu das Symbol für einen Zeilenumbruch „\n“): Es zeigt an, dass die Sequenz zu Ende ist:

```
11. def words_to_char_matrix( words: list,
12.                           char_dic: dict,
13.                           max_len: int,
14.                           end_sign='\n'):
15.    x = np.zeros( shape=(len(words), max_len, len(char_dic)),
16.                  dtype='int32')
17.    for idx, word in enumerate(words):
18.       for i, char in enumerate(word):
19.          x[idx, i, char_dic[char]] = 1
20.       for i in range(len(word), max_len):
21.          x[idx, i, char_dic[end_sign]] = 1
22.    return x
23.
24. char_dic = dict([(char, i) for i, char
25.                 in enumerate(
26.                 list('.abcdefghijklmnopqrstuvwxyzäöüß\n\t'))
27.                  ])
28.
29. X1_ = words_to_char_matrix(X1, char_dic, 15)
30. X2_ = words_to_char_matrix(X2, char_dic, 15)
31. y_ = words_to_char_matrix(y, char_dic, 15)
32. X1_.shape, X2_.shape, y_.shape
```

Ausgabe:

```
((15921, 15, 33), (15921, 15, 33), (15921, 15, 33))
```

Nach der Umwandlung liegen uns drei dreidimensionale NumPy-Arrays vor, die für jeden Datensatz jeweils ein zweidimensionales Array der Länge *15 × 33* beinhalten (*Anzahl der Buchstaben des längsten Wortes × Encodierung der Buchstaben als One-Hot-Set*).[1] Dass die Sequenzlänge von *X1_* (Input Encoder) und *X2_* bzw. *y* (Input Decoder bzw. Zielvariable) gleich ausfällt, ist dabei Zufall. Theoretisch könnte die Anzahl der Einheiten zwischen der Eingabe in den Encoder und der Eingabe in den Decoder auch abweichen, da die Sequenzen im Modell später von unterschiedlichen RNN-Layern verarbeitet werden. Die Länge der Eingabesequenzen von Decoder und der Ausgabesequenz müssen dagegen aufeinander abgestimmt sein, weil sie von der gleichen RNN-Einheit verarbeitet bzw. produziert werden.

Was die Encodierung von Input und Output betrifft, gilt, dass sowohl für die beiden Inputs (Encoder/Decoder) als auch für die Zielvariable je eigene One-Hot-Encodierungen möglich gewesen wären. In unserem Fall weichen die vorkommenden Zeichen jedoch nur im Hinblick auf das Startsymbol ab (wird nur im Input des Decoders verwendet), sodass es aus Gründen der Übersichtlichkeit naheliegt, für alle Daten nur ein einziges Dictionary zur Encodierung zu verwenden.

Damit sind die Trainingsdaten vorbereitet. Wie ein einzelner Datensatz aussähe, wenn wir die One-Hot-Arrays in Buchstabensequenzen zurückübersetzten, zeigt das Beispiel unten für das falsche Wort *„jeddes“*, das auf das Zielwort *„jedes“* trainiert wird:

[1] Das One-Hot-Set setzt sich zusammen aus den 30 Kleinbuchstaben des deutschen Alphabets (plus des Satzzeichens „.“). Hinzu kommen das Sonderzeichen für das Startsymbol („\t“) und das Symbol, das das Ende einer Sequenz bezeichnet („\n“).

```
Input Encoder: [j,  e, d, d, e, s, \n, \n, \n, \n, \n, \n, \n, \n, \n]
Input Decoder: [\t, j, e, d, e, s, \n, \n, \n, \n, \n, \n, \n, \n, \n]
Zievariable:   [j, e, d, e, s, \n, \n, \n, \n, \n, \n, \n, \n, \n, \n]
```

10.3.1.2 Aufbau des Encoder-Decoder-Modells

Im nächsten Schritt bauen wir das neuronale Netz auf, das sich zum Anlernen der Sequenzen eignet. Wir bedienen uns dabei der funktionalen API, weil wir zwei Inputs benötigen. Außerdem müssen wir ja den Zustand des Encoders nach der vollständigen Prozessierung der Eingabesequenz als initialen Zustand des Decoders übergeben. Die Architektur des Modells stellen wir uns dabei vor wie in Bild 10.3 veranschaulicht.

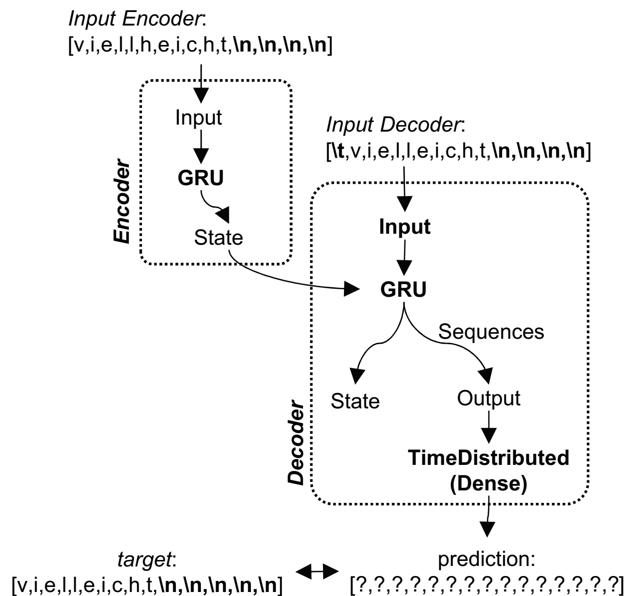


Bild 10.3 Aufbau eines Sequence-to-Sequence-Modells mit TensorFlow/Keras.

Der *Encoder* besteht aus einer Eingabeschicht, der die One-Hot-encodierten Sequenzen aufnimmt (die mit Rechtschreibfehlern behafteten Wörter) und an eine Gated Reccurent Unit (GRU) weiterleitet. Die GRU verarbeitet die Eingabe Schritt für Schritt und erzeugt nach Erhalt des letzten Buchstabens einen Zustand, den wir als *Kontextvektor* im Decoder-Modell weiterverwenden.

Der *Decoder* besteht wie der Encoder aus einem Input Layer, der den präparierten Decoder-Input aufnimmt. Der Input wird dann an eine rekurrente Schicht (wiederum eine GRU) weitergeleitet und dort verarbeitet. Zusätzlich erhält der Decoder den letzten Zustand aus dem Encoder als initialen Zustand. Im Vergleich zur GRU des Encoders stellen wir die GRU des Decoders allerdings so ein, dass sie die Eingabesequenz nicht erst komplett verarbeitet und dann einen Zustand an die nachgelagerte Schicht weiterleitet. Wir benötigen ja deren Zustände nach Verarbeitung jeden einzelnen Buchstabens der Input-Sequenz. Damit können wir dann unter Verwendung eines nachgelagerten Dense Layers den Zielbuchstaben an jeder Position schätzen.

Da die Eingabe Buchstabe für Buchstabe verarbeitet wird, erhält die Decoder-GRU ganz zu Beginn also jeweils als Input den Startbefehl *„\t“*. Auf dieser Grundlage und ausgerüstet mit dem Kontextvektor muss sie nun einen Zustand erzeugen, aus dem der nachgeschaltete Dense Layer den ersten Buchstaben des korrekten Wortes schätzen kann. Danach erhält die rekurrente Schicht den richtigen ersten Buchstaben als Input und erzeugt auf dieser Grundlage und mithilfe des vorherigen Zustands einen neuen Zustand, den der nachgeschaltete Dense Layer zur Produktion des zweiten Buchstabens verwendet. Dieser Prozess wird so lange wiederholt, bis die komplette Ausgabesequenz erzeugt wurde.

Kommen wir nun noch zur Ausgabeschicht, die in Bild 10.3 als *TimeDistributed* bezeichnet ist. Die Klasse *TimeDistributed* bezeichnet in Keras einen Wrapper, mit dem sich zum Beispiel eine Dense-Schicht mehrfach hintereinander zur Produktion von Schätzwerten einsetzen lässt. Die Klasse geht davon aus, dass sie von der zuvor geschalteten Schicht mehrfach nacheinander aufgerufen wird. Da wir eine rekurrente Schicht vorgelagert haben, die zu jedem Zeitschritt eine Ausgabe erzeugt, werden diese Ausgaben automatisch als Aufforderung zur Aktivierung interpretiert. Der Dense Layer wird also in unserem Fall 15-mal nacheinander aufgefordert auf Grundlage eines neuen Zustands, den die Decoder-RNN anliefert, einen Schätzwert zu erzeugen.

Das ist alles. Die Ausgebeschicht müssen wir jetzt nur mit 33 Neuronen und einer Softmax-Aktivierung bestücken, sodass sie bei jedem Aufruf einen der Buchstaben schätzen kann. Wenn wir das Ganze mit TensorFlow/Keras umsetzen, sieht der Code wie nachfolgend abgedruckt aus:

```
33. from tensorflow.keras.models import Model
34. from tensorflow.keras.layers import Input, GRU, Dense, TimeDistributed
35.
36. units= 100
37. encoder_in = Input(shape=(None, len(char_dic)), name='encoder_in')
38. encoder_state = GRU( units=units,
39.                      name='encoder_gru')(encoder_in)
40.
41. decoder_in = Input(shape=(None, len(char_dic)), name='decoder_in')
42. decoder_gru = GRU( units=units,
43.                    return_sequences=True, return_state=True,
44.                    name='decoder_gru')
45. gru_out, gru_state = decoder_gru( decoder_in,
46.                                   initial_state=encoder_state)
47. dense = Dense(units=len(char_dic), activation='softmax')
48. decoder_out = TimeDistributed( dense,
29.                                name='time_distributed')(gru_out)
50.
51. model = Model([encoder_in, decoder_in], decoder_out )
52.
53. model.compile( loss='categorical_crossentropy',
54.                optimizer='adam',
55.                metrics=['accuracy'])
56. model.summary()
```

Ausgabe:

```
____________________________________________________________________________
Layer (type)                 Output Shape         Param #     Connected to
============================================================================
encoder_in (InputLayer)      [(None, None, 33)]   0
____________________________________________________________________________
decoder_in (InputLayer)      [(None, None, 33)]   0
____________________________________________________________________________
encoder_gru (GRU)            (None, 100)          40500      encoder_in[0][0]
____________________________________________________________________________
decoder_gru (GRU)            [(None, None, 100),  40500      decoder_in[0][0]
                                                             encoder_gru[0][0]
____________________________________________________________________________
time_distributed (TimeDist) (None, None, 33)     3333       decoder_gru[0][0]
============================================================================
Total params: 84,333
Trainable params: 84,333
Non-trainable params: 0
```

Code-Hinweise:

Zunächst importieren wir alle notwendigen Klassen (*Zeilen 33–34*). Danach legen wir den Parameter *units* an, der die Anzahl der Neuronen in den RNN-Schichten von Encoder und Decoder festlegen soll (*Zeile 36*). Die Anzahl muss übereinstimmen, weil wir ja den Encoder-State auf den Decoder-State übertragen wollen (wir pflanzen den Kontextvektor des Encoders als *Initial State* in die Decoder-RNN ein.

In den *Zeilen 37–39* setzen wir den *Encoder* auf. Er besteht aus einem Input Layer, dessen Struktur durch eine noch unbekannte Sequenzlänge (*None*, warum, sehen wir später) und durch die bekannte Anzahl der Buchstaben pro Sequenzeinheit (33) gekennzeichnet ist. Danach kommt eine Standard-GRU-Schicht, die die Eingabe verarbeitet und am Ende einen Zustand produziert, den wir als Kontextvektor verwenden.

Der *Decoder* (*Zeilen 41–49*) ist etwas komplexer veranlagt. Der Input Layer ist gleich aufgebaut wie der Input Layer des Encoders. Danach kommt die Decoder-GRU. Sie verfügt über die gleiche Anzahl an Neuronen wie die Encoder-GRU, allerdings lassen wir uns von ihr sowohl die Zustände zu jedem Zeitschritt ausgeben (*return_sequences=True*) als auch den Zustand am Ende des letzten Zeitschritts in einem separaten Array (*return_state=True*). Den Zustand benötigen wir zwar erst im Inferenz-Modell (mit dem wir die Schätzungen durchführen), müssen ihn an dieser Stelle aber ausgeben lassen, um ihn später verfügbar zu haben.

In *Zeile 47* legen wir dann einen Standard-Dense-Layer an, den wir für die Schätzung der 33 Buchstaben einer jeden Sequenzeinheit ausstatten und mit Softmax aktivieren. Allerdings fügen wir diese Schicht nicht einfach ins Modell ein, sondern wickeln sie in den oben beschriebenen *TimeDistributed*-Wrapper, der die Dense-Schicht für jede Ausgabe der vorherigen GRU einmal aufruft, um Schätzwerte zu produzieren.

Dann sind wir fertig und müssen das Modell nur noch zusammenfügen (*Zeile 51*). Der Graph besteht am Eingang aus den beiden Input-Layern, über die wir die Daten für den Encoder und Decoder anliefern (*encoder_in, decoder_in*) und am Ausgang aus der Ausgabe des TimeDistributed bzw. des Dense Layers (*decoder_out*).

Den Rest kennen wir. Wir kompilieren das Modell mit einer passenden Verlustfunktion und stellen den Optimierer ein. Da wir in der Ausgabeschicht mit Softmax arbeiten, die Daten zu jedem Zeitschritt aber als One-Hot-Array anliefern, verwenden wir als Verlustfunktion nicht wie sonst *sparse_categorical_crossentropy*, sondern greifen auf die Variante *categorical_crossentropy* zurück. ■

In der Ausgabe kann man noch einmal die Grundanlage des Modells besichtigen. Interessant ist in erster Linie die Decoder-GRU, die zum einen mit den Decoder-Inputs und zum anderen mit dem Encoder-State verdrahtet ist.

Jetzt können wir das Modell anlernen. Dazu rufen wir wie gehabt die fit-Methode auf, wobei wir zuvor noch zwei Callbacks für die Steuerung des Trainings zusammenstellen. Da es sich nur um ein Spielbeispiel handelt, stellen wir außerdem den Parameter *validation_split* auf 0.2, sodass Keras für uns die Separierung von Trainings- und Testdaten übernimmt:

```
57. from tensorflow.keras.callbacks import ( EarlyStopping,
58.                                          ModelCheckpoint )
59.
60. stopping = EarlyStopping( monitor='val_loss',
61.                           patience=3,
62.                           restore_best_weights=True)
63. checkpoint = ModelCheckpoint( filepath='SeqToSeq_spelling.h5',
64.                               monitor='val_loss',
65.                               save_best_only=True)
66. History = model.fit([X1_, X2_], y_,
67.                      epochs=50,
68.                      batch_size=32,
69.                      validation_split=.2,
70.                      callbacks=[stopping, checkpoint])
```

Nach ca. 45 Epochen beendet das EarlyStopping-Callback den Anlernprozess. Dabei erreichen wir auf den Validierungsdaten eine Accuracy von 98,8 %. Doch Vorsicht! Die Verwendung der Accuracy als Kennzahl liefert bei dieser Aufgabe einen ziemlich geschönten Blick auf die Realität. Sie sagt nicht etwa aus, dass wir 98,8 % richtige Wörter produzieren. Vielmehr produzieren wir im Durchschnitt in einem Wort 98,8 % richtige Buchstaben. Da klingt zwar immer noch ziemlich viel, wird aber dadurch relativiert, dass viele Wörter ziemlich kurz sind, das heißt, zu einem großen Teil aus den Symbolen bestehen, die das Ende des Wortes markieren („*\n*“). Davon abgesehen enthalten auch schon die falschen Wörter ziemlich viele richtige Buchstaben. Wenn unser Netz also einfach die falschen Wörter aus dem Encoder-Input reproduzierte, würden wir bereits eine Accuracy von 87 % erreichen!

Das heißt aber natürlich nicht, dass unser Modell notwendigerweise schlecht ist. Nur müssen wir uns einen gesonderten Test überlegen, der ein realistischeres Bild ergibt. Das ist aber

nicht so schwer. Schließlich könnten wir ja einfach Schätzungen mit den Gesamtdaten erzeugen und dann einen einfachen Algorithmus entwickeln, der nur Sequenzen als richtig zählt, die vollständig mit den Sequenzen der Zielvariablen übereinstimmen. Alle anderen werden als falsch gewertet. Wenn wir diesen Ansatz verfolgen, kommen wir immerhin noch auf eine Korrektklassifizierungsrate von 94,1 %.

10.3.1.3 Das Inferenzmodell aufbauen und einsetzen

Wir haben nun also ein Encoder-Decoder-Modell aufgebaut und angelernt, das unseren Ansprüchen für diese Aufgabe genügt. Das Problem ist nur, dass wir mit diesem Modell keine Schätzungen durchführen können. Wie sollte das auch gehen? Der Decoder erwartet als Eingabe schließlich eine Sequenz von Buchstaben, die das richtige Wort bereits beinhalten. Das liegt uns aber natürlich nicht vor, wenn wir das Modell in der Produktion einsetzen wollen. Was also tun?

Wir müssen aus dem bestehenden Modell mit den angelernten Gewichten ein spezielles Inferenzmodell extrahieren und neu zusammensetzen. Der Unterschied zwischen dem Anlernmodell und dem Inferenzmodell besteht im Wesentlichen darin, dass wir den Encoder-Teil und den Decoder-Teil jetzt als separate Modelle konzipieren. Im ersten Schritt füttern wir dann den Encoder mit dem falschen Wort als Eingabe und erhalten als Ausgabe den Kontextvektor. Damit ist die Arbeit des Encoders im Schätzprozess erledigt. Im zweiten Schritt füttern wir den Decoder in einer Schleife jeweils zum einen mit dem Encoder-State bzw. mit dem durch die Decoder-RNN aktualisierten Encoder-State und zum anderen mit einer Input-Sequenz der Länge 1. Schließlich stehen uns zu Beginn des Prozesses jeweils nur das Startzeichen und der Kontextvektor zur Verfügung. Das ist im Übrigen auch der Grund, weshalb wir das Trainingsmodell nicht auf eine feste Sequenzlänge eingestellt haben. Da wir den *shape*-Parameter der Input-Sequenz des Decoders auf *None* eingestellt haben, ist es kein Problem, wenn wir im Training Sequenzen der Länge 15 und bei der Inferenz Sequenzen der Länge 1 übergeben (vgl. Codeblock oben).

Zurück zum Prozedere: Im ersten Umlauf liefern wir dem Decoder also den Kontextvektor und das Startsymbol („*\t*“) an. Der Decoder weiß nun, dass er den ersten Buchstaben des Zielwortes erzeugen soll. Als Ausgabe erhalten wir daher den ersten geschätzten Buchstaben. Gleichzeitig müssen wir den Zustand der Decoder-GRU nach der Produktion des ersten Buchstabens abfangen. Schließlich hat sich der Eingangszustand, den wir aus dem Encoder bezogen haben (der Kontextvektor), geändert, da wir in der Decoder-Sequenz um einen Buchstaben vorgerückt sind. Also brauchen wir den aktuellen Zustand der Decoder-GRU, um damit und mit dem vom Decoder erzeugten ersten Buchstaben als Input den Decoder aufs Neue zu füttern.

Im nächsten Schleifendurchlauf produziert der Decoder auf dieser Grundlage den nächsten Buchstaben bzw. den nächsten Zustand. Beide fangen wir wieder ab und starten damit den nächsten Decoder-Zyklus. Als Resultat erhalten wir den nächsten Buchstaben und den nächsten Zustand usw. Diesen Kreislauf halten wir so lange aufrecht, bis entweder die maximale Länge der Sequenz erreicht ist oder der Decoder ein Endzeichen („*\n*“) als Ausgabe produziert. Dann wissen wir, dass das zu schätzende Wort komplett erzeugt wurde.

Wenden wir uns aber zunächst der Konstruktion der Encoder- und Decoder-Modelle aus dem angelernten Trainingsmodell zu:

```
1.  from tensorflow.keras.models import load_model
2.  model = load_model('SeqToSeq_spelling.h5')
3.
4.  # 1) Encoder Modell
5.  encoder_in = model.get_layer('encoder_in').input
6.  encoder_state = model.get_layer('encoder_gru').output
7.
8.  encoder_model = Model(encoder_in, encoder_state)
9.
10. # 2) Decoder-Modell
11. units = 100
12. decoder_state_input = Input(shape=(units,), name='decoder_state_in')
13. decoder_inputs = model.get_layer('decoder_in').input
14. decoder_gru = model.get_layer('decoder_gru')
15. decoder_time = model.get_layer('time_distributed')
16.
17. decoder_outputs, decoder_state = decoder_gru(decoder_inputs,
18.                                  initial_state=decoder_state_input)
19. decoder_outputs = decoder_time(decoder_outputs)
20.
21. decoder_model = Model( [decoder_inputs, decoder_state_input],
22.                        [decoder_outputs, decoder_state])
```

Code-Hinweise:

In *Zeile 2* laden wir zunächst das angelernte Gesamtmodell (falls es nicht bereits im Arbeitsspeicher liegt). Danach steht die Zusammenstellung des *Encoder-Modells* auf dem Programm. Es besteht nur aus dem Input und dem Output der Encoder-GRU und lässt sich unter Abruf des entsprechenden Layers aus dem Gesamtmodell einfach zusammenstellen und als eigenständiges Modell aufbauen (*Zeile 5–8*).

Interessanter ist der *Decoder* (*ab Zeile 11*). Wenn wir ihn als eigenständiges, vom Encoder abgekoppeltes Modell erstellen wollen, benötigt er zusätzlich eine Eingabeschicht für den Anfangszustand der Decoder-GRU. Also erzeugen wir einen neuen Input-Layer mit der erforderlichen Struktur, den wir zur Übergabe nutzen können (*Zeile 12*).[2] Danach beschaffen wir uns den bereits bestehenden Decoder-Input-Layer, über den wir die geschätzten Buchstaben füttern können (*Zeile 13*). Außerdem brauchen wir die angelernte Decoder-GRU und die Ausgabeschicht (*TimeDistributed, Zeile 14–15*).

In *Zeile 17–18* bauen wir die extrahierten Schichten dann zu einem eigenständigen Decoder-Modell zusammen. Wir bestücken die Decoder-GRU zum einen mit der Eingabe für die zeitversetzten Buchstaben (*decoder_inputs*), zum anderen mit

[2] Dass wir für die Ausgabe eines anderen Netzes einen Input Layer brauchen, mag auf den ersten Blick etwas verwunderlich sein. Allerdings behandelt ein neuronales Netz alles, was von außen kommt und verarbeitet werden soll, gleichberechtigt und verlangt dafür die Anlage einer Input-Schicht, in der die Struktur dieser Eingabe definiert ist.

der Eingabeschicht, die für den initialen Zustand vorgesehen ist (*decoder_state_input*). Als Rückgabe erhalten wir zwei Ausgaben: erstens die Ausgabe der Decoder-GRU (hier *decoder_outputs*), die für jede Sequenzeinheit einen Zustand beinhaltet, und zweitens den Zustand des Decoders am Ende der Sequenz (*decoder_state*). Den sequenziellen Output des Decoders (*decoder_outputs*) verwenden wir als Eingabe in den TimeDistributed-Wrapper. Er produziert unter Verwendung der vollständig verbundenen Ausgabeschicht die finalen Schätzwerte des Decoders (einen Buchstaben).

Zum Schluss stellen wir das Decoder-Modell zusammen (*Zeilen 21–22*). Die Eingabe besteht jetzt neben der Eingabesequenz aus der Eingabe des initialen Zustands, für den wir eine eigene Schicht angelegt haben. Die Ausgabe besteht aus der finalen Ausgabe des Decoder-Teils (*decoder_outputs*), der die Wahrscheinlichkeit für die einzelnen Buchstaben enthält, und aus dem aktuellen Zustand des Decoders, nachdem er die Eingabe verarbeitet hat. Diesen Zustand brauchen wir, weil wir damit beim nächsten Schätzvorgang zusammen mit dem Buchstaben das Decoder-Modell füttern werden. ■

Wir verfügen jetzt also über ein separates Encoder- und über ein separates Decoder-Modell. Beide sind mit den Gewichten des trainierten Gesamtmodells ausgestattet und können in Kombination zur Schätzung von korrekten Wörtern aus falschen Wörtern herangezogen werden. Um den Schätzprozess durchzuführen, brauchen wir allerdings noch eine Routine, die die Arbeit der Datenverarbeitung übernimmt. Im Kern müssen wir bei einem einzelnen Schätzvorgang folgende Teilaufgaben erledigen:

a) Ein falsch geschriebenes Wort (ein String) wird in eine Sequenz der Länge 15, in der jeder Buchstabe nach dem oben skizzierten System in ein One-Hot-Array kodiert wird, übertragen.

b) Das Array übergeben wir der predict-Methode des Encoders zur Produktion des Kontextvektors.

c) Eine Eingabesequenz für den Decoder mit der Länge 1 wird erzeugt. Sie enthält das Startzeichen („*\t*“) in der One-Hot-Encodierung.

d) Die Decoder-Eingabe wird zusammen mit dem Kontextvektor des Encoders dem Decoder-Modell zur Schätzung des ersten Buchstabens übergeben (predict-Methode). Als Ausgabe erhalten wir den Decoder-Output (den ersten Buchstaben) und den aktuellen Zustand der Decoder-GRU.

e) Den geschätzten Buchstaben sammeln wir und übergeben ihn danach der predict-Methode des Decoders (in der geforderten Codierung) zusammen mit dem aktuellen Zustand der Decoder-GRU. Diesen Prozess wiederholen wir in einer Schleife so lange, bis die maximale Länge der Sequenz erreicht ist oder das Endzeichen („*\n*“) als Ausgabe geschätzt wird.

Die Funktion *predict_correct_word* übernimmt diese Aufgabe. Sie nimmt als Eingabe ein falsch geschriebenes Wort und liefert als Rückgabe das zugrundeliegende korrekte Wort bzw. eine Kombination von Buchstaben, von denen jeder einzelne die höchsten Einzelwahrscheinlichkeiten bei der Schätzung erreicht:

```
23. def predict_correct_word( misspelled: str, encoder, decoder,
24.                           char_dict: dict, len_seq=15):
25.
26.    input_seq = words_to_char_matrix([misspelled], char_dict, len_seq)
27.    index_char_dict = dict([(i, char) for
28.                            char, i in char_dict.items()])
29.
30.    state = encoder.predict(input_seq)
31.    decoder_seq = np.zeros(shape=(1, 1, len(char_dict)), dtype='int32')
32.    decoder_seq[0, 0, char_dict['\t']] = 1
33.
34.    word = ''
35.    for i in range(len_seq):
36.       output_char, dec_state = decoder.predict(
37.                                 [decoder_seq] + [state])
38.       char_index = np.argmax(output_char[0, 0])
39.       char = index_char_dict[char_index]
40.       if char == '\n':
41.          return word
42.       word += char
43.
44.       decoder_seq = np.zeros(shape=(1, 1, len(char_dict)), dtype='int32')
45.       decoder_seq[0, 0, char_index] = 1
46.       state = dec_state
47.
48.    return word
49.
50. misspelled = 'vielheiicht'
51. predict_correct_word(misspelled, encoder_model, decoder_model, char_dic)
```

Ausgabe:

```
'vielleicht'
```

Code-Hinweise:

Die Funktion *predict_correct_word* benötigt, um arbeiten zu können, neben dem falsch geschriebenen Wort (als String) die Encoder- und Decoder-Modelle und ein Dictionary zum Nachschlagen der Indexpositionen von Buchstaben. Außerdem muss es die Länge der Eingabe- bzw. Ausgabesequenz kennen (da in unserem Fall die Längen gleich sind, weisen wir nur einen Parameter zu).

Die eigentliche Arbeit beginnt in *Zeile 26*, wo wir die Eingabe mithilfe der oben beschriebenen Funktion *word_to_char_matrix* in eine Sequenz übertragen, die für jeden Buchstaben ein One-Hot-Array vorsieht. In *Zeile 27–28* erzeugen wir aus dem übergebenen Wort-Index-Dictionary ein Reverse-Dictioary, das wir zum Nachschlagen von Buchstaben bei gegebenen Integer-Werten benötigen (Output des Decoders).

In *Zeile 30* rufen wir dann die predict-Methode des Encoders unter Übergabe des encodierten fehlerhaften Wortes auf und erhalten als Rückgabe den Kontextvektor (*state*).

In *Zeile 31* und *32* bereiten wir die Eingabe für den Decoder vor (*decoder_seq*). Sie besteht aus einer Sequenz der Länge 1, die nur das Startzeichen (*„\t“*) beinhaltet.

Danach beginnt der sich wiederholende Part des Schätzprozesses. Dazu öffnen wir eine Schleife, in der wir den Decoder nacheinander mit Eingaben füttern und Ausgaben (einzelne Buchstaben) zurückerhalten. Im ersten Durchlauf rufen wir die predict-Methode des Decoders mit der zuvor präparierten Eingabe (*decoder_seq*) und dem Kontextvektor des Encoders (*state*) auf. Als Rückgabe erhalten wir die Wahrscheinlichkeiten für den ersten Buchstaben (Sequenz der Länge 1, da wir eine Sequenz der Länge 1 gefüttert haben, *output_char*). Außerdem erhalten wir den Zustand der Decoder-GRU nach dem ersten Durchlauf (*dec_state*).

Um zu wissen, welchen Buchstaben der Decoder geschätzt hat, müssen wir die Ausgabe interpretieren: Zunächst lassen wir uns die Indexposition des One-Hot-Sets mit der höchsten Wahrscheinlichkeit ausgeben (*Zeile 38*). Dann schlagen wir diese Indexposition im Reverse-Dictionary nach und erhalten den geschätzten Buchstaben (*Zeile 39*).

Bevor wir weiterarbeiten, prüfen wir, ob es sich bei dem vorhergesagten Buchstaben um ein Endzeichen (*„\n“*) handelt. Wenn ja, geben wir den bis dahin decodierten String zurück und brechen die Funktion ab (*Zeilen 40–41*). Wenn nein, fügen wir den geschätzten Buchstaben dem dekodierten Wortteil, das wir in *Zeile 34* – unmittelbar vor Start der Schleife – als leeren String angelegt haben, hinzu (*Zeile 42*).

Danach bereiten wir alles für den nächsten Durchlauf der Schleife vor. Wir erzeugen einen neuen leeren Decoder-Input und füllen ihn mit dem gerade geschätzten Buchstaben (*Zeile 44* und *45*). In *Zeile 46* weisen wir dann noch den aktuellen Zustand des Decoders (*dec_state*) der Variablen *state* zu, die wir beim nächsten Durchlauf der Schleife in *Zeile 37* wieder als initialen Decoder-Zustand übergeben. Danach wird die Schleife aufs Neue ausgeführt: Die predict-Methode wird mit dem gerade erzeugten Buchstaben und dem aktuellen Zustand des Decoders gefüttert usw. ■

Wie wir an der Ausgabe erkennen, funktionieren unsere Modellzusammensetzung und die Routine, die die Datenverarbeitung organisiert. Bei Einspeisung des Beispielworts *„vielheiicht“* erhalten wir das korrekte Wort *„vielleicht“* zurück.

10.3.2 Encoder-Decoder-Modelle mit Attention-Mechanismus

Das einfache Encoder-Decoder-Modell ist eine elegante Architektur zur Erzeugung von Sequenzen. Wir haben im vorherigen Abschnitt gesehen, wie sich damit Eingaben variabler Länge in Ausgaben variabler Länge transformieren lassen. Das Problem dieser Architektur besteht darin, dass der Kontextvektor schnell zum Nadelöhr für die Reproduktion der Ausgabesequenz wird. Wenn Eingabe- und Ausgabesequenzen eine bestimmte Länge überschreiten, wird es für den Decoder schwer, allein aus dem Zustand des Encoders am Ende der Sequenz die Ausgabe zu reproduzieren.

Zur Bearbeitung dieses Problems wollen wir jetzt den *Attention-Mechanismus* einführen. Wie oben beschrieben, handelt es sich dabei um einen speziellen Layer, der nützliche Informationen einzelner Sequenzaspekte der Eingabe für die Produktion einzelner Wörter in der Ausgabe bereitstellt. Welche Informationen das sind, wird im Zuge des Trainings gelernt.

Sehen wir uns den Aufbau eines solchen Modells mit Attention-Mechanismus zunächst genauer an. Diesmal wollen wir eine Architektur aufbauen, die sich für Übersetzungen eignet: Wir wollen deutsche Sätze ins Englische übertragen.

In Bild 10.4 ist der grobe Aufbau skizziert. Zunächst zum *Encoder*: Der erste Unterschied zum Encoder aus dem einfachen Sequence-to-Sequence-Modell besteht darin, dass wir jetzt eine Embedding-Schicht aufsetzen, um die Wörter der Eingabe zu vektorisieren. Diese Änderung steht allerdings nicht im Zusammenhang mit dem Einsatz der Attention Layer. Die zweite Änderung dagegen schon: Aus der Encoder-GRU lassen wir uns jetzt nicht nur den Zustand am Ende des eingelesenen Satzes ausgeben (den Kontextvektor), sondern auch die einzelnen Zustände nach Verarbeitung der einzelnen vektorisierten Wörter.

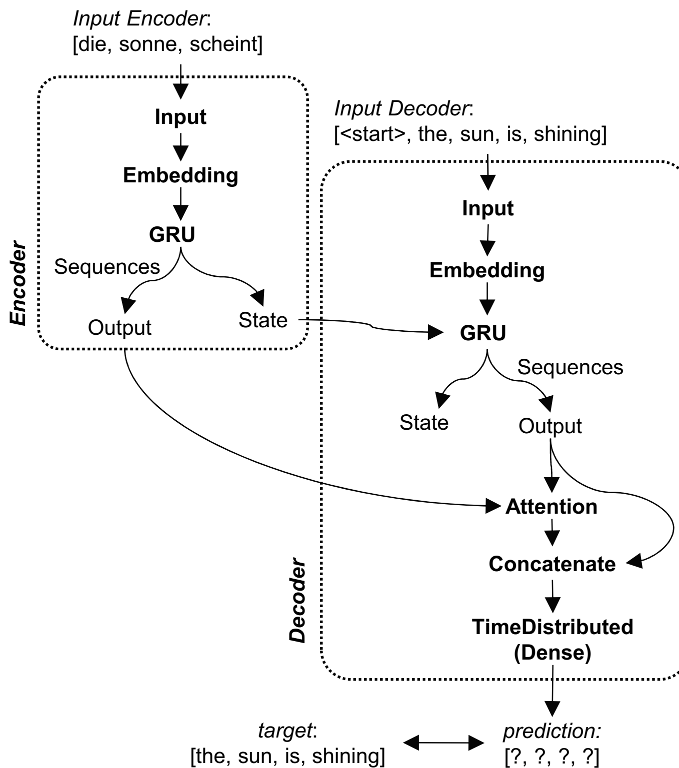


Bild 10.4 Architektur eines Encoder-Decoder-Modells mit Attention-Mechanismus in TensorFlow/Keras.

Kommen wir jetzt zum *Decoder*. Auch er verfügt über eine Embedding-Schicht, die die um eine Einheit nach rechts verschobenen *y*-Daten (die englischen Sequenzen) als Eingabe in Empfang nimmt und vektorisiert. Danach folgt die rekurrente-Schicht (GRU). Sie wird, genau wie bei der einfacheren Sequence-to-Sequence-Architektur, mit dem Endzustand der Encoder-

GRU als Anfangszustand gefüttert und liefert wiederum sowohl den Zustand nach Analyse der kompletten Sequenz als auch nach Analyse jeder einzelnen Einheit zurück.

Danach folgt die eigentliche Neuerung: der *Attention Layer*. Er benötigt zwei Eingaben, *zum einen* die Sequenzen aus der Encoder-GRU – das sind die auf einzelne Wörter fokussierten Informationen aus der Eingabe (die Annotationen) –, *zum anderen* die sequenziellen Ausgaben aus der Decoder-GRU. Aus diesen beiden Informationen wird mittels anzulernender Gewichte eine Ausgabematrix erzeugt, in der für jedes zu generierende Wort die relevantesten Informationen aus den verarbeiteten Eingabesequenzen hinterlegt sind.

Im letzten Schritt geht es dann um die Schätzung der einzelnen Wörter des Zielsatzes. Dazu verwenden wir wie üblich einen Dense Layer mit Softmax-Aktivierung eingewickelt in einen TimeDistributed-Wrapper. Allerdings können wir dieser Schicht nicht nur den Output der Attention Layer zur Schätzung der Wörter anbieten (darin sind nur die relevanten Informationen aus der Encoder-GRU enthalten), wir brauchen auch die Ausgaben der Zustände der Decoder-GRU. Daher verbinden wir die beiden Matrizen mithilfe einer Konkatenierungsschicht und übergeben die erweiterte Matrix – die beide Informationen enthält – der Ausgabeschicht zur Berechnung der jeweiligen Schätzwerte.

10.3.2.1 Vorbereitung der Daten

Die Implementierung des Attention-Mechanismus ändert nichts am Grundaufbau des Modells und an dessen Ein- und Ausgängen: Es handelt sich immer noch um einen Encoder-Decoder, der mit Teacher Forcing angelernt wird.

Wir benötigen also, um das Modell trainieren zu können, drei Arten von Daten: Deutsche Sätze für den Encoder (*X1*); die um einen Zeitschritt nach rechts verrückten korrespondierenden englischen Sätze als Eingabe für den Decoder (*X2*) und die einfachen englischen Sätze als Zielvariablen für den Ausgang (*y*).[3] Die Rohdaten (Listen von Strings) von *X1* und *y* liegen uns bereits vor. Bei *X2* handelt es sich streng genommen einfach um eine Version von *y*, bei der wir vor das erste Wort ein Startsymbol (<*start*>) platzieren. Das erledigen wir mit einer simplen Funktion, die wir gemäß ihrer Aufgabe *insert_start_sign* nennen:

```
1.  def insert_start_sign( data: list, start='<start>'):
2.     new_data = []
3.     for phrase in data:
4.        phrase = start + " " + phrase
5.        new_data.append(phrase)
6.     return new_data
7.
8.  X2 = insert_start_sign(y)
9.  X1[:4], X2[:4], y[:4]
```

Ausgabe:

```
(['Geh.', 'Hallo!', 'Grüß Gott!', 'Lauf!'],
 ['<start> Go.', '<start> Hi.', '<start> Hi.', '<start> Run!'],
 ['Go.', 'Hi.', 'Hi.', 'Run!'])
```

[3] Die Daten für diese Aufgabe (Sätze, die auf Englisch und Deutsch vorliegen) haben wir über folgende Quelle bezogen: *http://www.manythings.org/anki/*. Dabei wurde die Liste der deutschen Sätze um jene gekürzt, die 20 Wörter oder mehr umfassen.

Da wir im Encoder und Decoder jeweils mit einer Embedding-Schicht arbeiten (vgl. Bild 10.4), liefern wir die Wörter als Sequenzen von Integern an. Zur Transformation verwenden wir (wie in Abschnitt 8.3 ausgeführt) zwei Keras-Tokenizer: den einen zur Tokenisierung und Encodierung der deutschen Wörter, den anderen für die englischen Wörter:

```
10. from tensorflow.keras.preprocessing.text import Tokenizer
11. from tensorflow.keras.preprocessing.sequence import pad_sequences
12.
13. filters = '!"#$%&()*+,-./:;=?@[\\]^_`{|}~\t\n'
14. tok_en, tok_de = Tokenizer(filters=filters), Tokenizer(filters=filters)
15.
16. tok_de.fit_on_texts(X1)
17. tok_en.fit_on_texts(X2)
18.
19. X1_ = tok_de.texts_to_sequences(X1)
20. X2_ = tok_en.texts_to_sequences(X2)
21. y_ = tok_en.texts_to_sequences(y)
22.
23. X1_pad = pad_sequences( X1_, maxlen=19,
24.                         padding='post', truncating='post')
25. X2_pad = pad_sequences( X2_, maxlen=25,
26.                         padding='post', truncating='post')
27. y_pad = pad_sequences( y_, maxlen=25,
28.                        padding='post', truncating='post')
29. X1_pad.shape, X2_pad.shape, y_pad.shape
```

Ausgabe:

```
((220672, 19), (220672, 25), (220672, 25))
```

Code-Hinweise:

Die Textdaten liegen als Rohstrings vor (*X1, X2*). Die Wörter werden zunächst unter Verwendung der Keras Tokenizer-Klasse analysiert. Dabei wird im Hintergrund ein Wort-Dictionary aufgebaut, in dem alle vorkommenden Token mit einer Integer hinterlegt werden (Aufruf der *fit_on_texts*-Methode). Nach Aufruf der *texts_to_sequences*-Methode erhalten wir für jeden Satz eine Liste, in der die Wortsequenzen tokenisiert und die Token als Integer encodiert sind.

Danach geht es noch um das Zuschneiden der Sequenzen, sodass sie eine feste Länge haben. Das erledigt die *pad_sequences*-Funktion unter Übergabe eines Längenparameters (*maxlen*). Sätze, die kürzer als die voreingestellte Länge sind, werden automatisch mit dem Integer-Wert *0* aufgefüllt, der im Wort-Dictionary des Tokenizers unbelegt bleibt.

Das ist die grundlegende Verfahrensweise. Davon abweichend müssen wir aber einige Anpassungen vornehmen, damit alles reibungslos funktioniert. In *Zeile 13* definieren wir einen String *filters*, der die Zeichen enthält, die der Tokenizer herausschneiden und ignorieren soll. Wir übergeben sie bei der Instanziierung der Tokenizer-Klasse unter dem gleichnamigen Parameter. Dabei handelt es sich einfach um eine abgewandelte Form der Default-Einstellung des *filters*-Parameters,

aus der wir die Vergleichszeichen (<>) entfernt haben (weil wir sie im <*start*>-Token verwenden).

Dabei wird auch deutlich, dass der Tokenizer der Keras-Klasse standardmäßig alle Satzzeichen ignoriert. Das ist für ernst zu nehmende Textanwendungen natürlich nicht tragbar, da die Satzzeichen Informationen über die Struktur eines Textes beinhalten und als eigene Token behandelt werden sollten. In solchen Fällen macht es deshalb durchaus Sinn, statt des Keras-Tokenizers eine eigene Vorverarbeitungsklasse zu schreiben, die bei der Umwandlung nach eigens definierten Kriterien verfährt. An dieser Stelle geht es allerdings nur um die Demonstration des Verfahrens. Um die Vorverarbeitung nicht komplizierter als nötig zu machen, arrangieren wir uns deshalb mit den Eigenheiten der Keras-Klasse.

In den *Zeilen 23–28* wandeln wir die Integer-Listen dann in NumPy-Arrays gleicher Länge um. Hier gilt es Folgendes zu beachten: Die Eingaben in den Encoder und in den Decoder können ohne Weiteres im Hinblick auf die Länge der Sequenzen variieren. Allerdings muss der Decoder-Input mit der Länge der Ausgabe des Gesamtnetzes (und deshalb auch der Labels) übereinstimmen. Außerdem müssen wir bei der Wahl der Länge der Sequenzen darauf achten, dass wir die Ein- und Ausgabesätze nicht willkürlich abschneiden. Wir können nur vollständige deutsche Phrasen, für die eine vollständige englische Übersetzung vorliegt, zum Training verwenden. Zurechtgestutzte Eingabe- oder Ausgabesequenzen verhindern effektives Lernen. Die hier verwendeten Werte von 19 (*Encoder Input*) bzw. 25 (*Decoder Input* und *Output*) entsprechen den maximalen Längen der Beispieldaten. ■

Insgesamt liegen uns fürs Training 220 672 Beispielsätze vor. Das ist natürlich zu wenig, um damit einen produktionsreifen Übersetzungsalgorithmus zu trainieren. Es genügt aber, um zu demonstrieren, wie ein Übersetzungsmodell mit Attention-Mechanismus aufgesetzt und angelernt werden kann und wie sich damit nach dem Anlernen Übersetzungen erzeugen lassen.

10.3.2.2 Zusammenstellung des neuronalen Netzes

Bei der Umsetzung des Modells orientieren wir uns an Bild 10.4. Die Grundzüge der Anlage haben wir bereits im vorherigen Abschnitt besprochen, in dem es um den Aufbau eines einfachen Encoder-Decoder-Modells ging. Wir legen also jetzt den Fokus auf die Veränderungen, die nötig sind, um den Attention-Mechanismus zu implementieren.

Was den Attention Layer betrifft, der den von Bahdanau et al. [Bahdanau2015] vorgeschlagenen Mechanismus umsetzt, können wir leider auf keine Keras-interne Lösung setzen.[4] Stattdessen verwenden wir eine Kreation, die im Internet kursiert.[5] Da die programmierte Schicht von der Klasse *Layer* aus *tensorflow.keras.layers* erbt, verhält sie sich wie eine Standard-Keras-Schicht und kann auch so verwendet werden. Sie erhält zwei Eingaben und produziert zwei Ausgaben: Die erste Eingabe stammt wie oben beschrieben von der Encoder-RNN, die zweite von der Decoder-RNN. Die erste Ausgabe beinhaltet den Output, den wir zur Produktion

---

[4] Der Layer unter der Bezeichnung *Attention*, der derzeit verfügbar ist, ist leider für rekurrente Netze ungeeignet.

[5] Die Schicht, die wir benutzen, wurde von *Thushan Ganegedara* entworfen und ist verfügbar unter *https://github.com/thushv89/attention_keras*. Wir arbeiten damit, gehen auf die Implementierung im Detail aber nicht ein, da das den Rahmen dieser Einführung sprengen würde.

der Schätzwerte einsetzen: Dabei handelt es sich um die gewichtete Zusammenstellung der Encoder-Annotationen, die für die Erzeugung eines bestimmten Wortes an einer bestimmten Position von Bedeutung sind [Bahdanau2015, S. 3]. Die zweite Rückgabe beinhaltet die angelernten Gewichte des Attention Layers, die darüber bestimmen, mit welchem Gewicht eine bestimmte Encoder-Annotation für die Schätzung einer bestimmten Wortposition im Decoder herangezogen wird.

Die Zusammenstellung des Modells selbst gestaltet sich sehr ähnlich wie die Zusammenstellung des einfachen Encoder-Decoders. Allerdings entsteht durch die vorgelagerten Embedding-Schichten, durch den Einsatz einer bidirektionalen GRU im Encoder (die Bahdanau et al. [Bahdanau2015] vorschlagen) und durch den Attention Layer eine komplexere Grundstruktur, die sich in einem deutlichen Mehr an Code und atypischen Datenflüssen niederschlägt:

```
30. def get_attention_model( seq_len_encoder: int,
31.                          seq_len_decoder: int,
32.                          num_words_encoder: int,
33.                          num_words_decoder: int,
34.                          vec_dim = 64, hidden_size = 64):
35.
36.    encoder_inputs = Input( shape=(seq_len_encoder,),
37.                            name='encoder_inputs')
38.    decoder_inputs = Input( shape=(seq_len_decoder,),
39.                            name='decoder_inputs')
40.    # Encoder
41.    encoder_emb = Embedding( input_dim=num_words_encoder+1,
42.                             output_dim=vec_dim,
43.                             name='encoder_emb')(encoder_inputs)
44.    encoder_gru = Bidirectional( GRU(hidden_size,
45.                                 return_sequences=True,
46.                                 return_state=True),
47.                                 name='encoder_bi_gru')
48.    encoder_out, encoder_ffw_state, encoder_bw_state = \
49.                                 encoder_gru(encoder_emb)
50.    # Decoder
51.    decoder_emb = Embedding( input_dim=num_words_decoder+1,
52.                             output_dim=vec_dim,
53.                             name='decoder_emb')(decoder_inputs)
54.    decoder_gru = GRU( hidden_size*2, return_sequences=True,
55.                       return_state=True, name='decoder_gru')
56.    encoder_state = Concatenate(axis=-1,
57.                    name='concate_encoder')([encoder_ffw_state,
58.                                             encoder_bw_state])
59.    decoder_out, decoder_state = decoder_gru(decoder_emb,
60.                                 initial_state=encoder_state)
61.    # Attention layer
62.    attn_layer = AttentionLayer(name='attention_layer')
63.    attn_out, attn_states = attn_layer([encoder_out, decoder_out])
64.
65.    decoder_concat = Concatenate( axis=-1,
66.                     name='concat_layer')([decoder_out, attn_out])
67.    dense = Dense( num_words_decoder+1,
68.                   activation='softmax', name='softmax_layer')
```

```
69.     dense_time = TimeDistributed( dense,
70.                                   name='time_distributed_layer')
71.     decoder_pred = dense_time(decoder_concat)
72.     # Model
73.     model = Model( inputs=[encoder_inputs, decoder_inputs],
74.                    outputs=decoder_pred)
75.     model.compile( optimizer='adam',
76.                    loss='sparse_categorical_crossentropy',
77.                    metrics=['accuracy'])
78.     return model
79.
80. model = get_attention_model( seq_len_encoder=X1_pad.shape[1],
81.                              seq_len_decoder=X2_pad.shape[1],
82.                        num_words_encoder=len(tok_de.word_index),
83.                        num_words_decoder=len(tok_en.word_index) )
```

Code-Hinweise:

Da die Zusammenstellung des Modells bzw. des Anlernmodells aufwendiger als sonst ist, legen wir dafür die Funktion *get_attention_model* an. Die notwendigen Parameter sollten selbsterklärend sein: Die Sequenzlängen der Eingaben für den Encoder und Decoder und die Anzahl der Wörter der Quell- und Zielsprache, die wir für die Einstellungen der Embedding-Schichten und der Ausgabeschicht benötigen.

Los geht es in den *Zeilen 36–39* mit der Definition der Inputs für den Encoder und Decoder. Dabei stellen wir den *shape*-Parameter jeweils auf die Länge der Sequenzen ein (19 bzw. 25).

In den *Zeilen 40–49* setzen wir dann den Hauptteil des *Encoders* zusammen, zuerst den Embedding Layer und danach die GRU-Schicht. Dabei halten wir uns an das von Bahdanau et al. [Bahdanau2015] vorgeschlagene Modell und konzipieren die GRU als bidirektionalen Layer, der den Quellsatz von vorne nach hinten und parallel von hinten nach vorne analysiert. Außerdem müssen wir diese Schicht so einstellen, dass sie neben den finalen Zuständen auch die Zustände für die einzelnen Zwischenschritte ausgibt (*return_sequences=True*). Am Ende erhalten wir dann (und etwas überraschend) *drei Ausgaben* aus dem Encoder-Part: neben der Matrix mit den Zuständen zu jedem Zeitschritt (*encoder_out*) in getrennten Matrizen die finalen Ausgaben nach Abarbeitung der kompletten Sequenz für die von vorne nach hinten (*encoder_ffw_state*) und von hinten nach vorne analysierten Sequenzen (*encoder_bw_state*).

Damit sind wir mit dem Encoder fertig und können uns dem *Decoder* zuwenden (*Zeilen 50–71*). Den Input haben wir bereits weiter oben definiert, also starten wir auch hier mit dem Embedding-Layer, der die um eine Einheit nach rechts versetzten englischen Sätze aufnimmt und die Wörter vektorisiert (*Zeile 51–53*). Danach kommt die Decoder-GRU (*Zeilen 54–55*), die natürlich nicht bidirektional sein darf, weil wir sonst die Wörter bereits vor der Produktion kennen müssten. Dabei ist zu beachten, dass die Anzahl der Neuronen auf die Anzahl der Neuronen der

Encoder-GRU abgestimmt sein muss – schließlich produziert jedes Neuron bei jedem Zeitschritt genau einen Zustand *h*. Da sich durch die Bidirektionalität der Encoder-GRU die Anzahl der voreingestellten Neuronen automatisch verdoppelt hat, müssen wir in der einfachen Decoder-GRU von vorherein die doppelte Anzahl an Neuronen voreinstellen (*hidden_size**2*). Nur so können wir den initialen Zustand von der Encoder-GRU auf die Decoder-GRU übertragen.

Bevor wir dann die Decoder-GRU an die verschiedenen Eingänge (Embedding und finalen Encoder-State) anschließen, müssen wir noch einen Zwischenschritt gehen: Wie oben beschrieben, gibt die bidirektionale Encoder-GRU die finalen Zustände in getrennten Arrays für die Vorwärts- und Rückwärtsbewegung der Analyse aus. Wir können dem *initial_state*-Parameter der Decoder-GRU allerdings keine zwei Matrizen übergeben, sondern müssen die beiden Ausgaben zuvor mithilfe einer Konkatenierungsschicht zusammenschließen (*Zeilen 56–58*). Die Decoder-GRU selbst produziert dann wieder die beiden aus dem einfachen Modell bekannten Ausgaben: den Output, der für jede Einheit der Sequenz einen Zustand enthält (*decoder_out*), und der Zustand am Ende der Verarbeitung der gesamten Sequenz (*decoder_state*).

Den sequenziellen Output (*decoder_out*) verwenden wir im Folgenden gleich zweifach. Zum einen fließt er zusammen mit dem sequenziellen Output der Encoder-GRU (*encoder_out*) in den *Attention Layer* ein (*Zeilen 62–63*). Der Attention Layer berechnet daraus die für die Produktion der Ausgabe relevanten Teile aus den Annotationen der Encoder-GRU. Als Ausgabe erhalten wir diese Informationen (*attn_out*) und die Gewichte des Attention Layers (*attn_state*), die den Einfluss jeder Eingabeposition auf die Produktion jeder Ausgabeposition enthalten (diese Information benötigen wir zwar im weiteren Verlauf nicht mehr, können damit aber gegebenenfalls die Arbeitsweise des Attention Layers bildlich darstellen, siehe unten).

Mit der Ausgabe des Attention Layers allein können wir allerdings keine Schätzungen durchführen. Deshalb verwenden wir die Ausgabe der Decoder-GRU (*decoder_out*) ein zweites Mal: diesmal als Eingabe für die Dense-Schicht, die die finalen Schätzwerte zu jedem Zeitschritt produziert. Da wir für diese Aufgabe allerdings sowohl diese Informationen als auch den Output aus dem Attention Layer benötigen, verbinden wir die beiden Matrizen mithilfe einer weiteren Konkatenierungsschicht (*Zeile 65–66*). Zum Abschluss setzen wir wie gewohnt einen Dense Layer auf und verwenden den bereits bekannten *TimeDistributed*-Wrapper, um zu jedem Zeitschritt eine separate Ausgabe zu generieren (*Zeilen 69–71*).

In den *Zeilen 73–74* instanziieren wir den aufgebauten Graphen dann noch als Keras-Modell. Dazu übergeben wir die beiden Eingabe-Schichten (*encoder_inputs, decoder_inputs*) als Inputs und die Ausgabe der Dense-Schicht, die die Schätzwerte produziert (*decoder_pred*), als Output.

Zuletzt kompilieren wir das Modell, um es für den Trainingseinsatz vorzubereiten (*Zeilen 75–77*). Auch das sollte inzwischen keine Überraschung mehr sein. Die zu schätzenden Wörter liefern wir als Integer-Werte in einer Variablen an, also verwenden wir *Sparse Categorical Crossentropy* als Verlustfunktion. ■

Um das Modell aufzusetzen, bestücken wir einfach die Parameter der Funktion *get_attention_model* mit den Angaben, die wir aus dem Tokenizer und den Testdaten ziehen. Die beiden Tokenizer beinhalten Informationen über die Wörter, die in den Quell- bzw. Zieldaten vorkommen (wobei uns hier nur deren Anzahl interessiert). Anhand der Länge der einzelnen Datensätze können wir dagegen die Sequenzlängen der Ein- und Ausgaben festlegen. Die restlichen Parameter lassen wir auf den Voreinstellungen (Anzahl Vektoren der Embedding-Schicht und Anzahl Neuronen der GRU-Schichten im Encoder und Decoder).

10.3.2.3 Anlernen des Modells

Das resultierende Modell ist schon etwas größer. Es verfügt über mehr als 7,6 Millionen trainierbare Parameter. Das liegt natürlich insbesondere an den beiden Embedding-Schichten und an der Ausgabeschicht, die mehr als 16 000 verschiedene Wörter schätzen muss. Das Training dauert daher mit einem herkömmlichen Rechner etwas länger.

Der Vollständigkeit halber nachfolgend noch einmal der Anstoß des Trainingsprozesses im Detail:

```
84. from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
85. stopping = EarlyStopping( monitor='val_loss',
86.                           patience=3, restore_best_weights=False)
87. checkpoint = ModelCheckpoint( filepath='SeqToSeq_bi_attention.h5',
88.                               monitor='val_loss',
89.                               save_best_only=True)
90.
91. history = model.fit( [ X1_pad, X2_pad], y_pad,
92.                      epochs=20,
93.                      batch_size=128,
94.                      validation_split=.1,
95.                      callbacks=[stopping, checkpoint])
```

Nach ca. 15 Epochen erreichen wir auf den Trainingsdaten eine Accuracy von ca. 97,5 %. Wie im Abschnitt oben dargelegt, bezieht sich dieser Wert allerdings nicht auf die effektiv komplett richtig geschätzten Sequenzen, sondern lediglich auf den durchschnittlichen Anteil richtiger Wörter in einer Sequenz. Bei den Validierungsdaten liegt der Wert noch deutlich geringer. Das hat allerdings weniger mit unserem Modell als mit der geringen Anzahl an Beispieldatensätzen zu tun, die wir für das Training zur Verfügung haben. Wenn wir einen ernsthaften Übersetzungsalgorithmus anlernen wollten, müssen wir größere Datenmengen beschaffen.[6]

10.3.2.4 Aufbau des Inferenzmodells

Kommen wir jetzt zum Aufbau bzw. zur Rekonstruktion des Inferenzmodells aus dem trainierten Gesamtmodell. Wie bei der einfachen Encoder-Decoder-Architektur ohne Attention, müssen wir wieder zwei getrennte Architekturen zusammenstellen, um den Prozess der Übersetzung deutscher in englische Sätze abwickeln zu können: den Encoder und den Decoder. Verdeutlichen wir noch einmal kurz, welche Aufgabe den beiden Komponenten im Schätzprozess zukommt, um alles richtig zusammenzustecken:

[6] An der Verfügbarkeit solcher Daten herrscht dabei kein Mangel, eher an den Rechenkapazitäten, um solche Modelle in überschaubaren Zeiträumen anzulernen. Wer Modelle mit größeren Daten anlernen möchte, sei zum Beispiel auf die Korpusse des europäischen Parlaments verwiesen, die unter *http://www.statmt.org/europarl/* zum Download verfügbar sind.

Der Encoder hat zunächst die Aufgabe, die komplette Sequenz im Deutschen einzulesen, mit der bidirektionalen RNN-Schicht zu analysieren und als Ausgabe den Kontextvektor (für die Decoder-GRU) und die um die einzelnen Wörter fokussierten Zustände (für den Attention Layer) zu generieren. Im zweiten Schritt müssen wir dann in einer Schleife den Decoder mehrfach aufrufen und mit den jeweils aktuellen Zuständen und den bereits generierten englischen Wörtern füttern, um damit die Ausgabesequenz Wort für Wort zu erzeugen. Beim ersten Aufruf übergeben wir als Input wiederum ein Startzeichen (<*start*>), außerdem den Zustand des Encoders und die Annotationen, die der Encoder produziert hat. In den jeweils folgenden Aufrufen übergeben wir dagegen das zuvor generierte Wort, den Zustand der Decoder-GRU aus dem vorherigen Zeitschritt und wiederum die immer gleichen Annotationen aus dem Encoder.

Mit diesem Hintergrundwissen ausgerüstet, können wir das Inferenzmodells zusammenstellen:

```
96.  def get_inference_model(model, len_seq_encoder: int, hidden_size=64):
97.
97.      batch_size = 1
99.
100.     # Encoder
101.     encoder_in = model.get_layer('encoder_inputs').input
102.     encoder_output, encoder_ffw_h, encoder_bw_h = \
103.                              model.get_layer('encoder_bi_gru').output
104.     encoder_states = Concatenate( axis=-1, name='encoder_concat')(
105.                                [encoder_ffw_h, encoder_bw_h])
106.     encoder = Model( inputs=encoder_in,
107.                  outputs=[encoder_output, encoder_states])
108.
109.     # Decoder
110.     decoder_inputs = Input( batch_shape=(batch_size, 1),
111.                             name='decoder_word_inputs')
112.     encoder_seq_states = Input( batch_shape=(batch_size,
113.                                 len_seq_encoder,
114.                                 hidden_size*2),
115.                                 name='encoder_seq_states')
116.     decoder_init_state = Input( batch_shape=(batch_size,
116.                                 hidden_size*2),
117.                                 name='decoder_init')
119.     decoder_emb = model.get_layer('decoder_emb')
120.     decoder_gru = model.get_layer('decoder_gru')
121.     decoder_att = model.get_layer('attention_layer')
122.     time_distributed = model.get_layer('time_distributed_layer')
123.
124.     decoder_emb = decoder_emb(decoder_inputs)
125.     decoder_out, decoder_state = decoder_gru(decoder_emb,
126.                                  initial_state=decoder_init_state)
127.     attn_out, attn_states = decoder_att( [ encoder_seq_states,
128.                                          decoder_out])
129.     decoder_concat = Concatenate( axis=-1, name='concat')(
130.                                  [decoder_out, attn_out])
131.     decoder_pred = time_distributed(decoder_concat)
132.
133.     decoder = Model( inputs=[encoder_seq_states, decoder_init_state,
```

```
134.                     decoder_inputs],
135.                     outputs=[decoder_pred, attn_states, decoder_state] )
136.    return encoder, decoder
137.
138. encoder, decoder = get_inference_model( model, len_seq_encoder=19)
```

Code-Hinweise:

Die Funktion *get_inference_model* benötigt im Wesentlichen das angelernte Trainingsmodell als Eingabe (*model*), Informationen über die Länge der zu verarbeitenden deutschen Eingaben (*len_seq_encoder*) und Angaben über die Anzahl der Neuronen der Encoder GRU (*hidden_size*).

Im Körper der Funktion geht es dann an die Arbeit: die Zusammensetzung des *Encoders* und *Decoders*. Dabei ist der Encoder wie schon im einfachen Sequence-to-Sequence-Modell wiederum sehr viel einfacher aufgebaut als der Decoder (*Zeilen 100–107*). Wir besorgen uns einfach den Eingang und den Ausgang des Encoders des Trainingsmodells (*model*) und setzen den Graphen dann als eigenständiges Netz wieder zusammen. Abweichend vom Trainingsmodell verbinden wir allerdings die Ausgaben der beiden finalen Zustände aus der bidirektionalen Encoder-GRU (die wie oben beschrieben getrennt für die Vorwärts- und Rückwärtsanalyse erfolgt) bereits an dieser Stelle zu einer gemeinsamen Matrix. Dadurch ersparen wir uns die Arbeit der Anlage und Verwaltung zweier getrennter Eingänge im Decoder. Das Zusammenspielen der beiden Matrizen wickeln wir wie gewohnt mithilfe einer Konkatenierungsschicht ab (*Zeile 104–105*).

Der restliche Teil des Codes ist für die Zusammenstellung des *Decoders* reserviert (*Zeilen 109–135*). Da wir jetzt allerdings die für die Decoder-GRU und den Attention Layer notwendigen Zustände nicht mehr direkt aus dem Encoder beziehen können (weil der als eigenes abgeschlossenes Netz konzipiert ist), müssen wir zunächst drei Eingänge anlegen: einen Eingang für den Input der zeitversetzten englischen Wörter (*Zeilen 110–111*), einen Eingang für die Annotationen pro Wort aus der Encoder-GRU (*Zeilen 112–115*) und einen Eingang für den Endzustand der Encoder-GRU (*Zeilen 116–118*). Wir definieren in diesem Fall allerdings nicht nur die Struktur der einzelnen Datensätze, sondern legen auch fest, dass wir dem Decoder jeweils nur genau einen Datensatz auf einmal zur Verarbeitung übergeben (Parameter *batch_shape*).

Nachdem wir alle relevanten Inputs definiert haben, beschaffen wir uns aus dem trainierten Modell die Schichten, die für den Aufbau des Decoders notwendig sind (*Zeilen 119–122*). Danach stecken wir alles sachgemäß zusammen (*Zeilen 124–131*). Die Embedding-Schicht verbinden wir mit den Decoder-Inputs (*Zeile 124*), die GRU mit der Embedding-Schicht einerseits und dem initialen Zustand aus dem Encoder andererseits (*Zeilen 125–126*). Der Attention Layer erhält den sequenziellen Output aus der Decoder-GRU-Schicht und den sequenziellen Output aus dem Encoder (*Zeilen 127–128*). Wenn das erledigt ist, verbinden wir die Ausgabe aus der Decoder-GRU mit der Ausgabe aus dem Attention Layer und übergeben das Resultat dem Dense Layer zur Produktion der Schätzwerte (*Zeilen 129–131*).

Zum Schluss erzeugen wir aus dem Graphen ein abgeschlossenes Decoder-Modell (*Zeilen 133–135*): Es besteht aus drei Eingaben (vgl. oben) und drei Ausgängen: dem geschätzten Wort (*decoder_pred*), den Gewichten des Attention Layers (*attn_states*, die wir nicht unbedingt brauchen, aber später zur Veranschaulichung des Attention-Mechanismus einsetzen) und dem Zustand der Decoder-GRU nach Verarbeitung der aktuellen Sequenz (*decoder_state*, die wir bei der Produktion mehrerer Wörter jeweils wieder als Eingabe füttern müssen). ■

10.3.2.5 Das Modell für Übersetzungen einsetzen

Genau wie beim einfachen Sequence-to-Sequence-Modell folgt die Produktion von Schätzwerten einem bestimmten Prozedere, für das wir wieder eine gesonderte Funktion anlegen.

Im ersten Schritt füttern wir den Encoder mit einem Quellsatz und erhalten als Rückgabe den Encoder-Zustand und den Encoder-Sequence-Zustand. Dann beauftragen wir den Decoder in einer Schleife mit der Erzeugung des ersten Wortes bzw. aller nachfolgenden Wörter des Zielsatzes. Bei jedem Schätzvorgang werden dabei drei Inputs verlangt: *erstens*, die einzelnen Zustände, die bei der Verarbeitung der Eingabesequenz in der Encoder-RNN entstehen; *zweitens*, der finale Encoder-Zustand bzw. der durch die Decoder-RNN jeweils aktualisierte finale Encoder-Zustand; und *drittens* das Startzeichen bzw. die danach vom Decoder geschätzten Buchstaben:

```
139.  def make_predictions( german: str, encoder, decoder,
140.                        tok_encoder, tok_decoder,
141.                        seq_len_encoder=19, seq_len_decoder=26):
142.     print(german)
143.     x_pred = tok_encoder.texts_to_sequences([german])
144.     x_pred = pad_sequences( x_pred, maxlen=seq_len_encoder,
145.                             padding='post',
146.                             truncating='post')
147.     en_seq_state, en_state = encoder.predict(x_pred)
148.     dec_state = en_state
149.
150.     target_seq = np.zeros((1, 1))
151.     target_seq[0, 0] = tok_en.word_index['<start>']
152.     print('translated:')
153.
154.     for i in range(seq_len_decoder):
155.        dec_out, attention, dec_state = \
156.                 decoder.predict([en_seq_state, dec_state, target_seq])
157.        idx_word = np.argmax(dec_out[0][0])
158.        if idx_word == 0:
159.           break
160.        print(tok_decoder.index_word[idx_word], end=' ')
161.        target_seq = np.zeros((1, 1))
162.        target_seq[0, 0] = idx_word
163.
164.  german = 'auf der anderen seite der straße steht ein baum'
165.  attn = make_predictions( german, encoder, decoder,
166.                           tok_de, tok_en)
```

Ausgabe:

```
auf der anderen seite der straße steht ein baum
translated:
there is a tree across the street
```

Code-Hinweise:

Um einen Satz zu übersetzen, brauchen wir eine ganze Menge Informationen bzw. Objekte. Zum einen natürlich der zu übersetzende Satz als String (Parameter *german*), davon abgesehen das Encoder- und Decoder-Modell (*encoder, decoder*), die beiden angelernten Tokenizer, die die Dictionarys für die Encodierung und Decodierung der deutschen und englischen Wörter enthalten (*tok_encoder, tok_decoder*). Außerdem müssen wir wissen, wie lange die Eingabesequenzen des Encoders und die des Decoders sind (*seq_len_encoder, seq_len_decoder*).

Im Körper der Funktion beginnen wir dann zunächst damit, den zu übersetzenden Satz auf der Konsole auszugeben (*Zeile 142*). Dann bereiten wir den Satz für die Analyse im Encoder vor, indem wir ihn mit dem deutschen Tokenizer in eine Sequenz von Integern überführen (*Zeile 143*) und danach die Sequenz in die vom Encoder erwartete Länge bringen (*Zeile 144–146*).

Wenn das erledigt ist, können wir die Eingabe der predict-Methode des Encoders übergeben (*Zeile 147*). Als Rückgabe erhalten wir die Zustände für die einzelnen Wörter der Sequenz (*en_seq_state*) und den Zustand nach Analyse des gesamten deutschen Satzes (*en_state*).

In *Zeile 148* legen wir die Variable *dec_state* (Decoder-State) an und initialisieren sie mit dem Wert von *en_state* (der Encoder-Zustand wird nur beim ersten Zeitschritt als Decoder-GRU-Zustand eingesetzt, später wird er mit dem jeweiligen Zustand der Decoder-GRU aus dem jeweils letzten Schätzvorgang ersetzt).

In den *Zeilen 150–151* bereiten wir die dritte Eingabe des Decoders vor. Dabei handelt es sich um die bereits geschätzten Wörter bzw. um die Eingabe des Start-Zeichens. Wir liefern dem Decoder jeweils nur ein einzelnes Wort zur Verarbeitung, daher die Struktur *1, 1* (ein Batch, ein Wort). Das enthaltene Wort setzen wir auf das Startzeichen (<*start*>) und verwenden, um herauszufinden, um welche Integer es sich dabei handelt, das Dictionary aus dem Decoder-Tokenizer.

Ab *Zeile 154* beginnt dann die Arbeit der Übersetzung mithilfe des Decoders in einer Schleife. Die Schleife darf höchsten so lange laufen, bis die maximale Anzahl der Decoder-Wörter erreicht ist. Innerhalb der Schleife beginnen wir gleich mit der Schätzung des ersten Wortes und rufen dazu die predict-Methode des Decoders unter Übergabe der drei relevanten Inputs auf (Zustände des Encoders und Startzeichen, *Zeilen 155–156*). Zurück erhalten wir drei Ausgaben, von denen wir im Moment allerdings nur zwei brauchen: die Ausgabe des Decoders (*dec_out*), die das geschätzte Wort enthält, und den aktuellen Zustand der Decoder-GRU. Mit diesem Zustand überschreiben wir die Variable *dec_state* (die wir weiter oben angelegt haben, deren Inhalt jetzt aber nicht mehr aktuell ist).

Im nächsten Schritt (*Zeile 157*) beschaffen wir uns den Integerwert, der bei der Schätzung die höchste Wahrscheinlichkeit erreicht (Softmax-Verfahren). Dabei handelt es sich um das erste noch codierte Zielwort. Falls es sich dabei um den Indexwert 0 handelt, wissen wir, dass wir den Schätzvorgang beenden können. *0* bezeichnet schließlich das Endzeichen einer Sequenz (*Zeilen 158–159*)[7]. Wenn es sich um einen anderen Zahlenwert handelt, bemühen wir das Dictionary des Decoder-Tokenizers und geben das decodierte Wort auf der Konsole aus (*Zeile 160*).

Jetzt müssen wir nur noch alles für den nächsten Schleifendurchlauf vorbereiten. Den Zustand des Decoders haben wir bereits aktualisiert. Der sequenzielle Zustand des Encoders bleibt immer gleich. Wir müssen also nur noch den Input für das geschätzte Wort vorbereiten. Dazu erzeugen wir einfach ein neues NumPy-Array mit der geforderten Struktur und hinterlegen dort die Integer des geschätzten Wortes. Danach geht es in den nächsten Schleifendurchlauf, und das nächste Wort wird generiert. ■

Die Ausgabe entspricht unseren Erwartungen, aus *„Auf der anderen Seite der Straße steht ein Baum“* wird *„There is a tree across the street“*. Das heißt natürlich nicht, dass das für alle Übersetzungen gelten würde – viele gehen schief, was wie fast immer daran liegt, dass wir das Modell über eine zu kleine Datenmenge angelernt haben, an der es die grammatischen Regeln und die Bedeutung der Wörter nur unvollständig erlernen konnte.

Werfen wir zum Abschluss noch einen Blick auf die Arbeit des *Attention Layers*. Wie oben beschrieben, besteht seine Aufgabe darin, die für die Produktion eines Zielwortes relevanten Wörter im Quellsatz bzw. die um diese Wörter fokussierten Zustände aus der Encoder-GRU (die Annotationen) zur Verfügung zu stellen.

Wenn Sie sich die Funktion *make_predictions* genauer ansehen, werden Sie feststellen, dass der Attention Layer neben dem eigentlichen Output, den wir für die Erstellung der Schätzungen in der nachfolgenden Ausgabeschicht benötigen, eine weitere Ausgabe zurückgibt: Das sind die Attention-Gewichte (Variable *attention,* Codezeile *155*). Sie bestimmen darüber, welcher Einfluss den jeweiligen Annotationen auf die Produktion einer Ausgabe zukommt. Im jetzigen Code der Funktion werden die Gewichte zwar in einer Variablen abgelegt, bleiben aber ansonsten ungenutzt. Wenn wir die Ausgaben abfangen würden, erhielten wir für jedes Wort des Zielsatzes einen Vektor, der angibt, mit welchem Gewicht die einzelnen Zustände der Encoder-RNN bei der Schätzung der Zielwörter einfließen. In Bild 10.5 sind diese Gewichte bildlich dargestellt. Die Helligkeit der Pixel zeigt, wie hoch der jeweilige Gewichtungsfaktor ausfällt: je heller, umso stärker der Einfluss eines Quellwortes auf die Produktion eines Zielwortes.

Deutlich zu sehen sind insbesondere die hellen Stellen, die *„Straße“* mit *„street“* und *„Baum“* mit *„tree“* verbinden. Aber auch der unbestimmte Artikel *„ein“* ist mit seinem englischen Pendant *„a“* assoziiert, genauso wie das Verb *„steht“* mit *„is“*.

Die Vorteile der Attention Layer für die Reproduktion des Zielsatzes aus dem Gedächtnis werden an der Illustration noch einmal deutlich: Die Fokussierung auf einzelne Annotatio-

[7] Zur Erinnerung: Die 0 ist im Tokenizer keinem Wort zugeordnet, sondern wird bei der pad_sequences-Funktion verwendet, um zu kurze Sequenzen vom Ende her aufzufüllen.

nen, die im Quellsatz an beliebigen Positionen auftreten dürfen, können im besten Fall eins zu eins übersetzt werden. So wird die Übertragung zu einer koordinierten Übersetzung einzelner Wörter.

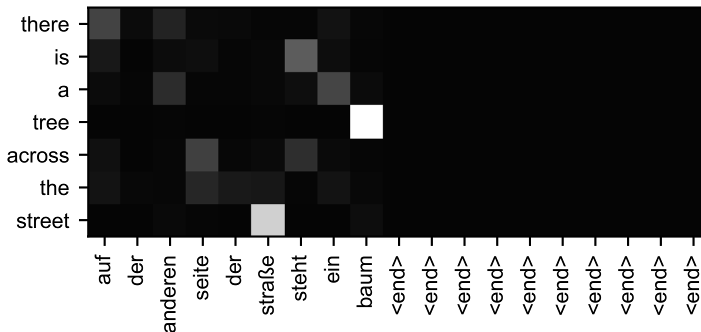
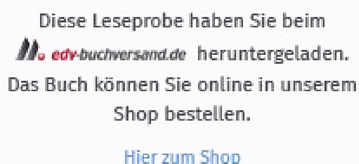


Bild 10.5 Bildliche Darstellung der Gewichte des Attention Layers, die bei der Wort-für-Wort-Übersetzung von *„Auf der anderen Seite der Straße steht ein Baum“* in *„There is a tree across the street“* verwendet werden. Helle Punkte markieren hohe Gewichte, dunkle Punkte niedrige Gewichtungen. Auffällig sind insbesondere die beiden hellen Punkte, die das Wort *„Straße“* mit *„street“* und das Wort *„Baum“* mit *„tree“* in Verbindung setzen.


Diese Leseprobe haben Sie beim
edv-buchversand.de heruntergeladen.
Das Buch können Sie online in unserem
Shop bestellen.
Hier zum Shop