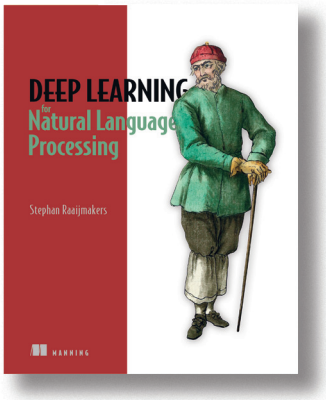
Author Picks

FREE

Exploring Natural Language Processing

Chapters selected by Hobson Lane

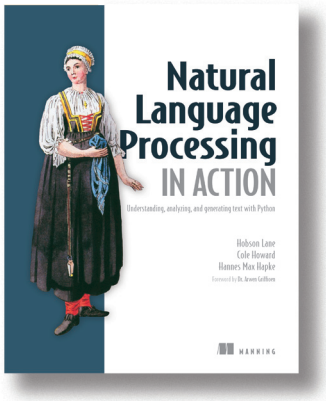
manning



Deep Learning for Natural Language Processing

by Stephan Raaijmakers

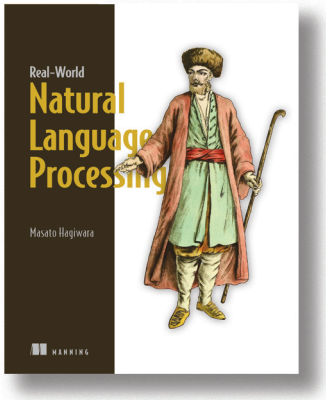
ISBN 9781617295447
325 pages
$39.99



Natural Language Processing in Action

by Hobson Lane, Cole Howard, Hannes Hapke

ISBN 9781617294631
544 pages
$39.99



Real World Natural Language Processing

by Masato Hagiwara

ISBN 9781617296420
500 pages
$47.99

Exploring Natural Language Processing

Chapters chosen by Hobson Lane

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Erin Twohey, corp-sales@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617297342

contents

introduction

If you've been waiting for Natural Language Processing to capture your imagination with exciting use cases, your wait may be over. A tectonic shift in society and artificial intelligence technology has begun. Bots powered by NLP have begun to direct the evolution of societies, countries, and even entire continents. The AI explosion that futurists have been anticipating is upon us, and this explosion is powered by Natural Language Processing technology in the hands of people like you.

Ever since computers were invented, engineers have dreamed of machines that could understand and communicate with us by using natural language text. Computer scientists, linguists, and machine learning engineers made steady progress toward that goal. Recently, that incremental progress hit an inflection point and produced a rapid succession of leaps in the accuracy and breadth of Natural Language Processing power. Search engines have made us all experts in virtually everything. Virtual assistants (chatbots) for customer service, scheduling, recruiting, navigation, and even therapy make it possible to automate much of our lives and influence one another without lifting a finger. We have more and more time for high-level thinking and exploration. And we've come full circle with bots composing tweets, Facebook posts, advertisements, and even entire news reports and movie scripts that are being indexed by search engines to satisfy our voracious appetite for information.

We hope that these chapters will surprise and inspire you with your own ideas for applying NLP to your personal and professional life. More importantly, we hope that the prosocial sentiment of these chapters will become part of your thinking. We hope that you'll be inspired by our emphasis on virtual assistants that truly assist rather than manipulate. The changes the AI explosion will bring will be largely positive for us all if you use your newfound society-influencing NLP powers for good rather than evil!

Hobson Lane
Co-author of *Natural Language Processing in Action*

Chapter 2 from *Deep Learning for Natural Language Processing* by Stephan Raaijmakers

Deep Learning has been the acceleration force behind the explosion in Natural Language Processing capability over the past six years. Recently, deep learning approaches have replaced conventional NLP techniques for the most challenging applications, such as translation, virtual assistants, and question--answering. In this chapter, Stephan Raaijmakers lays the foundation that you'll need to harness that power. You'll learn about spatial filtering (Convolutional Neural Networks) and temporal filtering (Recurrent Neural Networks)—the cornerstones of advanced Natural Language Processing applications, from word embeddings to sentence encodings and from spam detection to translation and natural language generation.

Deep learning and language: the basics

This chapter covers

- The fundamental architectures of deep learning
- Deep learning models for natural language processing

After reading this chapter, you will have a clear idea of how deep learning works, why it is different from other machine learning approaches, and what it brings to the field of natural language processing. As a prerequisite, you should be familiar with the Keras (Python) library for deep learning. This chapter will introduce you to some high-level Keras concepts and their details through examples. You can find a thorough introduction to deep learning in the Manning book *Deep Learning with Python by François Chollet.*

Figure 2.1 shows the organization of this chapter.

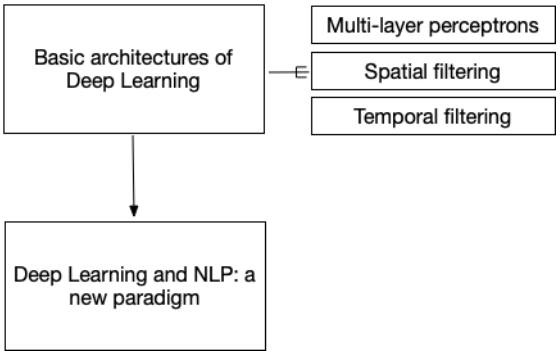


Figure 2.1 Basic architecture of deep learning

2.1 Basic architectures of deep learning

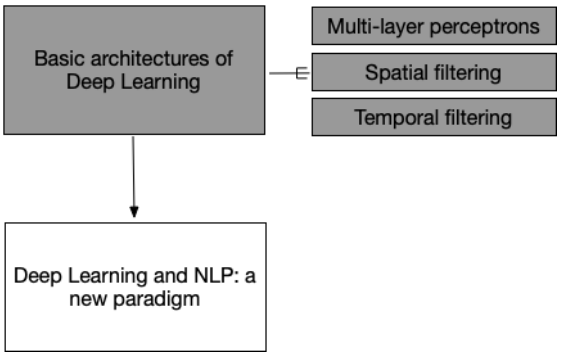


Figure 2.2

2.1.1 Deep multilayer perceptrons

The prototypical deep learning network is a multilayer perceptron (MLP). See figure 2.3 for a simple MLP, which has a single hidden layer.

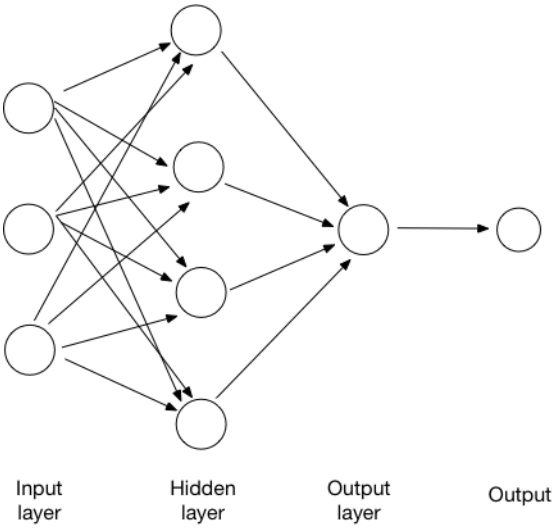


Figure 2.3 A generic MLP

MLPs consist of layers containing *artificial neurons*, which are mathematical functions that receive input from weighted connections to other neurons. They produce output values through a variety of mathematical operations. Deep neural networks have lots of neurons and lots of weights to manipulate. A typical deep network usually has many hidden layers. But how many? Is there a magic threshold that demarcates the boundary between shallow and deep learning? As you may guess, there is no such magic number, but informally, a network with more than two layers between its input and output layer may be deemed deep.

Let's go through the architecture of a deep MLP step by step, omitting the processing of training data.

Listing 2.1 A ten-layer-deep MLP: creating the model

```
from keras.models import Sequential
from keras.utils import np_utils
from keras.preprocessing.text import Tokenizer
from keras.layers.core import Dense, Activation, Dropout
# ... process data
model = Sequential()
model.add(Dense(128, input_dim=input_dim))
model.add(Activation('relu'))
```

We start our network by importing the relevant Keras facilities: a Sequential model, which defines a container for a stacked set of layers, and facilities for defining layers. We initialize such a container, called "model". To this container, we add a Dense layer with an input dimension of input_dim (a self-defined variable) and an output dimension of 128. The layer consumes input with dimension (*,input_dim), which is a *tensor* (a container for numerical data) of size (batch_size ,input_dim). The batch_size determines the grouping of data points in batches that are handled collectively. Leaving the batch_size implicit is permitted; then it defaults to 1.

This Dense layer feeds into a next layer of dimension 128 by producing a tensor (*,128). Figure 2.4 illustrates how the model is built up.

Every input unit (that is: every component of the input_dim size input vector) feeds into every one of those 128 next-layer components, which is why the layer is called Dense. Notice that you need to specify the size of the data feeding into a Dense layer only once: upon initialization. We add a ReLU activation function, closing our layer and preparing its output to be sent to subsequent layers.

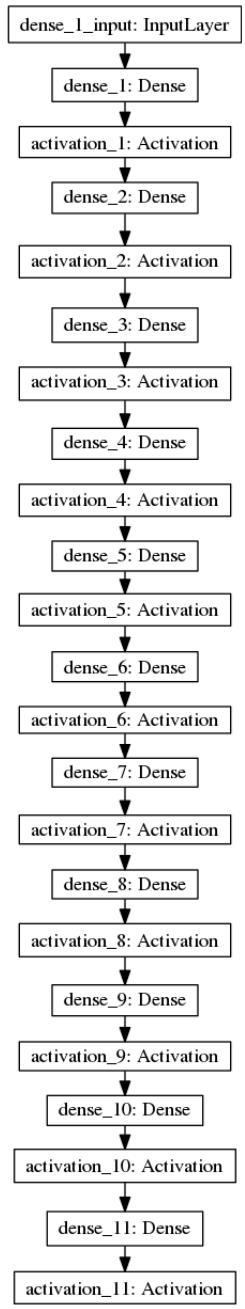


Figure 2.4 A plot of our deep model

Let's add a couple of those layers to our network. Every one of those layers deploys a ReLU activation function.

Listing 2.2 A five-layer-deep MLP: adding layers

```
# ...
model.add(Dense(128))
model.add(Activation('relu'))

model.add(Dense(128))
model.add(Activation('relu'))

model.add(Dense(128))
model.add(Activation('relu'))

model.add(Dense(128))
model.add(Activation('relu'))

model.add(Dense(nb_classes))
model.add(Activation('softmax'))
```

After devising our model, we *compile* it (prepare the model for training) by specifying a *loss function* (a real-valued function that computes the mismatch between predictions the model makes, and the labels it should assign according to the training data). We also specify a numerical optimizer algorithm, which carries out the gradient descent process, and an evaluation metric, which performs intermediate evaluation of the model during training, and which specifies the loss of the loss function (like accuracy, or mean squared error). The evaluation metric evaluates the model performance during training of held-out test data, taken from the training dataset. In the fit step, we can determine the size of this held-out portion of the training data (validation_split=0.1, which means use 10 percent of the training data for testing), the number of training epochs (epochs=10), and the size of the batches of training data that are taken into account at every training step (batch_size=32).

Listing 2.3 A five-layer-deep MLP: compilation and fitting the model to data

```
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.1,
     shuffle=True,verbose=2)
```

This network is deep according to our informal definition; it has five layers. Its structure is quite simplistic: it connects five similar-size Dense layers, in a standard feed-for-ward manner. Out deep learning toolkit has more to offer, though. In section 2.1.2, we take a look at two basic operators at our disposal.

2.1.2 Two basic operators: spatial and temporal

Deep learning networks often display the interplay between two types of information filtering: spatial filtering and temporal filtering. *Spatial filters* address properties of the structure of input data, weeding out irrelevant patches and letting the valuable ones pass. *Temporal filters* do something similar but work on sequences of memory states that store information from the past; they are typically used for processing sequences.

SPATIAL FILTERING: CONVOLUTIONAL NEURAL NETWORKS

The convolutional neural network (CNN) is the driving force behind many of the big successes in image processing. It can be applied to text analysis as well. A CNN applies a set of weighted filters (called *convolutions*) to input data and learns the weights of these filters on the basis of training data. The filters scan the data in multiple locations and gradually become specialized feature detectors, extracting (*emphasizing*) important pieces of information. Stacking layers of these detectors re-creates the marvel of deep learning, in which we focus on increasingly abstract representations of our data. Convolutional filters are easiest to understand when they're applied to images, so let's take a look at those filters first.

Images can be represented as three-dimensional objects (height, width, and color depth), as shown in figure 2.5.

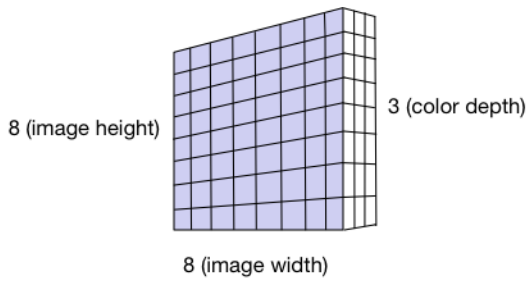


Figure 2.5 A 3D representation of an image

A CNN applies a prespecified number of filters to the grid of pixels that constitutes an image. These filters are nothing but weight matrices. Together, they emphasize certain parts of the input image with respect to the classification task at hand. Picture such a filter as a piece of paper smaller than the image shifted over the image from left to right and top to bottom, visiting all possible arrangements one at a time, and continuously shifting a certain fixed number of pixels. This step size is called a *stride*. In the case of processing images, every filter performs a weighted *aggregation* (a sum of products) on the $N \times N \times 3$ grid of pixel (RGB) values it is visiting.

For every time the filter slides over the image, a separate value is computed based on all the pixels in the image the filter visits. This value is entered into a new result matrix, which has a dimension of $H \times V$, with N being the number of horizontal moves the filter makes and V being the number of vertical moves.

Figure 2.6 shows a 2x2 filter being applied to the top-left corner of our 8x8x3 image.

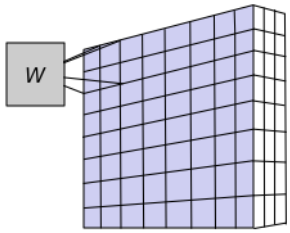


Figure 2.6 Filtering an image with a convolutional filter

To understand what's going on here, assume that we're dealing with grayscale images with one channel rather than the three RGB channels used in color images. Figure 2.7 represents such an image.

1	0	1	1
1	1	1	0
1	0	0	0
1	0	1	1

Figure 2.7 An image represented as a binary grid of numbers

Assume that we have a filter like the one in figure 2.8.

1	0
0	1

Figure 2.8 A convolutional filter

In figure 2.9, the filter slides over the top-left corner of the grayscale image.

1	0	...	...
1	1	...	...
...	...	...	...
...	...	...	...

Figure 2.9 Applying a filter to an image

The filter produces the following sum-product number:

```
1 x 1 + 0 x 0 + 0 x 1 + 1 x 1 = 2
```

Because we can shift maximally three times in the horizontal direction and three times in the vertical direction, the filter creates a 3x3 output matrix, with the sum product of the filter output in every cell (figure 2.10).

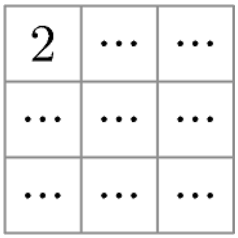


Figure 2.10 Filter output

The crucial trick that CNNs perform is learning their filters from training data. So rather than prespecifying the filter matrices by hand, we let CNNs start with random initializations of these filters. We have a few specifications to make, of course, including the number of filters, the stride, and the filter size. Then CNNs figure out better weights for the filters during training and learn to emphasize or deemphasize certain parts of the data. Although these weights will never be optimal in a theoretical sense, they will be optimized with the training data and the parameters of the training procedure, and they certainly will be better than the initial random weights.

In separate max-pooling layers, so-called max filters perform a maximum selection over the grids they visit. If a filter visits a 4x4 image, addressing 2x2 patches with the values

- [1,1;5,6]
- [[2,4;7,8]
- [3,2;1,2]
- [1,0;3,4]

max pooling would take out the maximum value (6,8,3,4) from this array and send it to the next layer in the network (figure 2.11).

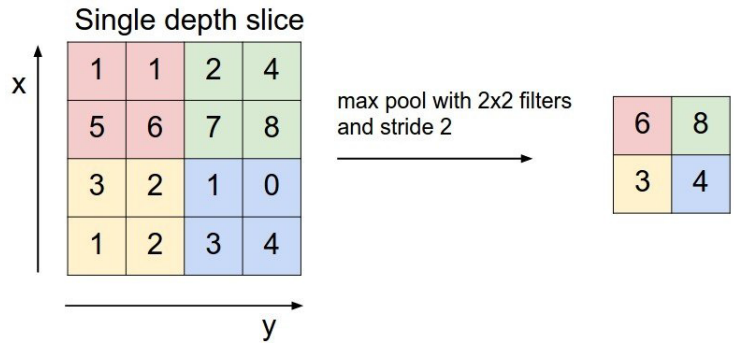


Figure 2.11 Max pooling(Credits: Stanford CS231n)

Max pooling can be interpreted as a form of downsampling. Notice that patches of images in our example are converted to separate numbers by picking out maximum values. This process performs dimensionality reduction of feature representations, converting representations of a certain dimension into lower-dimensional representations. In our example, a 4x4 matrix was turned into a 2x2 matrix.

A full-fledged CNN, then, may look like figure 2.12.

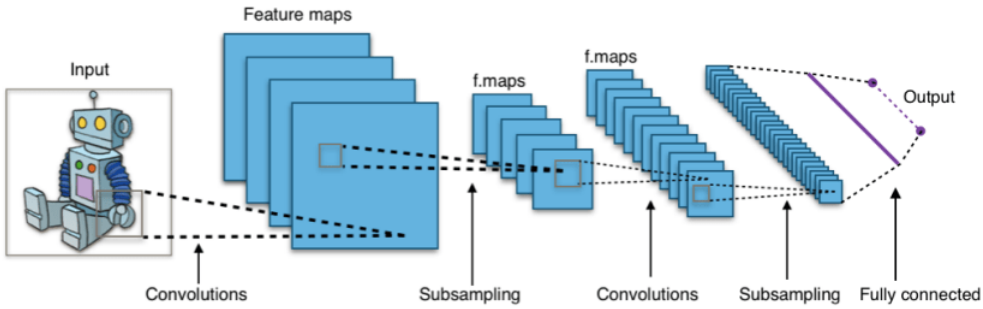


Figure 2.12 **Full-fledged CNN (Image: http://www.jessicayung.com/explaining-tensorflow-code-for-a-convolutional-neural-network/; rework to proprietary)**

CNNs FOR TEXT

CNNs can be applied to text as well. .Textual objects like strings typically are 1D objects—streams of characters extending into one dimension, which for strings is a horizontal dimension (figure 2.13). It makes no real sense also to discern a vertical dimension of strings, but it might make sense for documents.

Figure 2.13 **A 1D representation of a text**

What is the meaning of the spatial sampling that CNNs perform on texts? Recall that a regular filter is a weight matrix that emphasizes or deemphasizes its input. Applied to text, a CNN could detect interesting words or other features that are relevant to a certain NLP task. Metaphorically, this process is not dissimilar to the process in which a human scans a document and marks certain fragments as relevant.

Let's construct a sentiment-analysis network that uses convolutional layers instead of Dense layers. Listing 2.4 introduces the code.

Listing 2.4 A CNN for sentiment analysis

```
from keras.models import Sequential
from keras.layers import Dense
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Convolution1D, Flatten, Dropout
```

We import the necessary modules.

```
from keras.layers.embeddings import Embedding
from sklearn.model_selection import train_test_split
from keras.preprocessing import sequence
import pandas as pd
import sys
from keras.utils.vis_utils import plot_model

data = pd.read_csv(sys.argv[1],sep='\t')
max_words = 1000
tokenizer = Tokenizer(num_words=max_words, split=' ')
tokenizer.fit_on_texts(data['text'].values)

X = tokenizer.texts_to_sequences(data['text'].values)
X = pad_sequences(X)
Y = pd.get_dummies(data['label']).values

X_train, X_test, y_train, y_test = train_test_split(X,Y, test_size = 0.2,
     random_state = 36)

embedding_vector_length = 100

model = Sequential()
model.add(Embedding(max_words, embedding_vector_length,
    input_length=X.shape[1]))
model.add(Convolution1D(64, 3, padding="same"))
model.add(Convolution1D(32, 3, padding="same"))
model.add(Convolution1D(16, 3, padding="same"))
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(2,activation='sigmoid'))

# Model inspection
model.summary()
plot_model(model, to_file='model.png')

model.compile(loss='binary_crossentropy', optimizer='adam',
     metrics=['accuracy'])

model.fit(X_train, y_train, epochs=3, batch_size=64)

# Evaluation on the test set
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Here, we read out tab-delimited input data and process it to become a pair of numerical vectors (X) and associated labels (Y).

Our model is declared and defined.

We split X and Y into training and test partitions.

The first (input) layer is an embedding.

Three convolutional layers are stacked.

A flattening operation and a Dropout() action randomly deselect neurons to prevent overfitting. Too tight a fit to the training data prevents the model from coping successfully with new, unseen cases.

The model is compiled and prepared for fitting the data. It consumes its training data in batches of size 64 during training.

The model we built, depicted in figure 2.14, was generated by the built-in plot_model function of Keras.

Let's go through the code step by step. First, we read and split our data (our tab-separated file) with sentiment-labeled texts.

Listing 2.5 Reading and processing the training data

```
data = pd.read_csv(sys.argv[1],sep='\t') # tsv
     file
max_words = 1000
tokenizer = Tokenizer(num_words=max_words,
     split=' ')

tokenizer.fit_on_texts(data['text'].values)

X = tokenizer.texts_to_sequences(data['text'].
     values)
X = pad_sequences(X)
Y = pd.get_dummies(data['label']).values

X_train, X_test, y_train, y_test =
     train_test_split(X,Y, test_size = 0.2,
     random_state = 36)
```

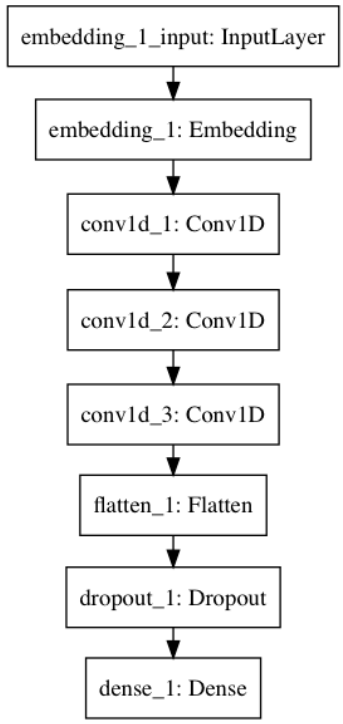


Figure 2.14 Our CNN model

X and Y together constitute our training data, with X being the set of documents and Y being the labels. Each document is represented by an array of integers that refer to its words, padded to a uniform length by the Keras function `pad_sequences`. This uniform length is the maximum length of the documents in X, which in our case happens to be 65. We split our data into 80 percent for training and 20 percent for testing.

Subsequently, we produce an embedding of the words in our data, based on 100-dimensional vector embeddings for every word. *Embeddings* are procedures for turning text into vector representations.

Listing 2.6 Creating an Embedding layer

```
embedding_vector_length = 100

model = Sequential()

model.add(Embedding(max_words, embedding_vector_length,
     input_length=X.shape[1]))
```

We go on to define our model, which contains three convolutional layers. Every layer specifies the dimensionality of the output space (64,32,16) and the size of every filter (3), also known as the *kernel size*. The values (64,32,16) were chosen arbitrarily; good practice would be to estimate these values (*hyperparameters*) on some held-out validation data. The stride (step size) is set to 1 per default, but this setting can be overridden.

Listing 2.7 Adding layers to the model

```
model.add(Convolution1D(64, 3, padding="same"))
model.add(Convolution1D(32, 3, padding="same"))
model.add(Convolution1D(16, 3, padding="same"))
```

```
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(2,activation='sigmoid'))
```

The Dropout layer randomly resets a specified fraction (0.2, in our case) of its input units to 0 during training time, at every update step, to prevent overfitting. The longest document in our texts contains 65 words.

Keras has a `model.summary()` function that generates the layout of a model. The architecture of our model is

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 65, 100)           100000
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 65, 64)            19264
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 65, 32)            6176
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 65, 16)            1552
_________________________________________________________________
flatten_1 (Flatten)          (None, 1040)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1040)              0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 2082
=================================================================
Total params: 129,074
Trainable params: 129,074
Non-trainable params: 0
_________________________________________________________________
```

The Flatten layer coerces the 65x16 output of the final convolutional layer into a 1040-dimensional array, which is fed into Dropout, and a final two-dimensional Dense layer, which contains the binary representation of the output labels (subjective or objective sentiment).

Training the classifier for three iterations and running it on the 20 percent held-out data gives

```
8000/8000 [==============================] - 4s - loss: 0.4339 - acc: 0.7729
Epoch 2/3
8000/8000 [==============================] - 3s - loss: 0.2558 - acc: 0.8948
Epoch 3/3
8000/8000 [==============================] - 3s - loss: 0.2324 - acc: 0.9052
Accuracy: 87.80%
```

Each line represents the output from an epoch (3). An *epoch* represents a full sweep through the training data. The `*s` indicates the amount of time spent on this epoch (which, of course, is machine- and dataset-dependent). The loss reported is the error obtained on the training data, which in our case, is measured with binary cross-entropy. Cross-entropy-based loss expresses the performance of a classification model producing probabilities (values between 0 and 1). This `loss` function produces larger

values when the predictions deviate from the correct labels, so we want to keep it as low as possible. The `acc` scores express the accuracy of the model obtained on the training data during training.

Let's see how this works up close. Figure 2.15 describes the process.

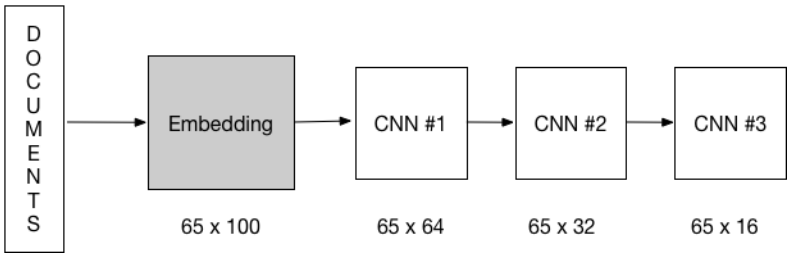


Figure 2.15 A convolutional network for sentiment analysis

The words in our data are squeezed through an embedding of size 100, with *1000 x 100 x 1* (100,000) parameters. The output matrix, padded for the maximum length of our input documents (65), has a size of *65 x 100*. So every input document is represented by a matrix of size *65 x 100*. The rows (65) correspond to the original word order in the input document, which implements some form of temporal structure. All this is fed to the first convolutional layer, which applies 64 convolution operations with kernel size 3 to the 65 100-dimensional entries of the input vector.

Because the filter length (the kernel size) is 3, we start with the first three words (or rows in the matrix). The first filter is applied to the three vector representations of these words. Then we shift one word (*stride=1*) and apply the filter to the vectors of the words 2,3,4, and so on until we reach the final group (words 63,64,65).

In an example like

- The quick brown fox jumps over the lazy dog

we would proceed as follows

- the quick brown
- quick brown fox
- brown fox jumps

and so on.

For every step, the filter creates a single value. The final list of values is padded to become the same length as the input. (In our case, we have *65-3=62* shifts, so we pad the result vector to 65.). This produces one sequence of 65 numbers. Repeating the process 64 times with different filters, we end up with a matrix with *65 x 64* values.

The second convolutional layer receives this 2D matrix as input. Because the layer is a 1D convolutional layer, it applies its 32-fold convolution to the 65 64-dimensional entries. Like the first layer, it works on groups of three word vectors at a time, again producing 2D output, now with the dimensions *65 x 32*. And so on.

TEMPORAL FILTERING: RECURRENT NETWORKS

Recurrent neural networks (RNNs) are deep networks with depth extending across the horizontal direction of a timeline. When it is unfolded, a recurrent neural network becomes a sequence of many dependent layers, and the dependencies among weights across these layers are exactly what makes such a network deep.

RNNs emphasize or deemphasize certain aspects of the information that they process in an end-to-end fashion, working in a temporal dimension rather than a spatial dimension. This selection process boils down to learning weight matrices. Crucially, RNNs can memorize their previous cognitive states and the decisions they made in the past. This feature provides a facility for implementing a certain bias into a neural net, allowing it to carry out classifications that are in line with what it has done in the recent past.

There are two main brands of these networks: simple RNNs and *long short-term memories* (LSTMs). Let's start with simple RNNs.

A *simple RNN* is a neural network with a limited amount of memory. At every time tick, it deploys the hidden memory state of a previous time tick for producing a current output. At any time, an RNN memory state is determined by three factors:

- The memory state at the previous *time tick* (assuming that time passes in a discrete fashion, one tick at a time)
  - A weight matrix weighting the previous memory state
  - A weight matrix weighting the current input to the RNN

Figure 2.16 illustrates a simple RNN.

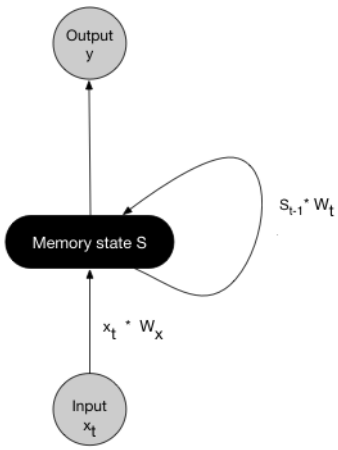


Figure 2.16 A simple RNN

The memory state *S* is updated iteratively, for every time tick *t*, as follows:

$S_t = S_{t-1} * W_t + x_t * W_x$

Importantly, weights (W_t and W_x) are shared by all updates. Notice that this network has a recurrent loop, which can be expanded (*unrolled*) as shown in figure 2.17.

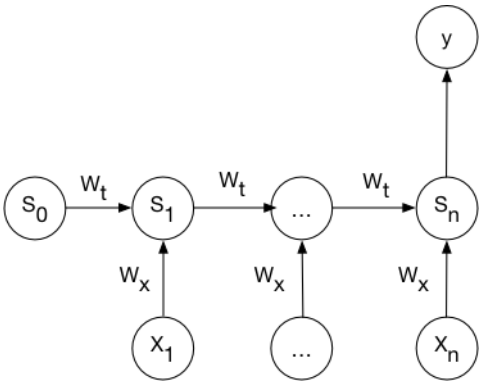


Figure 2.17 A simple unrolled RNN

Notice that there are as many states as inputs. Every new input transforms the RNN into a new cognitive state.

The number of time steps arising from this unrolling constitutes the depth of the network across the temporal dimension. By sharing and updating weights across all inputs (the x_t) to which the network is exposed, we obtain a system that learns from experience and optimizes its weights globally to minimize training errors.

Let's dive into an example. We will build a simple RNN that learns to predict the next character for a string, based on the characters that precede that character. Figure 2.18 illustrates the network.

In this diagram, we start with a one-hot encoding of characters. The one-hot vectors are fed into the hidden units of the RNN, forming a temporal chain, with every hidden unit feeding into the next unit through time. Every time tick processes one character. We end with a reconstruction step, in which one-hot vectors are reconstructed from the hidden unit outputs. Here is our code:

Listing 2.8 Predicting one character at a time with a simple RNN

```
from keras.models import Sequential
from keras.layers import SimpleRNN, TimeDistributed, Dense
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
import numpy as np

#data = ['character']

#data = ['character']

enc = LabelEncoder()
alphabet = np.array(list(set([c for w in data for c in w]))) #
enc.fit(alphabet)
int_enc=enc.fit_transform(alphabet)
onehot_encoder = OneHotEncoder(sparse=False)
int_enc=int_enc.reshape(len(int_enc), 1)
onehot_encoded = onehot_encoder.fit_transform(int_enc)
```

Keras offers a dedicated RNN layer (SimpleRNN) and a facility for passing weight information through time (TimeDistributed).

We use sklearn's LabelEncoder and OneHotEncoder to vectorize our data, using one-hot vectors for every character.

Our data is the string 'character'.

We derive an alphabet from our data.

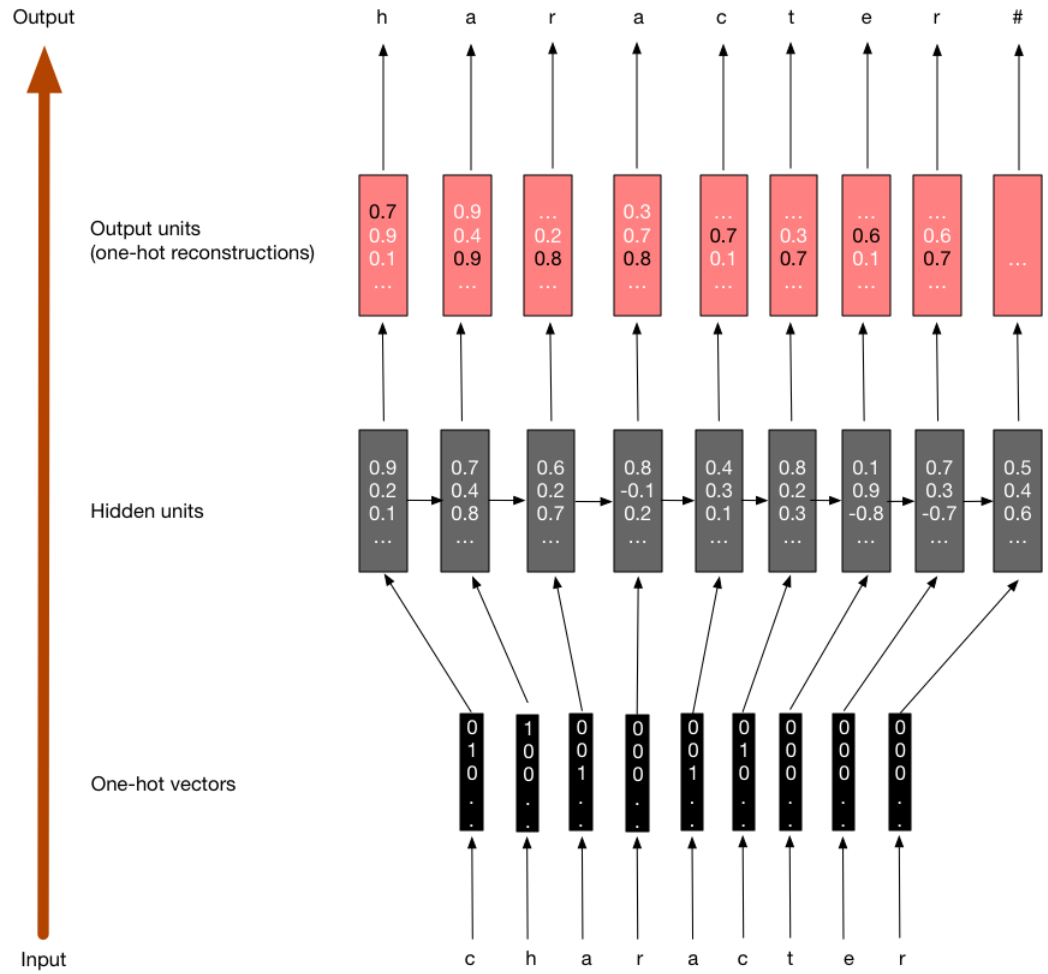


Figure 2.18 **A simple RNN predicting characters. Normally, we would show this diagram top-down, but they are usually presented bottom-up in literature.**

```
X_train=[]
y_train=[]

for w in data:
    for i in range(len(w)-1):
        X_train.extend(onehot_encoder.transform([enc.transform([w[i]])]))
        y_train.extend(onehot_encoder.transform([enc.transform([w[i+1]])]))


X_test=[]
y_test=[]

test_data = ['character']

for w in test_data:
```

We create our training data.

We create our test data.

```
    for i in range(len(w)-1):
        X_test.extend(onehot_encoder.transform([enc.transform([w[i]])]))
        y_test.extend(onehot_encoder.transform([enc.transform([w[i+1]])]))


sample_size=256
sample_len=len(X_train)

X_train =
     np.array([X_train*sample_size]).reshape(sample_size,sample_len,len(alpha
     bet))
y_train =
     np.array([y_train*sample_size]).reshape(sample_size,sample_len,len(alpha
     bet))
test_len=len(X_test)
X_test= np.array([X_test]).reshape(1,test_len,len(alphabet))
y_test= np.array([y_test]).reshape(1,test_len,len(alphabet))

model=Sequential()
model.add(SimpleRNN(input_dim  = len(alphabet), output_dim = 100,
     return_sequences = True))
model.add(TimeDistributed(Dense(output_dim = len(alphabet), activation  =
     "sigmoid")))
model.compile(loss="binary_crossentropy",metrics=["accuracy"], optimizer =
     "adam")
model.fit(X_train, y_train, nb_epoch = 10, batch_size = 32)

preds=model.predict(X_test)[0]
for p in preds:
    m=np.argmax(p)
    print(enc.inverse_transform(m)),

print(model.evaluate(X_test,y_test,batch_size=32))
```

We generate 256 samples of our training data to arrive at a sizable training set for this toy problem.

We apply the model to our test data.

The model is defined and fitted on the training data. It has a SimpleRNN layer feeding into a so-called TimeDistributed *layer wrapper*. This is a Keras facility for applying a layer (in our case, a Dense layer) to every sample (corresponding to a time step) in a received set of samples (in this case, produced by the SimpleRNN layer). The return_sequences=True setting in the SimpleRNN layer ensures that the SimpleRNN layer sends out predictions as characters.

This code produces the following output:

```
Epoch 1/10
256/256 [==============================] - 0s - loss: 0.5248 - acc: 0.8125
Epoch 2/10
256/256 [==============================] - 0s - loss: 0.2658 - acc: 0.9375
Epoch 3/10
256/256 [==============================] - 0s - loss: 0.1633 - acc: 0.9661
Epoch 4/10
256/256 [==============================] - 0s - loss: 0.1169 - acc: 0.9792
Epoch 5/10
256/256 [==============================] - 0s - loss: 0.0913 - acc: 0.9922
Epoch 6/10
256/256 [==============================] - 0s - loss: 0.0752 - acc: 1.0000
Epoch 7/10
```

```
256/256 [==============================] - 0s - loss: 0.0637 - acc: 1.0000
Epoch 8/10
256/256 [==============================] - 0s - loss: 0.0548 - acc: 1.0000
Epoch 9/10
256/256 [==============================] - 0s - loss: 0.0474 - acc: 1.0000
Epoch 10/10
256/256 [==============================] - 0s - loss: 0.0411 - acc: 1.0000

h
a
r
a
c
t
e
r

[0.037899412214756012, 1.0]
```

The network starts its work right after it sees the initial `c` and predicts every next character in turn. From the output, we see that the network has generated the correct characters for its (only) training sample. The two final numerical scores express the loss and accuracy of the classifier for the test data; the 1.0 indicates a 100 percent accuracy score.

Despite this modest success, simple RNNs are rather crude temporal networks. They fail on long sequences (they have small, limited-capacity memories) and blindly reuse hidden states in their entirety without discriminating between information that is rubbish and information that has value.

LSTM networks attempt to remedy the defects of simple RNNs by adding gating operations to the passage of historical network information to the present.

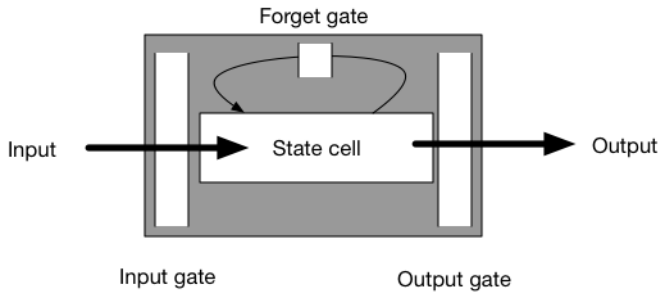


Figure 2.19 An LSTM cell

So we have the following ingredients:

- Input data
- An input gate that weights new input by applying a weight matrix to it
- A State cell, the hidden state of the network, which consumes the weighted input data, and applies a forget gate that weights information from the previous State cell

- A forget gate, selectively letting through old information from the previous State cell and weighing the input with a separate weight matrix.
- An output gate, sending out the information from the current State cell selectively
- Output, computed from what the output gate lets through

The careful gating of information through an LSTM cell takes place on the basis of nine weight matrices, which play a role in an ingenious computation. We'll step through the equations describing it. Although the details of these equations need not concern us now, if you read these equations carefully, you'll get a feel for how historical information percolates through the network.

The nine weight matrices are

- W_i—weights on the input gate, applied to the input
- W_o—weights on the output gate, applied to the input
- W_f—weights on the forget gate, applied to the input
- W_c—weights on the input, for computing the activity of the entire cell
- H_i—input gate weights applied to the previous hidden state of the net h_{t-1} (which is defined in terms of its output at a previous time step, and its previous hidden state)
- H_c—weights applied to the hidden state of the net, for computing the activity of the entire cell
- H_f—forget gate weights applied to the previous hidden state of the net
- H_o—output gate weights applied to the previous hidden state of the net
- V_c—weights applied to the cell activity

How are all these weight matrices used in computing simple output of an LSTM cell? Let's work through the various gates at a given time step *t* with an associated input *xt* and every *bi* a vector of bias values. The network computes a candidate cell activation C', which is turned into an actual cell activation based on the output of the input and forget gates. Then this cell activation is used for the final output computation.

- Input gate—$i_t=\text{sigmoid}(W_i\, x_t + H_i\, h_{t-1} + b_i)$

  Apply a sigmoid activation function to weighted input plus the weighted hidden state and a bias term.
- Forget gate—$f_t=\text{sigmoid}(W_f\, x_t + H_f\, h_{t-1} + b_f)$

  Apply a sigmoid function to input weighted with forget gate weights plus forget gate weighted hidden state and a bias term.
- $C'_t=\tanh(W_c\, x_t + H_c\, h_{t-1} + b_c)$

  This auxiliary state cell variable (the candidate cell activation) is a tanh function of weighted input, weighted hidden states, and a bias term.
- $C_t=i_t \times C'_t + f_t \times C_{t-1}$

  The current cell activation is computed from the input weighted auxiliary state C' and the forget gate weighted previous cell activation.

- o_t=sigmoid$(W_o\, x_t + H_o\, h_{t-1} + V_c\, C_t + b_o)$

  The output gate produces the output of the sigmoid function applied to output gate weighted input plus the output gate weighted hidden state plus the state call weighted cell activation plus a bias term.

- $h_t = o_t \tanh(C_t)$

  The hidden state at time *t* is a tanh function of the current state cell C_t.

Notice the heavy use of addition in these equations. Avoiding multiplication is one way of tackling the vanishing gradient problem. Using tanh ensures that values will be in the interval [-1,1]. Similarly, the sigmoid function recasts values to the interval [0,1].

The current cell activation C_t combines incoming data (passed through the input gate) with the forget gate applied to the previous state of the cell, which is a ledger of old information kept alive. Notice how much weighting and bias application is going on in this tiny memory structure. In Keras, an LSTM layer incorporates exactly one such cell or memory state for every batch. So given a dataset of 1000 sequences, if we have a batch size of 100 with 20 sequences of length 10, we obtain 10 batches, each of which contains one memory state or instantiation of the LSTM cell presented above. Parameter updates are shared across batches during the training phase. After each sequence has been processed during training or testing, the memory states are reset. The memory states can be collected for all sequences and preserved across batches if desired with so-called *stateful* LSTMs.

Let us rework our previous example, migrating from a simple RNN to a full-fledged LSTM network. After that, we will put both the RNN and the LSTM to the test: memorizing parts of a string.

In Keras, an LSTM needs 3D input of the form *(samples, timesteps, features)* and consumes its input data in batches. The *samples* dimension is the amount of data blocks we feed to the LSTM; the *timesteps* dimension refers to the number of observations in each batch; and the *features* dimension refers to the number of features in each observation.

Listing 2.9 Predicting one character at a time with an LSTM

```
from keras.models import Sequential
from keras.layers import LSTM, TimeDistributed, Dense
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
import numpy as np

np.random.seed(1234)

data = ['xyzaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaxyz',
        'pqraaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaapqr']

test_data = ['xyzaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaxyz',
             'pqraaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaapqr']
```

Keras has a separate Layer for LSTMs.

Our data is long strings.

```
enc = LabelEncoder()
alphabet = np.array(list(set([c for w in data for c in w])))
enc.fit(alphabet)
int_enc=enc.fit_transform(alphabet)
onehot_encoder = OneHotEncoder(sparse=False)
int_enc=int_enc.reshape(len(int_enc), 1)
onehot_encoded = onehot_encoder.fit_transform(int_enc)


X_train=[]
y_train=[]

for w in data:
    for i in range(len(w)-1):
        X_train.extend(onehot_encoder.transform([enc.transform([w[i]])]))
        y_train.extend(onehot_encoder.transform([enc.transform([w[i+1]])]))

X_test=[]
y_test=[]

for w in test_data:
    for i in range(len(w)-1):
        X_test.extend(onehot_encoder.transform([enc.transform([w[i]])]))
        print i,w[i],onehot_encoder.transform([enc.transform([w[i]])])
        y_test.extend(onehot_encoder.transform([enc.transform([w[i+1]])]))


sample_size=512
sample_len=len(X_train)

X_train =
     np.array([X_train*sample_size]).reshape(sample_size,sample_len,len(alpha
     bet))
y_train =
     np.array([y_train*sample_size]).reshape(sample_size,sample_len,len(alpha
     bet))

test_len=len(X_test)
X_test= np.array([X_test]).reshape(1,test_len,len(alphabet))
y_test= np.array([y_test]).reshape(1,test_len,len(alphabet))
```

The model setup is similar to the RNN.

```
model=Sequential()
model.add(LSTM(input_dim  = len(alphabet), output_dim = 100, return_sequences
     = True))
model.add(TimeDistributed(Dense(output_dim = len(alphabet), activation  =
     "sigmoid")))
model.compile(loss="binary_crossentropy",metrics=["accuracy"], optimizer =
     "adam")
```

We fit the model as long as it does not produce a 100 percent correct result.

```
n=1
while True:
        score = model.evaluate(X_test, y_test, batch_size=32)
        print "[Iteration %d] score=%f"%(n,score[1])
        if score[1] == 1.0:
            break
```

```
        n+=1
        model.fit(X_train, y_train, nb_epoch = 1, batch_size = 32)


preds=model.predict(X_test)[0]
for p in preds:
    m=np.argmax(p)
    print(enc.inverse_transform(m))

print(model.evaluate(X_test,y_test,batch_size=32))
```

The tricky task that this LSTM is performing is (as in the RNN example) the reconstruction of a piece of training data. In this case, the 52-character strings are

```
xyzaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa- axyz
pqraaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa- apqr
```

Take a close look at these strings. The final characters (*suffixes*, in NLP lingo) `-xyz` and `-pqr` are conditioned on the start of the strings (*prefixes*) `xyz` and `pqr-`. Notice that the suffixes are not directly conditioned on the preceding `…a` sequence; normally, an `a` is followed by an `a` in these strings. So the task is to keep the remote prefixes (`'xyz-','pqr-'`) in mind and do some counting of the intervening `a` characters to decide when to generate the suffixes, which are exact copies of the prefixes. Although such a task could be implemented much more easily with an explicit counter, the important takeaway here is that the network figures out by itself the moment to produce the suffixes and conditions the suffixes on old information (the prefixes). This takeway shows that the network has some memory capacity.

This seemingly simple task is actually quite complex in terms of bookkeeping, as you see. We feed the training and test strings to our original RNN, increasing the number of samples from 256 to 512. We're creating 512 batches of (in our case) 102 sequences—the size of our training samples, based on two strings of 52 characters, minus the first character of each.

Running the RNN for 100 iterations does not fare well. The RNN produces the output

```
-yzaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaayz
-qraaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaqr
```

which shows that it misses the first character of both suffixes (the `x` in `xyz` and the `p` in `pqr`). With an otherwise identical setup (100 iterations, 512 samples), the LSTM succeeds in reconstructing both strings flawlessly in fewer than 100 iterations:

```
-yzaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaxyz
-qraaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaapqr
```

2.2 Deep learning and NLP: a new paradigm

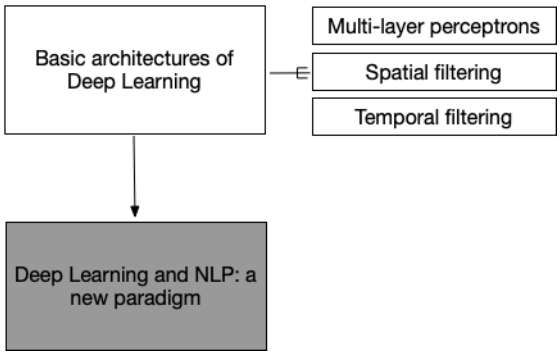


Figure 2.20 A new paradigm of deep learning and NLP

From a superficial point of view, deep learning appears to be a revamped, highly performant type of machine learning. When applied to language, however, deep learning brings two novel, useful ingredients to the table: temporal and spatial filtering. Language consisting of words arranged in sequences, flowing from the past into the present, the combination of spatial and temporal filtering opens a range of new, interesting possibilities for NLP. The important thing to realize is that we are working on abstract representations. Spatial processing of an abstract, intermediate layer representation eludes our imagination. Yet it is just another way of emphasizing of deemphasizing parts of a complex data representation. In a similar fashion, the temporal dimension allows us to gate historical data into this process of abstraction, learning to forget or keep certain parts of these complex representation.

We've demonstrated how to apply convolutional operations to vector representations of words, pretending that there was some spatial structure in the juxtaposition of those vectors. The RNN and LSTM layers gate historical information into the present. Nothing prevents us from applying CNN layers to the output of LSTM layers, which would result in spatial filtering of temporal, gated information or, vice versa, LSTM operations to CNN outputs.

The combinations for composing layers seem to be unbounded. In the image domain, some best practices for layer composition have been proposed in the literature based on myriad experiments.

Finally, language is not only sequential, but also *recursive*:

```
John says that Mary says that Bill told Richard he was on his way.
```

The ability to handle recursive structures with deep learning is an active topic of research and may be one of the approaches used to establish long-distance relations across words.

Summary

In this chapter, you learned

- The basic deep learning architectures: multilayer perceptrons, spatial (convolutional) and temporal (RNN and LSTM-based) filters.
- How to apply Convolutional and Recurrent Neural Networks, as well as Long Short Term-based networks, and how to code them in Keras.
- We have gone through a number of examples applied to language, from sentiment analysis to character prediction.

Equipped with this background, we are now ready to get our hands dirty on a number of real-life NLP problems. We will start in the next chapter with getting text into a deep learning model, using text embeddings.

Chapter 6 from *Natural Language Processing in Action* by Hobson Lane, Cole Howard, and Hannes Max Hapke

The "discovery" of word vectors in 2012 was the first application of deep learning to NLP that caught the imagination of researchers and developers. This chapter shows you how to model some of the fuzziness and subtlety of natural language words yet still produce logical answers to quantitative questions about words and their meaning. Word vectors not only help you to identify close synonyms and near-perfect antonyms, but also to visualize and compute a full spectrum of meaning for any given word or phrase. Word vectors help you find the word that's on the tip of the tongue and even find words you've never seen.

Reasoning with word vectors (Word2vec)

This chapter covers

- Understanding how word vectors are created
- Using pretrained models for your applications
- Reasoning with word vectors to solve real problems
- Visualizing word vectors
- Uncovering some surprising uses for word embeddings

One of the most exciting recent advancements in NLP is the "discovery" of word vectors. This chapter will help you understand what they are and how to use them to do some surprisingly powerful things. You'll learn how to recover some of the fuzziness and subtlety of word meaning that was lost in the approximations of earlier chapters.

In the previous chapters, we ignored the nearby context of a word. We ignored the words around each word. We ignored the effect the neighbors of a word have on its meaning and how those relationships affect the overall meaning of a

statement. Our bag-of-words concept jumbled all the words from each document together into a statistical bag. In this chapter, you'll create much smaller bags of words from a "neighborhood" of only a few words, typically fewer than 10 tokens. You'll also ensure that these neighborhoods of meaning don't spill over into adjacent sentences. This process will help focus your word vector training on the relevant words.

Our new word vectors will be able to identify synonyms, antonyms, or words that just belong to the same category, such as people, animals, places, plants, names, or concepts. We could do that before, with latent semantic analysis in chapter 4, but your tighter limits on a word's neighborhood will be reflected in tighter accuracy of the word vectors. Latent semantic analysis of words, *n*-grams, and documents didn't capture all the literal meanings of a word, much less the implied or hidden meanings. Some of the connotations of a word are lost with LSA's oversized bags of words.

WORD VECTORS *Word vectors* are numerical vector representations of word semantics, or meaning, including literal and implied meaning. So word vectors can capture the connotation of words, like "peopleness," "animalness," "placeness," "thingness," and even "conceptness." And they combine all that into a dense vector (no zeros) of floating point values. This dense vector enables queries and logical reasoning.

6.1 Semantic queries and analogies

Well, what are these awesome word vectors good for? Have you ever tried to recall a famous person's name but you only have a general impression of them, like maybe this:

She invented something to do with physics in Europe in the early 20th century.

If you enter that sentence into Google or Bing, you may not get the direct answer you're looking for, "Marie Curie." Google Search will most likely only give you links to lists of famous physicists, both men and women. You'd have to skim several pages to find the answer you're looking for. But once you found "Marie Curie," Google or Bing would keep note of that. They might get better at providing you search results the next time you look for a scientist.[1]

With word vectors, you can search for words or names that combine the meaning of the words "woman," "Europe," "physics," "scientist," and "famous," and that would get you close to the token "Marie Curie" that you're looking for. And all you have to do to make that happen is add up the word vectors for each of those words that you want to combine:

```
>>> answer_vector = wv['woman'] + wv['Europe'] + wv[physics'] +\
...     wv['scientist']
```

[1] At least, that's what it did for us in researching this book. We had to use private browser windows to ensure that your search results would be similar to ours.

In this chapter, we show you the exact way to do this query. And we even show you how to subtract gender bias from the word vectors used to compute your answer:

```
>>> answer_vector = wv['woman'] + wv['Europe'] + wv[physics'] +\
...     wv['scientist'] - wv['male'] - 2 * wv['man']
```

With word vectors, you can take the "man" out of "woman"!

6.1.1 Analogy questions

What if you could rephrase your question as an analogy question? What if your "query" was something like this:

Who is to nuclear physics what Louis Pasteur is to germs?

Again, Google Search, Bing, and even Duck Duck Go aren't much help with this one.[2] But with word vectors, the solution is as simple as subtracting "germs" from "Louis Pasteur" and then adding in some "physics":

```
>>> answer_vector = wv['Louis_Pasteur'] - wv['germs'] + wv['physics']
```

And if you're interested in trickier analogies about people in unrelated fields, such as musicians and scientists, you can do that, too:

Who is the Marie Curie of music?

or

Marie Curie is to science as who is to music?

Can you figure out what the word vector math would be for these questions?

You might have seen questions like these on the English analogy section of standardized tests such as SAT, ACT, or GRE exams. Sometimes they are written in formal mathematical notation like this:

```
MARIE CURIE : SCIENCE :: ? : MUSIC
```

Does that make it easier to guess the word vector math? One possibility is this:

```
>>> wv['Marie_Curie'] - wv['science'] + wv['music']
```

And you can answer questions like this for things other than people and occupations, like perhaps sports teams and cities:

The Timbers are to Portland as what is to Seattle?

In standardized test form, that's

```
TIMBERS : PORTLAND :: ? : SEATTLE
```

[2] Try them all if you don't believe us.

But, more commonly, standardized tests use English vocabulary words and ask less fun questions, like the following:

```
WALK : LEGS :: ? : MOUTH
```

or

```
ANALOGY : WORDS :: ? : NUMBERS
```

All those "tip of the tongue" questions are a piece of cake for word vectors, even though they aren't multiple choice. When you're trying to remember names or words, just thinking of the *A*, *B*, *C*, and *D* multiple choice options can be difficult. NLP comes to the rescue with word vectors.

Word vectors can answer these vague questions and analogy problems. Word vectors can help you remember any word or name on the tip of your tongue, as long as the word vector for the answer exists in your word vector vocabulary.[3] And word vectors work well even for questions that you can't even pose in the form of a search query or analogy. You can learn about some of this non-query math with word vectors in section 6.2.1.

6.2 Word vectors

In 2012, Thomas Mikolov, an intern at Microsoft, found a way to encode the meaning of words in a modest number of vector dimensions.[4] Mikolov trained a neural network[5] to predict word occurrences near each target word. In 2013, once at Google, Mikolov and his teammates released the software for creating these word vectors and called it Word2vec.[6]

Word2vec learns the meaning of words merely by processing a large corpus of unlabeled text. No one has to label the words in the Word2vec vocabulary. No one has to tell the Word2vec algorithm that Marie Curie is a scientist, that the Timbers are a soccer team, that Seattle is a city, or that Portland is a city in both Oregon and Maine. And no one has to tell Word2vec that soccer is a sport, or that a team is a group of people, or that cities are both places as well as communities. Word2vec can learn that and much more, all on its own! All you need is a corpus large enough to mention Marie Curie and Timbers and Portland near other words associated with science or soccer or cities.

This unsupervised nature of Word2vec is what makes it so powerful. The world is full of unlabeled, uncategorized, unstructured natural language text.

---

[3] For Google's pretrained word vector model, your word is almost certainly within the 100B word news feed that Google trained it on, unless your word was invented after 2013.

[4] Word vectors typically have 100 to 500 dimensions, depending on the breadth of information in the corpus used to train them.

[5] It's only a single-layer network, so almost any linear machine learning model will also work. Logistic regression, truncated SVD, linear discriminant analysis, and Naive Bayes would all work well.

[6] "Efficient Estimation of Word Representations in Vector Space," Sep 2013, Mikolov, Chen, Corrado, and Dean (https://arxiv.org/pdf/1301.3781.pdf).

Unsupervised learning and *supervised* learning are two radically different approaches to machine learning.

Supervised learning

In supervised learning, the training data must be labeled in some way. An example of a label is the spam categorical label on an SMS message in chapter 4. Another example is the quantitative value for the number of likes of a tweet. Supervised learning is what most people think of when they think of machine learning. A supervised model can only get better if it can measure the difference between the expected output (the label) and its predictions.

In contrast, unsupervised learning enables a machine to learn directly from data, without any assistance from humans. The training data doesn't have to be organized, structured, or labeled by a human. So unsupervised learning algorithms like Word2vec are perfect for natural language text.

Unsupervised learning

In unsupervised learning, you train the model to perform a task, but without any labels, only the raw data. Clustering algorithms such as k-means or DBSCAN are examples of unsupervised learning. Dimension reduction algorithms like principal component analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE) are also unsupervised machine learning techniques. In unsupervised learning, the model finds patterns in the relationships between the data points themselves. An unsupervised model can get smarter (more accurate) just by throwing more data at it.

Instead of trying to train a neural network to learn the target word meanings directly (on the basis of labels for that meaning), you teach the network to predict words near the target word in your sentences. So in this sense, you do have labels: the nearby words you're trying to predict. But because the labels are coming from the dataset itself and require no hand-labeling, the Word2vec training algorithm is definitely an unsupervised learning algorithm.

Another domain where this unsupervised training technique is used is in time series modeling. Time series models are often trained to predict the next value in a sequence based on a window of previous values. Time series problems are remarkably similar to natural language problems in a lot of ways, because they deal with ordered sequences of values (words or numbers).

And the prediction itself isn't what makes Word2vec work. The prediction is merely a means to an end. What you do care about is the internal representation, the vector that Word2vec gradually builds up to help it generate those predictions. This representation will capture much more of the meaning of the target word (its semantics) than the word-topic vectors that came out of latent semantic analysis and latent Dirichlet allocation in chapter 4.

NOTE Models that learn by trying to repredict the input using a lower-dimensional internal representation are called *autoencoders*. This may seem odd to you. It's like asking the machine to echo back what you just asked it, only it can't record the question as you're saying it. The machine has to compress your question into shorthand. And it has to use the same shorthand algorithm (function) for all the questions you ask it. The machine learns a new shorthand (vector) representation of your statements.

If you want to learn more about unsupervised deep learning models that create compressed representations of high-dimensional objects like words, search for the term "autoencoder."[7] They're also a common way to get started with neural nets, because they can be applied to almost any dataset.

Word2vec will learn about things you might not think to associate with all words. Did you know that every word has some geography, sentiment (positivity), and gender associated with it? If any word in your corpus has some quality, like "placeness," "peopleness," "conceptness," or "femaleness," all the other words will also be given a score for these qualities in your word vectors. The meaning of a word "rubs off" on the neighboring words when Word2vec learns word vectors.

All words in your corpus will be represented by numerical vectors, similar to the word-topic vectors discussed in chapter 4. Only this time the topics mean something more specific, more precise. In LSA, words only had to occur in the same document to have their meaning "rub off" on each other and get incorporated into their word-topic vectors. For Word2vec word vectors, the words must occur near each other—typically fewer than five words apart and within the same sentence. And Word2vec word vector topic weights can be added and subtracted to create new word vectors that mean something!

A mental model that may help you understand word vectors is to think of word vectors as a list of weights or scores. Each weight or score is associated with a specific dimension of meaning for that word. See the following listing.

Listing 6.1 Compute nessvector

```
>>> from nlpia.book.examples.ch06_nessvectors import *
>>> nessvector('Marie_Curie').round(2)
placeness      -0.46
peopleness      0.35
animalness      0.17
conceptness    -0.32
femaleness      0.26
```

Don't import this module unless you have a lot of RAM and a lot of time. The pretrained Word2vec model is huge.

I'm sure your nessvector dimensions will be much more fun and useful, like "trumpness" and "ghandiness."

[7] See the web page titled "Unsupervised Feature Learning and Deep Learning Tutorial" (http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/).

You can compute "nessvectors" for any word or *n*-gram in the Word2vec vocabulary using the tools from nlpia (https://github.com/totalgood/nlpia/blob/master/src/nlpia/book/examples/ch06_nessvectors.py). And this approach will work for any "ness" components that you can dream up.

Mikolov developed the Word2vec algorithm while trying to think of ways to numerically represent words in vectors. He wasn't satisfied with the less accurate word sentiment math you did in chapter 4. He wanted to do *vector-oriented reasoning*, like you just did in the previous section with those analogy questions. This concept may sound fancy, but really it means that you can do math with word vectors and that the answer makes sense when you translate the vectors back into words. You can add and subtract word vectors to *reason* about the words they represent and answer questions similar to your examples above, like the following:[8]

```
wv['Timbers'] - wv['Portland'] + wv['Seattle'] = ?
```

Ideally you'd like this math (word vector reasoning) to give you this:

```
wv['Seattle_Sounders']
```

Similarly, your analogy question "'Marie Curie' is to 'physics' as __ is to 'classical music'?" can be thought about as a math expression like this:

```
wv['Marie_Curie'] - wv['physics'] + wv['classical_music'] = ?
```

In this chapter, we want to improve on the LSA word vector representations we introduced in the previous chapter. Topic vectors constructed from entire documents using LSA are great for document classification, semantic search, and clustering. But the topic-word vectors that LSA produces aren't accurate enough to be used for semantic reasoning or classification and clustering of short phrases or compound words. You'll soon learn how to train the single-layer neural networks required to produce these more accurate and more fun word vectors. And you'll see why they have replaced LSA word-topic vectors for many applications involving short documents or statements.

6.2.1 Vector-oriented reasoning

Word2vec was first presented publicly in 2013 at the ACL conference.[9] The talk with the dry-sounding title "Linguistic Regularities in Continuous Space Word Representations" described a surprisingly accurate language model. Word2vec embeddings were four times more accurate (45%) compared to equivalent LSA models (11%) at answering analogy questions like those above.[10] The accuracy improvement was so surprising, in fact, that Mikolov's initial paper was rejected by the International Conference on Learn-

[8] For those not up on sports, the Portland Timbers and Seattle Sounders are major league soccer teams.

[9] See the PDF "Linguistic Regularities in Continuous Space Word Representations," by Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig (https://www.aclweb.org/anthology/N13-1090).

[10] See Radim Řehůřek's interview of Tomas Mikolov (https://rare-technologies.com/rrp#episode_1_tomas_mikolov_on_ai).

ing Representations.[11] Reviewers thought that the model's performance was too good to be true. It took nearly a year for Mikolov's team to release the source code and get accepted to the Association for Computational Linguistics.

Suddenly, with word vectors, questions like

```
Portland Timbers + Seattle - Portland = ?
```

can be solved with vector algebra (see figure 6.1).

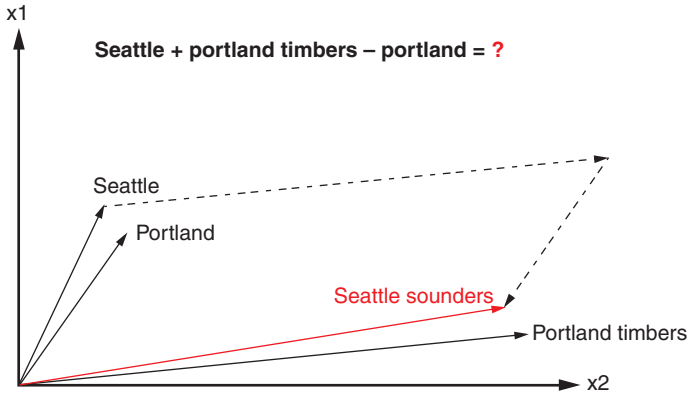


Figure 6.1 **Geometry of Word2vec math**

The Word2vec model contains information about the relationships between words, including similarity. The Word2vec model "knows" that the terms *Portland* and *Portland Timbers* are roughly the same distance apart as *Seattle* and *Seattle Sounders*. And those distances (differences between the pairs of vectors) are in roughly the same direction. So the Word2vec model can be used to answer your sports team analogy question. You can add the difference between *Portland* and *Seattle* to the vector that represents the *Portland Timbers*, which should get you close to the vector for the term *Seattle Sounders*:

$$\begin{bmatrix} 0.0168 \\ 0.007 \\ 0.247 \\ \dots \end{bmatrix} + \begin{bmatrix} 0.093 \\ -0.028 \\ -0.214 \\ \dots \end{bmatrix} - \begin{bmatrix} 0.104 \\ 0.0883 \\ -0.318 \\ \dots \end{bmatrix} = \begin{bmatrix} 0.006 \\ -0.109 \\ 0.352 \\ \dots \end{bmatrix}$$

Equation 6.1 ***Compute the answer to the soccer team question***

[11] See "ICRL2013 open review" (https://openreview.net/forum?id=idpCdOWtqXd60&noteId=C8Vn84fqSG8qa).

After adding and subtracting word vectors, your resultant vector will almost never exactly equal one of the vectors in your word vector vocabulary. Word2vec word vectors usually have 100s of dimensions, each with continuous real values. Nonetheless, the vector in your vocabulary that is closest to the resultant will often be the answer to your NLP question. The English word associated with that nearby vector is the natural language answer to your question about sports teams and cities.

Word2vec allows you to transform your natural language vectors of token occurrence counts and frequencies into the vector space of much lower-dimensional Word2vec vectors. In this lower-dimensional space, you can do your math and then convert back to a natural language space. You can imagine how useful this capability is to a chatbot, search engine, question answering system, or information extraction algorithm.

> **NOTE** The initial paper in 2013 by Mikolov and his colleagues was able to achieve an answer accuracy of only 40%. But back in 2013, the approach outperformed any other semantic reasoning approach by a significant margin. Since the initial publication, the performance of Word2vec has improved further. This was accomplished by training it on extremely large corpora. The reference implementation was trained on the 100 billion words from the Google News Corpus. This is the pretrained model you'll see used in this book a lot.

The research team also discovered that the difference between a singular and a plural word is often roughly the same magnitude, and in the same direction:

$$\vec{x}_{coffee} - \vec{x}_{coffees} \approx \vec{x}_{cup} - \vec{x}_{cups} \approx \vec{x}_{cookie} - \vec{x}_{cookies}$$

Equation 6.2 ***Distance between the singular and plural versions of a word***

But their discovery didn't stop there. They also discovered that the distance relationships go far beyond simple singular versus plural relationships. Distances apply to other semantic relationships. The Word2vec researchers soon discovered they could answer questions that involve geography, culture, and demographics, like this:

```
"San Francisco is to California as what is to Colorado?"

San Francisco - California + Colorado = Denver
```

MORE REASONS TO USE WORD VECTORS

Vector representations of words are useful not only for reasoning and analogy problems, but also for all the other things you use natural language vector space models for. From pattern matching to modeling and visualization, your NLP pipeline's accuracy and usefulness will improve if you know how to use the word vectors from this chapter.

For example, later in this chapter we show you how to visualize word vectors on 2D semantic maps like the one shown in figure 6.2. You can think of this like a cartoon

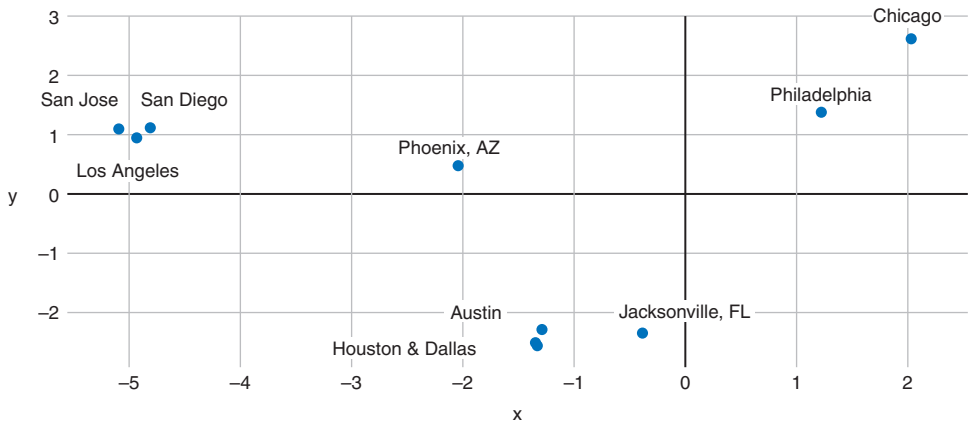


Figure 6.2 **Word vectors for ten US cities projected onto a 2D map**

map of a popular tourist destination or one of those impressionistic maps you see on bus stop posters. In these cartoon maps, things that are close to each other semantically as well as geographically get squished together. For cartoon maps, the artist adjusts the scale and position of icons for various locations to match the "feel" of the place. With word vectors, the machine too can have a feel for words and places and how far apart they should be. So your machine will be able to generate impressionistic maps like the one in figure 6.2 using word vectors you are learning about in this chapter.[12]

If you're familiar with these US cities, you might realize that this isn't an accurate geographic map, but it's a pretty good semantic map. I, for one, often confuse the two large Texas cities, Houston and Dallas, and they have almost identical word vectors. And the word vectors for the big California cities make a nice triangle of culture in my mind.

And word vectors are great for chatbots and search engines too. For these applications, word vectors can help overcome some of the rigidity, brittleness of pattern, or keyword matching. Say you were searching for information about a famous person from Houston, Texas, but didn't realize they'd moved to Dallas. From figure 6.2, you can see that a semantic search using word vectors could easily figure out a search involving city names such as Dallas and Houston. And even though character-based patterns wouldn't understand the difference between "tell me about a Denver omelette" and "tell me about the Denver Nuggets," a word vector pattern could. Patterns based on word vectors would likely be able to differentiate between the food item (omelette) and the basketball team (Nuggets) and respond appropriately to a user asking about either.

[12] You can find the code for generating these interactive 2D word plots at https://github.com/totalgood/nlpia/blob/master/src/nlpia/book/examples/ch06_w2v_us_cities_visualization.py.

6.2.2 How to compute Word2vec representations

Word vectors represent the semantic meaning of words as vectors in the context of the training corpus. This allows you not only to answer analogy questions but also reason about the meaning of words in more general ways with vector algebra. But how do you calculate these vector representations? There are two possible ways to train Word2vec embeddings:

- The *skip-gram* approach predicts the context of words (output words) from a word of interest (the input word).
- The *continuous bag-of-words* (CBOW) approach predicts the target word (the output word) from the nearby words (input words). We show you how and when to use each of these to train a Word2vec model in the coming sections.

The computation of the word vector representations can be resource intensive. Luckily, for most applications, you won't need to compute your own word vectors. You can rely on pretrained representations for a broad range of applications. Companies that deal with large corpora and can afford the computation have open sourced their pretrained word vector models. Later in this chapter we introduce you to using these other pretrained word models, such as GloVe and fastText.

TIP Pretrained word vector representations are available for corpora like Wikipedia, DBPedia, Twitter, and Freebase.[13] These pretrained models are great starting points for your word vector applications:

- Google provides a pretrained Word2vec model based on English Google News articles.[14]
- Facebook published their word model, called *fastText*, for 294 languages.[15]

But if your domain relies on specialized vocabulary or semantic relationships, general-purpose word models won't be sufficient. For example, if the word "python" should unambiguously represent the programming language instead of the reptile, a domain-specific word model is needed. If you need to constrain your word vectors to their usage in a particular domain, you'll need to train them on text from that domain.

SKIP-GRAM APPROACH

In the skip-gram training approach, you're trying to predict the surrounding window of words based on an input word. In the sentence about Monet, in our following example, "painted" is the training input to the neural network. The corresponding

[13] See the web page titled "GitHub - 3Top/word2vec-api: Simple web service providing a word embedding model" (https://github.com/3Top/word2vec-api#where-to-get-a-pretrained-model).

[14] Original Google 300-D Word2vec model on Google Drive (https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM).

[15] See the web page titled "GitHub - facebookresearch/fastText: Library for fast text representation and classification" (https://github.com/facebookresearch/fastText).

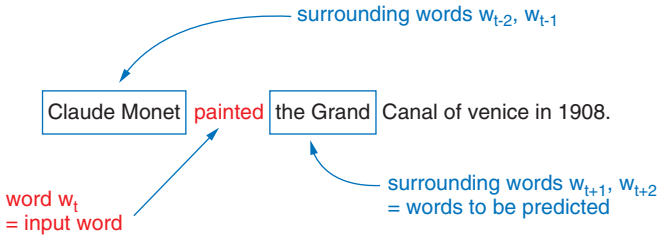


Figure 6.3 **Training input and output example for the skip-gram approach**

training output example skip-grams are shown in figure 6.3. The predicted words for these skip-grams are the neighboring words "Claude," "Monet," "the," and "Grand."

> **WHAT IS A SKIP-GRAM?** Skip-grams are *n*-grams that contain gaps because you skip over intervening tokens. In this example, you're predicting "Claude" from the input token "painted," and you skip over the token "Monet."

The structure of the neural network used to predict the surrounding words is similar to the networks you learned about in chapter 5. As you can see in figure 6.4, the network consists of two layers of weights, where the hidden layer consists of n neurons; n is the number of vector dimensions used to represent a word. Both the input and output layers contain M neurons, where M is the number of words in the model's vocabulary. The output layer activation function is a softmax, which is commonly used for classification problems.

WHAT IS SOFTMAX?

The softmax function is often used as the activation function in the output layer of neural networks when the network's goal is to learn classification problems. The softmax will squash the output results between 0 and 1, and the sum of all outputs will always add up to 1. That way, the results of an output layer with a softmax function can be considered as probabilities.

For each of the K output nodes, the softmax output value can be calculated using the normalized exponential function:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

If your output vector of a three-neuron output layer looks like this

$$v = \begin{bmatrix} 0.5 \\ 0.9 \\ 0.2 \end{bmatrix}$$

Equation 6.3 ***Example 3D vector***

The "squashed" vector after the softmax activation would look like this:

$$\sigma(v) = \begin{bmatrix} 0.309 \\ 0.461 \\ 0.229 \end{bmatrix}$$

Equation 6.4 ***Example 3D vector after softmax***

Notice that the sum of these values (rounded to three significant digits) is approximately 1.0, like a probability distribution.

Figure 6.4 shows the numerical network input and output for the first two surrounding words. In this case, the input word is "Monet," and the expected output of the network is either "Claude" or "painted," depending on the training pair.

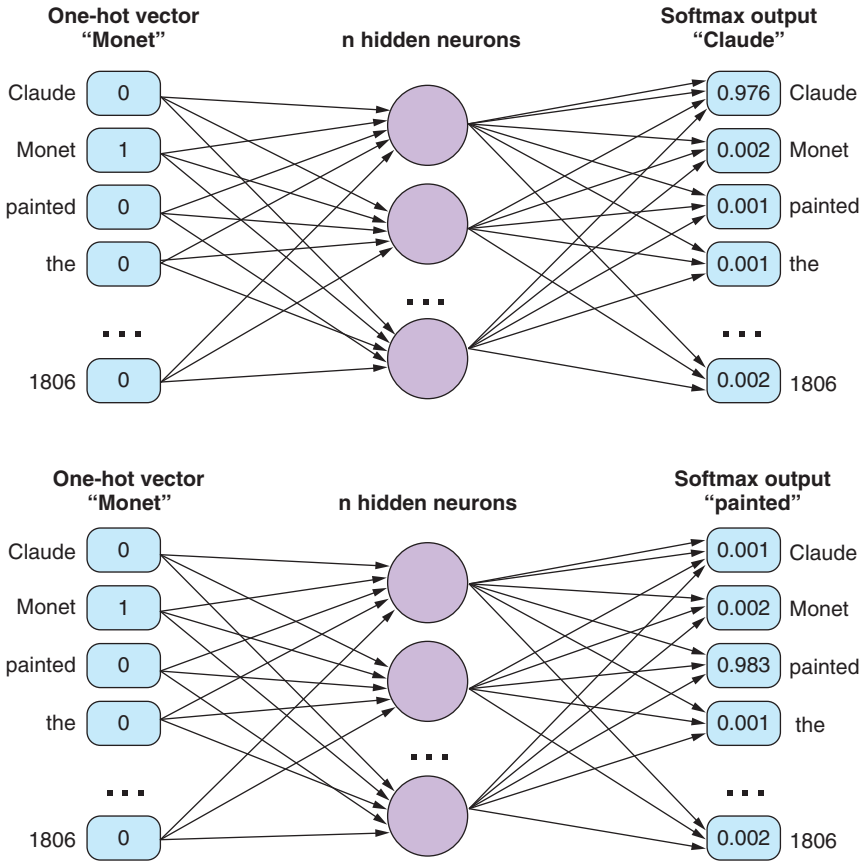


Figure 6.4 **Network example for the skip-gram training**

NOTE When you look at the structure of the neural network for word embedding, you'll notice that the implementation looks similar to what you discovered in chapter 5.

How does the network learn the vector representations?

To train a Word2vec model, you're using techniques from chapter 2. For example, in table 6.1, *wt* represents the one-hot vector for the token at position *t*. So if you want to train a Word2vec model using a skip-gram window size (radius) of two words, you're considering the two words before and after each target word. You would then use your 5-gram tokenizer from chapter 2 to turn a sentence like this

```
>>> sentence = "Claude Monet painted the Grand Canal of Venice in 1806."
```

into 10 5-grams with the input word at the center, one for each of the 10 words in the original sentence.

Table 6.1 Ten 5-grams for sentence about Monet

Input word *wt*	Expected output *wt-2*	Expected output *wt-1*	Expected output *wt+1*	Expected output *wt+2*
Claude			Monet	painted
Monet		Claude	painted	the
painted	Claude	Monet	the	Grand
the	Monet	painted	Grand	Canal
Grand	painted	the	Canal	of
Canal	the	Grand	of	Venice
of	Grand	Canal	Venice	in
Venice	Canal	of	in	1908
in	of	Venice	1908	
1908	Venice	in		

The training set consisting of the input word and the surrounding (output) words are now the basis for the training of the neural network. In the case of four surrounding words, you would use four training iterations, where each output word is being predicted based on the input word.

Each of the words are represented as one-hot vectors before they are presented to the network (see chapter 2). The output vector for a neural network doing embedding is similar to a one-hot vector as well. The softmax activation of the output layer nodes (one for each token in the vocabulary) calculates the probability of an output word being found as a surrounding word of the input word. The output vector of word probabilities can then be converted into a one-hot vector where the word with

the highest probability will be converted to 1, and all remaining terms will be set to 0. This simplifies the loss calculation.

After training of the neural network is completed, you'll notice that the weights have been trained to represent the semantic meaning. Thanks to the one-hot vector conversion of your tokens, each row in the weight matrix represents each word from the vocabulary for your corpus. After the training, semantically similar words will have similar vectors, because they were trained to predict similar surrounding words. *This is purely magical!*

After the training is complete and you decide not to train your word model any further, the output layer of the network can be ignored. Only the weights of the inputs to the hidden layer are used as the embeddings. Or in other words: the weight matrix is your word embedding. The dot product between the one-hot vector representing the input term and the weights then represents the *word vector embedding*.

Retrieving word vectors with linear algebra

The weights of a hidden layer in a neural network are often represented as a matrix: one column per input neuron, one row per output neuron. This allows the weight matrix to be multiplied by the column vector of inputs coming from the previous layer to generate a column vector of outputs going to the next layer (see figure 6.5). So if you multiply (dot product) a one-hot *row* vector by the trained weight matrix, you'll get a vector that is one weight from each neuron (from each matrix column). This also works if you take the weight matrix and multiply it (dot product) by a one-hot *column* vector for the word you are interested in.

Of course, the one-hot vector dot product just selects that row from your weight matrix that contains the weights for that word, which is your word vector. So you could easily retrieve that row by just selecting it, using the word's row number or index number from your vocabulary.

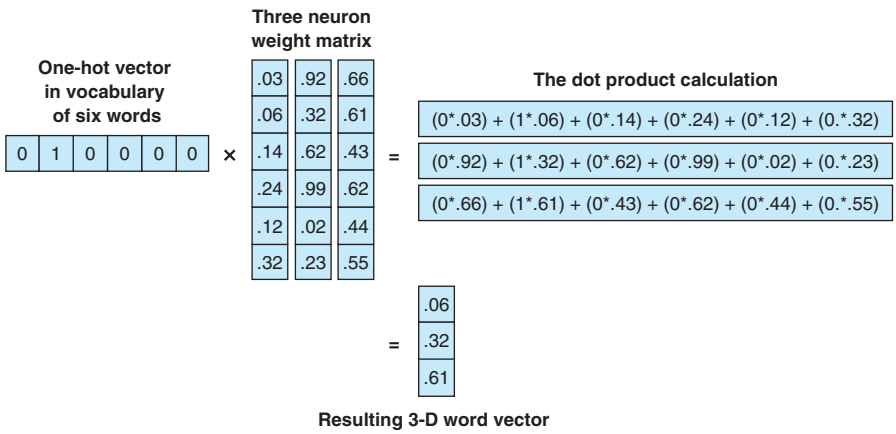


Figure 6.5 **Conversion of one-hot vector to word vector**

CONTINUOUS BAG-OF-WORDS APPROACH

In the continuous bag-of-words approach, you're trying to predict the center word based on the surrounding words (see figures 6.5 and 6.6 and table 6.2). Instead of creating pairs of input and output tokens, you'll create a multi-hot vector of all surrounding terms as an input vector. The multi-hot input vector is the sum of all one-hot vectors of the surrounding tokens to the center, target token.

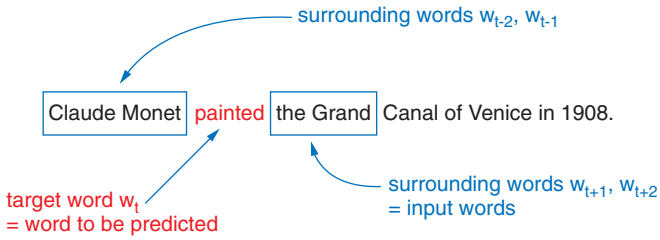


Figure 6.6 Training input and output example for the CBOW approach

Table 6.2 Ten CBOW 5-grams from sentence about Monet

Input word *wt-2*	Input word *wt-1*	Input word *wt+1*	Input word *wt+2*	Expected output *wt*
		Monet	painted	Claude
	Claude	painted	the	Monet
Claude	Monet	the	Grand	painted
Monet	painted	Grand	Canal	the
painted	the	Canal	of	Grand
the	Grand	of	Venice	Canal
Grand	Canal	Venice	in	of
Canal	of	in	1908	Venice
of	Venice	1908		in
Venice	in			1908

Based on the training sets, you can create your multi-hot vectors as inputs and map them to the target word as output. The multi-hot vector is the sum of the one-hot vectors of the surrounding words' training pairs *wt_2* + *wt_1* + *wt+1* + *wt+2*. You then build the training pairs with the multi-hot vector as the input and the target word *wt* as the output. During the training, the output is derived from the softmax of the output node with the highest probability (see figure 6.7).

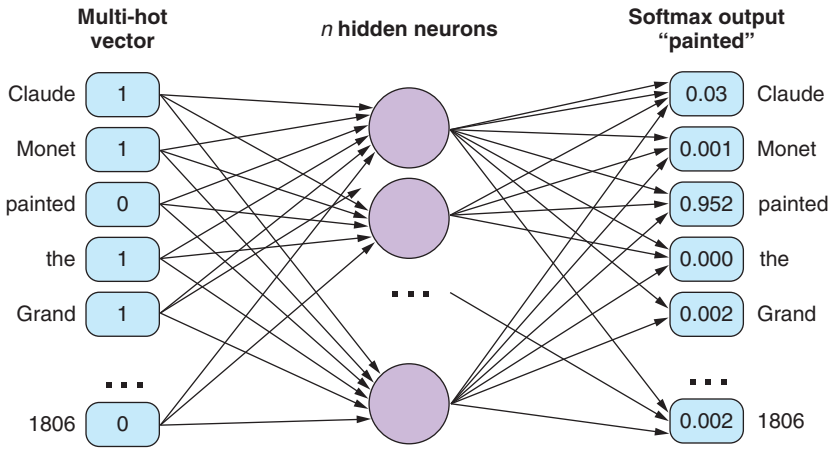
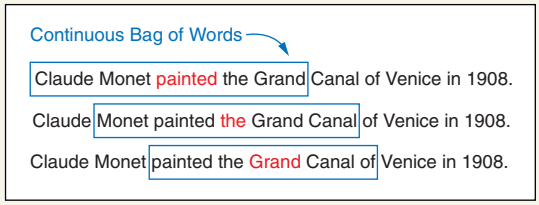


Figure 6.7 CBOW Word2vec network

Continuous bag of words vs. bag of words

In previous chapters, we introduced the concept of a bag of words, but how is it different than a continuous bag of words? To establish the relationships between words in a sentence you slide a rolling window across the sentence to select the surrounding words for the target word. All words within the sliding window are considered to be the content of the continuous bag of words for the target word at the middle of that window.



Example for a continuous bag of words passing a rolling window of five words over the sentence "Claude Monet painted the Grand Canal of Venice in 1908." The word painted is the target or center word within a five-word rolling window. "Claude," "Monet," "the," and "Grand" are the four surrounding words for the first CBOW rolling window.

SKIP-GRAM VS. CBOW: WHEN TO USE WHICH APPROACH

Mikolov highlighted that the skip-gram approach works well with small corpora and rare terms. With the skip-gram approach, you'll have more examples due to the network structure. But the continuous bag-of-words approach shows higher accuracies for frequent words and is much faster to train.

COMPUTATIONAL TRICKS OF WORD2VEC

After the initial publication, the performance of Word2vec models has been improved through various computational tricks. In this section, we highlight three improvements.

Frequent bigrams

Some words often occur in combination with other words—for example, "Elvis" is often followed by "Presley"—and therefore form bigrams. Since the word "Elvis" would occur with "Presley" with a high probability, you don't really gain much value from this prediction. In order to improve the accuracy of the Word2vec embedding, Mikolov's team included some bigrams and trigrams as terms in the Word2vec vocabulary. The team[16] used co-occurrence frequency to identify bigrams and trigrams that should be considered single terms, using the following scoring function:

$$score(w_i, w_j) = \frac{count(w_i, w_j) - \delta}{count(w_i) \times count(w_j)}$$

Equation 6.5 ***Bigram scoring function***

If the words *wi* and *wj* result in a high score and the score is above the threshold δ, they will be included in the Word2vec vocabulary as a pair term. You'll notice that the vocabulary of the model contains terms like "New_York" and "San_Francisco." The token of frequently occurring bigrams connects the two words with a character (usually "_"). That way, these terms will be represented as a single one-hot vector instead of two separate ones, such as for "San" and "Francisco."

Another effect of the word pairs is that the word combination often represents a different meaning than the individual words. For example, the MLS soccer team Portland Timbers has a different meaning than the individual words Portland and Timbers. But by adding oft-occurring bigrams like team names to the Word2vec model, they can easily be included in the one-hot vector for model training.

Subsampling frequent tokens

Another accuracy improvement to the original algorithm was to subsample frequent words. Common words like "the" or "a" often don't carry significant information. And the co-occurrence of the word "the" with a broad variety of other nouns in the corpus might create less meaningful connections between words, muddying the Word2vec representation with this false semantic similarity training.

> **IMPORTANT** All words carry meaning, including stop words. So stop words shouldn't be completely ignored or skipped while training your word vectors or composing your vocabulary. In addition, because word vectors are often used in generative models (like the model Cole used to compose sentences in

[16] The publication by the team around Tomas Mikolov (https://arxiv.org/pdf/1310.4546.pdf) provides more details.

this book), stop words and other common words must be included in your vocabulary and are allowed to affect the word vectors of their neighboring words.

To reduce the emphasis on frequent words like stop words, words are sampled during training in inverse proportion to their frequency. The effect of this is similar to the IDF effect on TF-IDF vectors. Frequent words are given less influence over the vector than the rarer words. Tomas Mikolov used the following equation to determine the probability of sampling a given word. This probability determines whether or not a particular word is included in a particular skip-gram during training:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

Equation 6.6 ***Subsampling probability in Mikolov's Word2vec paper***

The Word2vec C++ implementation uses a slightly different sampling probability than the one mentioned in the paper, but it has the same effect:

$$P(w_i) = \frac{f(w_i) - t}{f(w_i)} - \sqrt{\frac{t}{f(w_i)}}$$

Equation 6.7 ***Subsampling probability in Mikolov's Word2vec code***

In the preceding equations, $f(wi)$ represents the frequency of a word across the corpus, and t represents a frequency threshold above which you want to apply the subsampling probability. The threshold depends on your corpus size, average document length, and the variety of words used in those documents. Values between 10^{-5} and 10^{-6} are often found in the literature.

If a word shows up 10 times across your entire corpus, and your corpus has a vocabulary of one million distinct words, and you set the subsampling threshold to 10^{-6}, the probability of keeping the word in any particular *n*-gram is 68%. You would skip it 32% of the time while composing your *n*-grams during tokenization.

Mikolov showed that subsampling improves the accuracy of the word vectors for tasks such as answering analogy questions.

Negative sampling

One last trick Mikolov came up with was the idea of negative sampling. If a single training example with a pair of words is presented to the network, it'll cause all weights for the network to be updated. This changes the values of all the vectors for all the words in your vocabulary. But if your vocabulary contains thousands or millions of words, updating all the weights for the large one-hot vector is inefficient. To speed up the training of word vector models, Mikolov used negative sampling.

Instead of updating all word weights that weren't included in the word window, Mikolov suggested sampling just a few negative samples (in the output vector) to update their weights. Instead of updating all weights, you pick n negative example word pairs (words that don't match your target output for that example) and update the weights that contributed to their specific output. That way, the computation can be reduced dramatically and the performance of the trained network doesn't decrease significantly.

NOTE If you train your word model with a small corpus, you might want to use a negative sampling rate of 5 to 20 samples. For larger corpora and vocabularies, you can reduce the negative sample rate to as low as two to five samples, according to Mikolov and his team.

6.2.3 How to use the gensim.word2vec module

If the previous section sounded too complicated, don't worry. Various companies provide their pretrained word vector models, and popular NLP libraries for different programming languages allow you to use the pretrained models efficiently. In the following section, we look at how you can take advantage of the magic of word vectors. For word vectors you'll use the popular gensim library, which you first saw in chapter 4.

If you've already installed the nlpia package,[17] you can download a pretrained Word2vec model with the following command:

```
>>> from nlpia.data.loaders import get_data
>>> word_vectors = get_data('word2vec')
```

If that doesn't work for you, or you like to "roll your own," you can do a Google search for Word2vec models pretrained on Google News documents.[18] After you find and download the model in Google's original binary format and put it in a local path, you can load it with the gensim package like this:

```
>>> from gensim.models.keyedvectors import KeyedVectors
>>> word_vectors = KeyedVectors.load_word2vec_format(\
...     '/path/to/GoogleNews-vectors-negative300.bin.gz', binary=True)
```

Working with word vectors can be memory intensive. If your available memory is limited or if you don't want to wait minutes for the word vector model to load, you can reduce the number of words loaded into memory by passing in the `limit` keyword argument. In the following example, you'll load the 200k most common words from the Google News corpus:

```
>>> from gensim.models.keyedvectors import KeyedVectors
>>> word_vectors = KeyedVectors.load_word2vec_format(\
...     '/path/to/GoogleNews-vectors-negative300.bin.gz',
...         binary=True, limit=200000)
```

[17] See the README file at http://github.com/totalgood/nlpia for installation instructions.

[18] Google hosts the original model trained by Mikolov on Google Drive at https://bit.ly/GoogleNews-vectors-negative300.

But keep in mind that a word vector model with a limited vocabulary will lead to a lower performance of your NLP pipeline if your documents contain words that you haven't loaded word vectors for. Therefore, you probably only want to limit the size of your word vector model during the development phase. For the rest of the examples in this chapter, you should use the complete Word2vec model if you want to get the same results we show here.

The `gensim.KeyedVectors.most_similar()` method provides an efficient way to find the nearest neighbors for any given word vector. The keyword argument `positive` takes a list of the vectors to be added together, similar to your soccer team example from the beginning of this chapter. Similarly, you can use the `negative` argument for subtraction and to exclude unrelated terms. The argument `topn` determines how many related terms should be provided as a return value.

Unlike a conventional thesaurus, Word2vec synonomy (similarity) is a continuous score, a distance. This is because Word2vec itself is a continuous vector space model. Word2vec high dimensionality and continuous values for each dimension enable it to capture the full range of meaning for any given word. That's why analogies and even zeugmas, odd juxtopositions of multiple meanings within the same word, are no problem:[19]

```
>>> word_vectors.most_similar(positive=['cooking', 'potatoes'], topn=5)
[('cook', 0.6973530650138855),
 ('oven_roasting', 0.6754530668258667),
 ('Slow_cooker', 0.6742032170295715),
 ('sweet_potatoes', 0.6600279808044434),
 ('stir_fry_vegetables', 0.6548759341239929)]
>>> word_vectors.most_similar(positive=['germany', 'france'], topn=1)
[('europe', 0.7222039699554443)]
```

Word vector models also allow you to determine unrelated terms. The gensim library provides a method called `doesnt_match`:

```
>>> word_vectors.doesnt_match("potatoes milk cake computer".split())
'computer'
```

To determine the most unrelated term of the list, the method returns the term with the highest distance to all other list terms.

If you want to perform calculations (such as the famous example *king* + *woman* - *man* = *queen*, which was the example that got Mikolov and his advisor excited in the first place), you can do that by adding a `negative` argument to the `most_similar` method call:

```
>>> word_vectors.most_similar(positive=['king', 'woman'],
...     negative=['man'], topn=2)
[('queen', 0.7118192315101624), ('monarch', 0.6189674139022827)]
```

[19] *Surfaces and Essences: Analogy as the Fuel and Fire of Thinking* by Douglas Hoffstadter and Emmanuel Sander makes it clear why machines that can handle analogies and zeugmas are such a big deal.

The gensim library also allows you to calculate the similarity between two terms. If you want to compare two words and determine their cosine similarity, use the method `.similarity()`:

```
>>> word_vectors.similarity('princess', 'queen')
0.70705315983704509
```

If you want to develop your own functions and work with the raw word vectors, you can access them through Python's square bracket syntax (`[]`) or the `get()` method on a `KeyedVector` instance. You can treat the loaded model object as a dictionary where your word of interest is the dictionary key. Each float in the returned array represents one of the vector dimensions. In the case of Google's word model, your numpy arrays will have a shape of 1 x 300:

```
>>> word_vectors['phone']
array([-0.01446533, -0.12792969, -0.11572266, -0.22167969, -0.07373047,
       -0.05981445, -0.10009766, -0.06884766,  0.14941406,  0.10107422,
       -0.03076172, -0.03271484, -0.03125   , -0.10791016,  0.12158203,
        0.16015625,  0.19335938,  0.0065918 , -0.15429688,  0.03710938,
        ...
```

If you're wondering what all those numbers *mean*, you can find out. But it would take a lot of work. You would need to examine some synonyms and see which of the 300 numbers in the array they all share. Alternatively you can find the linear combination of these numbers that make up dimensions for things like "placeness" and "femaleness," like you did at the beginning of this chapter.

6.2.4 How to generate your own word vector representations

In some cases, you may want to create your own domain-specific word vector models. Doing so can improve the accuracy of your model if your NLP pipeline is processing documents that use words in a way that you wouldn't find on Google News before 2006, when Mikolov trained the reference Word2vec model. Keep in mind, you need a *lot* of documents to do this as well as Google and Mikolov did. But if your words are particularly rare on Google News, or your texts use them in unique ways within a restricted domain, such as medical texts or transcripts, a domain-specific word model may improve your model accuracy. In the following section, we show you how to train your own Word2vec model.

For the purpose of training a domain-specific Word2vec model, you'll again turn to gensim, but before you can start training the model, you'll need to preprocess your corpus using tools you discovered in chapter 2.

PREPROCESSING STEPS

First you need to break your documents into sentences and the sentences into tokens. The gensimword2vec model expects a list of sentences, where each sentence is broken up into tokens. This prevents word vectors learning from irrelevant word occurrences

in neighboring sentences. Your training input should look similar to the following structure:

```
>>> token_list
[
  ['to', 'provide', 'early', 'intervention/early', 'childhood', 'special',
   'education', 'services', 'to', 'eligible', 'children', 'and', 'their',
   'families'],
  ['essential', 'job', 'functions'],
  ['participate', 'as', 'a', 'transdisciplinary', 'team', 'member', 'to',
   'complete', 'educational', 'assessments', 'for']
  ...
]
```

To segment sentences and then convert sentences into tokens, you can apply the various strategies you learned in chapter 2. Detector Morse is a sentence segmenter that improves upon the accuracy segmenter available in NLTK and gensim for some applications.[20] Once you've converted your documents into lists of token lists (one for each sentence), you're ready for your Word2vec training.

TRAIN YOUR DOMAIN-SPECIFIC WORD2VEC MODEL

Get started by loading the Word2vec module:

```
>>> from gensim.models.word2vec import Word2Vec
```

The training requires a few setup details, shown in the following listing.

Listing 6.2 Parameters to control Word2vec model training

```
>>> num_features = 300
>>> min_word_count = 3
>>> num_workers = 2
>>> window_size = 6
>>> subsampling = 1e-3
```

Number of vector elements (dimensions) to represent the word vector

Min number of word count to be considered in the Word2vec model. If your corpus is small, reduce the min count. If you're training with a large corpus, increase the min count.

Number of CPU cores used for the training. If you want to set the number of cores dynamically, check out import multiprocessing: num_workers = multiprocessing.cpu_count().

Context window size

Subsampling rate for frequent terms

[20] Detector Morse, by Kyle Gorman and OHSU on pypi and at https://github.com/cslu-nlp/DetectorMorse, is a sentence segmenter with state-of-the-art performance (98%) and has been pretrained on sentences from years of text in the Wall Street Journal. So if your corpus includes language similar to that in the WSJ, Detector Morse is likely to give you the highest accuracy currently possible. You can also retrain Detector Morse on your own dataset if you have a large set of sentences from your domain.

Now you're ready to start your training, using the following listing.

Listing 6.3 Instantiating a Word2vec model

```
>>> model = Word2Vec(
...     token_list,
...     workers=num_workers,
...     size=num_features,
...     min_count=min_word_count,
...     window=window_size,
...     sample=subsampling)
```

Depending on your corpus size and your CPU performance, the training will take a significant amount of time. For smaller corpora, the training can be completed in minutes. But for a comprehensive word model, the corpus will contain millions of sentences. You need to have several examples of all the different ways the different words in your corpus are used. If you start processing larger corpora, such as the Wikipedia corpus, expect a much longer training time and a much larger memory consumption.

Word2vec models can consume quite a bit of memory. But remember that only the weight matrix for the hidden layer is of interest. Once you've trained your word model, you can reduce the memory footprint by about half if you freeze your model and discard the unnecessary information. The following command will discard the unneeded output weights of your neural network:

```
>>> model.init_sims(replace=True)
```

The `init_sims` method will freeze the model, storing the weights of the hidden layer and discarding the output weights that predict word co-ocurrences. The output weights aren't part of the vector used for most Word2vec applications. But the model cannot be trained further once the weights of the output layer have been discarded.

You can save the trained model with the following command and preserve it for later use:

```
>>> model_name = "my_domain_specific_word2vec_model"
>>> model.save(model_name)
```

If you want to test your newly trained model, you can use it with the same method you learned in the previous section; use the following listing.

Listing 6.4 Loading a saved Word2vec model

```
>>> from gensim.models.word2vec import Word2Vec
>>> model_name = "my_domain_specific_word2vec_model"
>>> model = Word2Vec.load(model_name)
>>> model.most_similar('radiology')
```

6.2.5 Word2vec vs. GloVe (Global Vectors)

Word2vec was a breakthrough, but it relies on a neural network model that must be trained using backpropagation. Backpropagation is usually less efficient than direct optimization of a cost function using gradient descent. Stanford NLP researchers[21] led by Jeffrey Pennington set about to understand the reason why Word2vec worked so well and to find the cost function that was being optimized. They started by counting the word co-occurrences and recording them in a square matrix. They found they could compute the singular value decomposition[22] of this co-occurrence matrix, splitting it into the same two weight matrices that Word2vec produces.[23] The key was to normalize the co-occurrence matrix the same way. But in some cases the Word2vec model failed to converge to the same global optimum that the Stanford researchers were able to achieve with their SVD approach. It's this direct optimization of the global vectors of word co-occurrences (co-occurrences across the entire corpus) that gives GloVe its name.

GloVe can produce matrices equivalent to the input weight matrix and output weight matrix of Word2vec, producing a language model with the same accuracy as Word2vec but in much less time. GloVe speeds the process by using the text data more efficiently. GloVe can be trained on smaller corpora and still converge.[24] And SVD algorithms have been refined for decades, so GloVe has a head start on debugging and algorithm optimization. Word2vec relies on backpropagation to update the weights that form the word embeddings. Neural network backpropagation is less efficient than more mature optimization algorithms such as those used within SVD for GloVe.

Even though Word2vec first popularized the concept of semantic reasoning with word vectors, your workhorse should probably be GloVe to train new word vector models. With GloVe you'll be more likely to find the global optimum for those vector representations, giving you more accurate results.

Advantages of GloVe are

- Faster training
- Better RAM/CPU efficiency (can handle larger documents)
- More efficient use of data (helps with smaller corpora)
- More accurate for the same amount of training

6.2.6 fastText

Researchers from Facebook took the concept of Word2vec one step further[25] by adding a new twist to the model training. The new algorithm, which they named fastText,

[21] Stanford GloVe Project (https://nlp.stanford.edu/projects/glove/).
[22] See chapter 5 and appendix C for more details on SVD.
[23] *GloVe: Global Vectors for Word Representation*, by Jeffrey Pennington, Richard Socher, and Christopher D. Manning: https://nlp.stanford.edu/pubs/glove.pdf.
[24] Gensim's comparison of Word2vec and GloVe performance: https://rare-technologies.com/making-sense-of-Word2vec/#glove_vs_word2vec.
[25] "Enriching Word Vectors with Subword Information," Bojanowski et al.: https://arxiv.org/pdf/1607.04606.pdf.

predicts the surrounding *n-character* grams rather than just the surrounding words, like Word2vec does. For example, the word "whisper" would generate the following 2- and 3-character grams:

wh, whi, hi, his, is, isp, sp, spe, pe, per, er

fastText trains a vector representation for every *n*-character gram, which includes words, misspelled words, partial words, and even single characters. The advantage of this approach is that it handles rare words much better than the original Word2vec approach.

As part of the fastText release, Facebook published pretrained fastText models for 294 languages. On the Github page of Facebook research,[26] you can find models ranging from *Abkhazian* to *Zulu*. The model collection even includes rare languages such as *Saterland Frisian*, which is only spoken by a handful of Germans. The pretrained fastText models provided by Facebook have only been trained on the available Wikipedia corpora. Therefore the vocabulary and accuracy of the models will vary across languages.

How to use the pretrained fastText models

The use of fastText is just like using Google's Word2vec model. Head over to the fastText model repository and download the bin+text model for your language of choice. After the download finishes, unzip the binary language file.[27] With the following code, you can then load it into gensim:

```
>>> from gensim.models.fasttext import FastText
>>> ft_model = FastText.load_fasttext_format(\
...     model_file=MODEL_PATH)
>>> ft_model.most_similar('soccer')
```

If you're using a gensim version before 3.2.0, you need to change this line to from gensim.models.wrappers.fasttext import FastText.

The model_file points to the directory where you stored the model's bin and vec files.

After loading the model, use it like any other word model in gensim.

The gensim fastText API shares a lot of functionality with the Word2vec implementations. All methods you learned about earlier in this chapter also apply to the fastText models.

6.2.7 Word2vec vs. LSA

You might now be wondering how Word2vec and GloVe word vectors compare to the LSA topic-word vectors of chapter 4. Even though we didn't say much about the LSA topic-document vectors in chapter 4, LSA gives you those, too. LSA topic-document

[26] See the web page titled "fastText/pretrained-vectors.md at master" (https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md).

[27] The en.wiki.zip file is 9.6GB.

vectors are the sum of the topic-word vectors for all the words in those documents. If you wanted to get a word vector for an entire document that is analogous to topic-document vectors, you'd sum all the Word2vec word vectors in each document. That's pretty close to how Doc2vec document vectors work. We show you those a bit later in this chapter.

If your LSA matrix of topic vectors is of size $N_{\{words\}} * N_{\{topics\}}$, the LSA word vectors are the rows of that LSA matrix. These row vectors capture the meaning of words in a sequence of around 200 to 300 real values, like Word2vec does. And LSA topic-word vectors are useful for finding both related and unrelated terms. As you learned in the GloVe discussion, Word2vec vectors can be created using the exact same SVD algorithm used for LSA. But Word2vec gets more use out of the same number of words in its documents by creating a sliding window that overlaps from one document to the next. This way it can reuse the same words five times before sliding on.

What about incremental or online training? Both LSA and Word2vec algorithms allow adding new documents to your corpus and adjusting your existing word vectors to account for the co-occurrences in the new documents. But only the existing bins in your lexicon can be updated. Adding completely new words would change the total size of your vocabulary and therefore your one-hot vectors would change. That requires starting the training over if you want to capture the new word in your model.

LSA trains faster than Word2vec does. And for long documents, it does a better job of discriminating and clustering those documents.

The "killer app" for Word2vec is the semantic reasoning it popularized. LSA topic-word vectors can do that, too, but it usually isn't accurate. You'd have to break documents into sentences and then only use short phrases to train your LSA model if you want to approach the accuracy and "wow" factor of Word2vec reasoning. With Word2vec you can determine the answer to questions like *Harry Potter* + *University* = *Hogwarts*.[28]

Advantages of LSA are

- Faster training
- Better discrimination between longer documents

Advantages of Word2vec and GloVe are

- More efficient use of large corpora
- More accurate reasoning with words, such as answering analogy questions

6.2.8 Visualizing word relationships

The semantic word relationships can be powerful and their visualizations can lead to interesting discoveries. In this section, we demonstrate steps to visualize the word vectors in 2D.

[28] As a great example for domain-specific Word2vec models, check out the models around Harry Potter, the Lord of the Rings, and so on at https://github.com/nchah/word2vec4everything#harry-potter.

NOTE If you need a quick visualization of your word model, we highly recommend using Google's TensorBoard word embedding visualization functionality. For more details, check out the section "How to visualize word embeddings" in chapter 13.

To get started, let's load all the word vectors from the Google Word2vec model of the Google News corpus. As you can imagine, this corpus included a lot of mentions of Portland and Oregon and a lot of other city and state names. You'll use the nlpia package to keep things simple, so you can start playing with Word2vec vectors quickly. See the following listing.

Listing 6.5 Load a pretrained Word2vec model using nlpia

```
>>> import os
>>> from nlpia.loaders import get_data
>>> from gensim.models.word2vec import KeyedVectors
>>> wv = get_data('word2vec')
>>> len(wv.vocab)
3000000
```

Downloads the pretrained Google News word vectors to nlpia/src/nlpia/bigdata/GoogleNews-vectors-negative300.bin.gz

WARNING The Google News Word2vec model is huge: three million words with 300 vector dimensions each. The complete word vector model requires 3 GB of available memory. If your available memory is limited or you quickly want to load a few most frequent terms from the word model, check out chapter 13.

This `KeyedVectors` object in gensim now holds a table of three million Word2vec vectors. We loaded these vectors from a file created by Google to store a Word2vec model that they trained on a large corpus based on Google News articles. There should definitely be a lot of words for states and cities in all those news articles. The following listing shows just a few of the words in the vocabulary, starting at the one millionth word.

Listing 6.6 Examine Word2vec vocabulary frequencies

```
>>> import pandas as pd
>>> vocab = pd.Series(wv.vocab)
>>> vocab.iloc[1000000:100006]
Illington_Fund             Vocab(count:447860, index:2552140)
Illingworth                 Vocab(count:2905166, index:94834)
Illingworth_Halifax       Vocab(count:1984281, index:1015719)
Illini                       Vocab(count:2984391, index:15609)
IlliniBoard.com           Vocab(count:1481047, index:1518953)
Illini_Bluffs              Vocab(count:2636947, index:363053)
```

Notice that compound words and common *n*-grams are joined together with an underscore character ("_"). Also notice that the value in the key-value mapping is a `gensimVocab` object that contains not only the index location for a word, so you can retrieve the Word2vec vector, but also the number of times it occurred in the Google News corpus.

As you've seen earlier, if you want to retrieve the 300-D vector for a particular word, you can use the square brackets on this `KeyedVectors` object to `.__getitem__()` any word or *n*-gram:

```
>>> wv['Illini']
array([ 0.15625   ,  0.18652344,  0.33203125,  0.55859375,  0.03637695,
       -0.09375   , -0.05029297,  0.16796875, -0.0625    ,  0.09912109,
       -0.0291748 ,  0.39257812,  0.05395508,  0.35351562, -0.02270508,
       ...
```

The reason we chose the one millionth word (in lexical alphabetic order) is because the first several thousand "words" are punctuation sequences like "#" and other symbols that occurred a lot in the Google News corpus. We just got lucky that "Illini"[29] showed up in this list. Let's see how close this "Illini" vector is to the vector for "Illinois," shown in the following listing.

Listing 6.7 Distance between "Illinois" and "Illini"

```
>>> import numpy as np
>>> np.linalg.norm(wv['Illinois'] - wv['Illini'])
3.3653798
>>> cos_similarity = np.dot(wv['Illinois'], wv['Illini']) / (
...     np.linalg.norm(wv['Illinois']) *\
...     np.linalg.norm(wv['Illini']))
>>> cos_similarity
0.5501352
>>> 1 - cos_similarity
0.4498648
```

Euclidean distance

Cosine similarity is the normalized dot product

Cosine distance

These distances mean that the words "Illini" and "Illinois" are only moderately close to one another in meaning.

Now let's retrieve all the Word2vec vectors for US cities so you can use their distances to plot them on a 2D map of meaning. How would you find all the cities and states in that Word2vec vocabulary in that `KeyedVectors` object? You could use cosine distance like you did in the previous listing to find all the vectors that are close to the words "state" or "city". But rather than reading through all three million words and word vectors, let's load another dataset containing a list of cities and states (regions) from around the world, as shown in the following listing.

Listing 6.8 Some US city data

```
>>> from nlpia.data.loaders import get_data
>>> cities = get_data('cities')
>>> cities.head(1).T
geonameid                     3039154
name                        El Tarter
```

[29] The word "Illini" refers to a group of people, usually football players and fans, rather than a single geographic region like "Illinois" (where most fans of the "Fighting Illini" live).

```
asciiname                      El Tarter
alternatenames     Ehl Tarter,•• ••••••
latitude                         42.5795
longitude                        1.65362
feature_class                          P
feature_code                         PPL
country_code                          AD
cc2                                  NaN
admin1_code                           02
admin2_code                          NaN
admin3_code                          NaN
admin4_code                          NaN
population                          1052
elevation                            NaN
dem                                 1721
timezone                  Europe/Andorra
modification_date             2012-11-03
```

This dataset from Geocities contains a lot of information, including latitude, longitude, and population. You could use this for some fun visualizations or comparisons between geographic distance and Word2vec distance. But for now you're just going to try to map that Word2vec distance on a 2D plane and see what it looks like. Let's focus on just the United States for now, as shown in the following listing.

Listing 6.9 Some US state data

```
>>> us = cities[(cities.country_code == 'US') &\
...      (cities.admin1_code.notnull())].copy()
>>> states = pd.read_csv(\
...      'http://www.fonz.net/blog/wp-content/uploads/2008/04/states.csv')
>>> states = dict(zip(states.Abbreviation, states.State))
>>> us['city'] = us.name.copy()
>>> us['st'] = us.admin1_code.copy()
>>> us['state'] = us.st.map(states)
>>> us[us.columns[-3:]].head()
                     city  st    state
geonameid
4046255       Bay Minette  AL  Alabama
4046274              Edna  TX    Texas
4046319    Bayou La Batre  AL  Alabama
4046332         Henderson  TX    Texas
4046430           Natalia  TX    Texas
```

Now you have a full state name for each city in addition to its abbreviation. Let's check to see which of those state names and city names exist in your Word2vec vocabulary:

```
>>> vocab = pd.np.concatenate([us.city, us.st, us.state])
>>> vocab = np.array([word for word in vocab if word in wv.wv])
>>> vocab[:5]
array(['Edna', 'Henderson', 'Natalia', 'Yorktown', 'Brighton'])
```

Even when you only look at United States cities, you'll find a lot of large cities with the same name, like Portland, Oregon and Portland, Maine. So let's incorporate into your

city vector the essence of the state where that city is located. To combine the meanings of words in Word2vec, you add the vectors together. That's the magic of vector-oriented reasoning. Here's one way to add the Word2vecs for the states to the vectors for the cities and put all these new vectors in a big DataFrame. We use either the full name of a state or just the abbreviations (whichever one is in your Word2vec vocabulary), as shown in the following listing.

Listing 6.10 Augment city word vectors with US state word vectors

```
>>> city_plus_state = []
>>> for c, state, st in zip(us.city, us.state, us.st):
...     if c not in vocab:
...         continue
...     row = []
...     if state in vocab:
...         row.extend(wv[c] + wv[state])
...     else:
...         row.extend(wv[c] + wv[st])
...     city_plus_state.append(row)
>>> us_300D = pd.DataFrame(city_plus_state)
```

Depending on your corpus, your word relationship can represent different attributes, such as geographical proximity or cultural or economic similarities. But the relationships heavily depend on the training corpus, and they will reflect the corpus.

Word vectors are biased!

Word vectors learn word relationships based on the training corpus. If your corpus is about finance then your "bank" word vector will be mainly about businesses that hold deposits. If your corpus is about geology, then your "bank" word vector will be trained on associations with rivers and streams. And if you corpus is mostly about a matriarchal society with women bankers and men washing clothes in the river, then your word vectors would take on that gender bias.

The following example shows the gender bias of a word model trained on Google News articles. If you calculate the distance between "man" and "nurse" and compare that to the distance between "woman" and "nurse," you'll be able to see the bias:

```
>>> word_model.distance('man', 'nurse')
0.7453
>>> word_model.distance('woman', 'nurse')
0.5586
```

Identifying and compensating for biases like this is a challenge for any NLP practitioner that trains her models on documents written in a biased world.

The news articles used as the training corpus share a common component, which is the semantical similarity of the cities. Semantically similar locations in the articles seem to be interchangeable and therefore the word model learned that they are similar. If you had trained on a different corpus, your word relationships might have dif-

fered. In this news corpus, cities that are similar in size and culture are clustered close together despite being far apart geographically, such as San Diego and San Jose, or vacation destinations such as Honolulu and Reno.

Fortunately you can use conventional algebra to add the vectors for cities to the vectors for states and state abbreviations. As you discovered in chapter 4, you can use tools such as principal components analysis to reduce the vector dimensions from your 300 dimensions to a human-understandable 2D representation. PCA enables you to see the projection or "shadow" of these 300-D vectors in a 2D plot. Best of all, the PCA algorithm ensures that this projection is the best possible view of your data, keeping the vectors as far apart as possible. PCA is like a good photographer that looks at something from every possible angle before composing the optimal photograph. You don't even have to normalize the length of the vectors after summing the city + state + abbrev vectors, because PCA takes care of that for you.

We saved these augmented city word vectors in the nlpia package so you can load them to use in your application. In the following code, you use PCA to project them onto a 2D plot.

Listing 6.11 Bubble chart of US cities

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> us_300D = get_data('cities_us_wordvectors')
>>> us_2D = pca.fit_transform(us_300D.iloc[:, :300])
```

The 2D vectors producted by PCA are for visualization. Retain the original 300-D Word2vec vectors for any vector reasoning you might want to do.

The last column of this DataFrame contains the city name, which is also stored in the DataFrame index.

Figure 6.8 shows the 2D projection of all these 300-D word vectors for US cities:

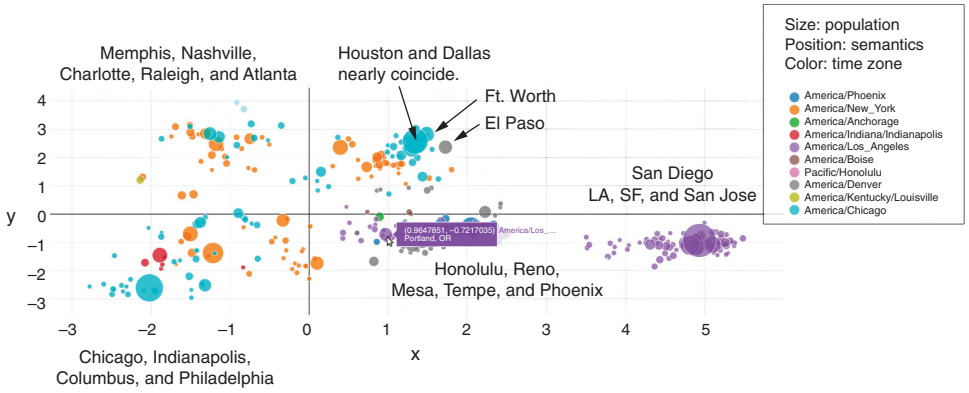


Figure 6.8 Google News Word2vec 300-D vectors projected onto a 2D map using PCA

NOTE Low semantic distance (distance values close to zero) represents high similarity between words. The semantic distance, or "meaning" distance, is determined by the words occurring nearby in the documents used for training. The Word2vec vectors for two terms are *close* to each other in word vector space if they are often used in similar contexts (used with similar words nearby). For example San Francisco is *close* to California because they often occur nearby in sentences and the distribution of words used near them is similar. A large distance between two terms expresses a low likelihood of shared context and shared meaning (they are semantically dissimilar), such as cars and peanuts.

If you'd like to explore the city map shown in figure 6.8, or try your hand at plotting some vectors of your own, listing 6.12 shows you how. We built a wrapper for Plotly's offline plotting API that should help it handle DataFrames where you've denormalized your data. The Plotly wrapper expects a DataFrame with a row for each sample and columns for features you'd like to plot. These can be categorical features (such as time zones) and continuous real-valued features (such as city population). The resulting plots are interactive and useful for exploring many types of machine learning data, especially vector-representations of complex things such as words and documents.

Listing 6.12 Bubble plot of US city word vectors

```
>>> import seaborn
>>> from matplotlib import pyplot as plt
>>> from nlpia.plots import offline_plotly_scatter_bubble
>>> df = get_data('cities_us_wordvectors_pca2_meta')
>>> html = offline_plotly_scatter_bubble(
...     df.sort_values('population', ascending=False)[:350].copy()\
...         .sort_values('population'),
...     filename='plotly_scatter_bubble.html',
...     x='x', y='y',
...     size_col='population', text_col='name', category_col='timezone',
...     xscale=None, yscale=None,  # 'log' or None
...     layout={}, marker={'sizeref': 3000})
{'sizemode': 'area', 'sizeref': 3000}
```

To produce the 2D representations of your 300-D word vectors, you need to use a dimension reduction technique. We used PCA. To reduce the amount of information lost during the compression from 300-D to 2D, reducing the range of information contained in the input vectors also helps. So you limited your word vectors to those associated with cities. This is like limiting the domain or subject matter of a corpus when computing TF-IDF or BOW vectors.

For a more diverse mix of vectors with greater information content, you'll probably need a nonlinear embedding algorithm such as t-SNE. We talk about t-SNE and other neural net techniques in later chapters. t-SNE will make more sense once you've grasped the word vector embedding algorithms here.

6.2.9 Unnatural words

Word embeddings such as Word2vec are useful not only for English words but also for any sequence of symbols where the sequence and proximity of symbols is representative of their meaning. If your symbols have semantics, embeddings may be useful. As you may have guessed, word embeddings also work for languages other than English.

Embedding works also for pictorial languages such as traditional Chinese and Japanese (Kanji) or the mysterious hieroglyphics in Egyptian tombs. Embeddings and vector-based reasoning even works for languages that attempt to obfuscate the meaning of words. You can do vector-based reasoning on a large collection of "secret" messages transcribed from "Pig Latin" or any other language invented by children or the Emperor of Rome. A *Caesar cipher*[30] such as RO13 or a *substitution cipher*[31] are both vulnerable to vector-based reasoning with Word2vec. You don't even need a decoder ring (shown in figure 6.9). You need only a large collection of messages or *n*-grams that your Word2vec embedder can process to find co-occurrences of words or symbols.

Word2vec has even been used to glean information and relationships from unnatural words or ID numbers such as college course numbers (CS-101), model numbers (Koala E7270 or Galaga Pro), and even serial numbers, phone numbers, and ZIP codes.[32] To get the most useful information about the relationship between ID numbers like this, you'll need a variety of sentences that contain those ID numbers. And if

Figure 6.9 Decoder rings (left: Hubert Berberich (HubiB) (https://commons.wikimedia.org/wiki/File:CipherDisk2000.jpg), CipherDisk2000, marked as public domain, more details on Wikimedia Commons: https://commons.wikimedia.org/wiki/Template:PD-self; middle: Cory Doctorow (https://www.flickr.com/photos/doctorow/2817314740/in/photostream/), Crypto wedding-ring 2, https://creativecommons.org/licenses/by-sa/2.0/legalcode; right: Sobebunny (https://commons.wikimedia.org/wiki/File:Captain-midnight-decoder.jpg), Captain-midnight-decoder, https://creativecommons.org/licenses/by-sa/3.0/legalcode)

[30] See the web page titled "Caesar cipher" (https://en.wikipedia.org/wiki/Caesar_cipher).
[31] See the web page titled "Substitution cipher" (https://en.wikipedia.org/wiki/Substitution_ cipher).
[32] See the web page titled "A non-NLP application of Word2Vec – Towards Data Science" (https://medium.com/towards-data-science/a-non-nlp-application-of-word2vec-c637e35d3668).

the ID numbers often contain a structure where the position of a symbol has meaning, it can help to tokenize these ID numbers into their smallest semantic packet (such as words or syllables in natural languages).

6.2.10 Document similarity with Doc2vec

The concept of Word2vec can also be extended to sentences, paragraphs, or entire documents. The idea of predicting the next word based on the previous words can be extended by training a paragraph or document vector (see figure 6.10).[33] In this case, the prediction not only considers the previous words, but also the vector representing the paragraph or the document. It can be considered as an additional word input to the prediction. Over time, the algorithm learns a document or paragraph representation from the training set.

How are document vectors generated for unseen documents after the training phase? During the *inference stage*, the algorithm adds more document vectors to the document matrix and computes the added vector based on the *frozen* word vector matrix, and its weights. By inferring a document vector, you can now create a semantic representation of the whole document.

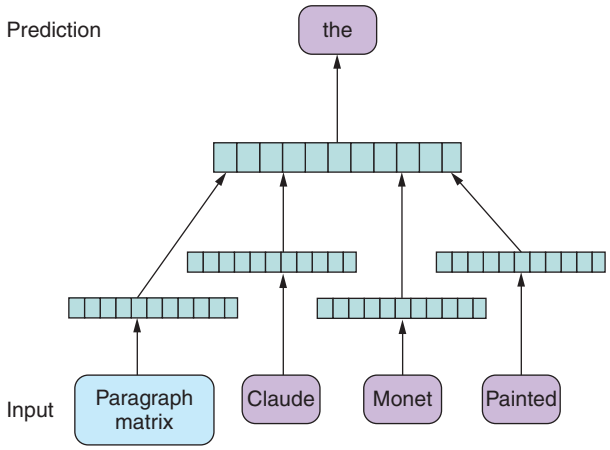


Figure 6.10 Doc2vec training uses an additional document vector as input.

By expanding the concept of Word2vec with an additional document or paragraph vector used for the word prediction, you can now use the trained document vector for various tasks, such as finding similar documents in a corpus.

HOW TO TRAIN DOCUMENT VECTORS

Similar to your training of word vectors, you're using the gensim package to train document vectors, as shown in the following listing.

[33] See the web page titled "Distributed Representations of Sentences and Documents" (https://arxiv.org/pdf/1405.4053v2.pdf).

Listing 6.13 Train your own document and word vectors

gensim uses Python's multiprocessing module to parallelize your training on multiple CPU cores, but this line only counts how many cores you have available to size the number of workers.

The gensim Doc2vec model contains your word vector embeddings as well as document vectors for each document in your corpus.

The simple_preprocess utility from gensim is a crude tokenizer that will ignore one-letter words and all punctuation. Any of the tokenizers from chapter 2 will work fine.

You need to provide an object that can iterate through your document strings one at a time.

MEAP reader 24231 (https://forums.manning.com/user/profile/24231.page) suggests that you preallocate a numpy array rather than a bulky python list. You may also want to stream your corpus to and from disk or a database if it will not fit in RAM.

```
>>> import multiprocessing
>>> num_cores = multiprocessing.cpu_count()

>>> from gensim.models.doc2vec import TaggedDocument,\
...     Doc2Vec
>>> from gensim.utils import simple_preprocess

>>> corpus = ['This is the first document ...',\
...           'another document ...']
>>> training_corpus = []
>>> for i, text in enumerate(corpus):
...     tagged_doc = TaggedDocument(\
...         simple_preprocess(text), [i])
...     training_corpus.append(tagged_doc)

>>> model = Doc2Vec(size=100, min_count=2,
...     workers=num_cores, iter=10)
>>> model.build_vocab(training_corpus)
>>> model.train(training_corpus, total_examples=model.corpus_count,
...     epochs=model.iter)
```

gensim provides a data structure to annotate documents with string or integer tags for category labels, keywords, or whatever information you want to associate with your documents.

Kick off the training for 10 epochs.

Before the model can be trained, you need to compile the vocabulary.

Instantiate the Doc2vec object with your window size of 10 words and 100-D word and document vectors (much smaller than the 300-D Google News Word2vec vectors). min_count is the minimum document frequency for your vocabulary.

TIP If you're running low on RAM, and you know the number of documents ahead of time (your corpus object isn't an iterator or generator), you might want to use a preallocated numpy array instead of Python list for your `training_corpus`:

```
training_corpus = np.empty(len(corpus), dtype=object);
    … training_corpus[i] = …
```

Once the Doc2vec model is trained, you can infer document vectors for new, unseen documents by calling `infer_vector` on the instantiated and trained model:

```
>>> model.infer_vector(simple_preprocess(\
...     'This is a completely unseen document'), steps=10)
```

Doc2vec requires a "training" step when inferring new vectors. In your example, you update the trained vector through 10 steps (or iterations).

With these few steps, you can quickly train an entire corpus of documents and find similar documents. You could do that by generating a vector for every document in your corpus and then calculating the cosine distance between each document vector. Another common task is to cluster the document vectors of a corpus with something like k-means to create a document classifier.

Summary

- You've learned how word vectors and vector-oriented reasoning can solve some surprisingly subtle problems like analogy questions and nonsynonomy relationships between words.
- You can now train Word2vec and other word vector embeddings on the words you use in your applications so that your NLP pipeline isn't "polluted" by the GoogleNews meaning of words inherent in most Word2vec pretrained models.
- You used gensim to explore, visualize, and even build your own word vector vocabularies.
- A PCA projection of geographic word vectors like US city names can reveal the cultural closeness of places that are geographically far apart.
- If you respect sentence boundaries with your *n*-grams and are efficient at setting up word pairs for training, you can greatly improve the accuracy of your latent semantic analysis word embeddings (see chapter 4).

Chapter 5 from *Real-World Natural Language Processing* by Masato Hagiwara

Sequential labeling, a framework in which the NLP system assigns a label to each word in an input sentence, is essential for many NLP tasks. For example, part-of-speech tagging, one application of sequential labeling, tags each word as a part of speech, which is helpful in linguistic analysis, an important task in many NLP systems. Likewise, named entity recognition (NER), which identifies proper nouns including personal names, organizations, locations, and geopolitical entities, is vital to extract information that will be of particular interest to the application's user. This chapter dives deep into these and other applications of sequential labeling and also discusses language modeling, an important NLP task in which the system learns the probability of word occurrence based on previous examples. Applications of language modeling in the real world include speech recognition, machine translation, image captioning, text summarization, and more.

Sequential labeling and language modeling

This chapter covers

- Solving part-of-speech (POS) tagging and named entity recognition (NER) using sequential labeling
- Making RNNs more powerful—multi-layer and bidirectional recurrent neural networks (RNNs)
- Capturing statistical properties of language using language models
- Using language models to evaluate and generate natural language text

In this chapter, we're going to discuss sequential labeling—an important NLP framework where systems tag individual words with corresponding labels. Many NLP applications, such as part-of-speech tagging and named entity recognition, can be framed as sequential labeling tasks. In the second half of the chapter, I'll introduce the concept of language models, one of the most fundamental yet excit-

ing topics in NLP. I'll talk about why they're important, and how to use them to evaluate and even generate natural language text.

5.1 Introduction to sequential labeling

In the previous section, we discussed sentence classification, where the task is to assign a label to a given sentence. Spam filtering, sentiment analysis, and language detection are several concrete examples of sentence classification. Although many real-world NLP problems can be formulated as a sentence classification task, it can also be quite limited, because the model, by definition, only allows to assign a single label to the whole sentence. But what if we wanted something more granular? For example, what if we wanted to do something with individual words, not only with the sentence? The most typical scenario you encounter is when you want to extract something from the sentence that cannot be easily solved by sentence classification. This is where sequential labeling comes into play.

5.1.1 What is sequential labeling

Sequential labeling is an NLP task where, given a sequence such as a sentence, the NLP system assigns a label to each element (for example, word) of the input sequence. This contrasts with sentence classification, where a label is assigned only to the input *sentence*. Figure 5.1 illustrates this contrast.

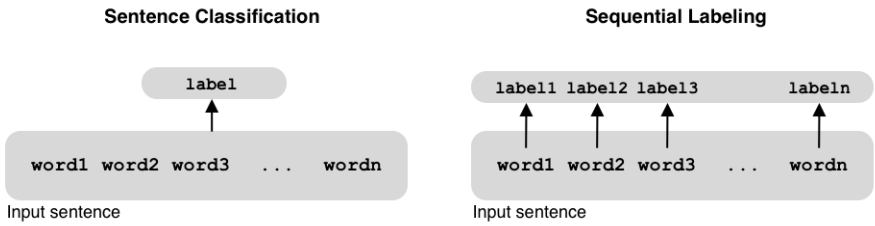


Figure 5.1 Sentence classification vs sequential labeling.

But why is this even a good idea? When do we need a label per word? A typical scenario where sequential labeling comes in handy is when you want to analyze a sentence and produce some linguistic information per word. For example, part-of-speech (POS) tagging, which I mentioned in chapter 1, produces a POS tag such as nouns, verbs, prepositions, and so on, for each word in the input sentence and is a perfect match for sequential labeling. See figure 5.2 for the illustration.

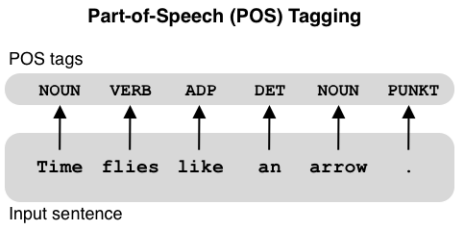


Figure 5.2 Part-of-speech (POS) tagging using sequential labeling.

POS tagging is one of the most fundamental, important NLP tasks. Many English words (and words in many other languages as well) are ambiguous, meaning that they have multiple possible interpretations. For example, the word "book" can be used to describe a physical or electronic object consisting of pages ("I read a book"), or an action for reserving something ("I need to book a flight"). Downstream NLP tasks, such as parsing, classification, and so on, benefit greatly by knowing what each appearance of "book" actually means to process the input sentence. If you were to build a speech synthesis system, you must know the POS of certain words to pronounce them correctly—"lead" as a noun (a kind of metal) rhymes with "bed" while "lead" as a verb (to direct, guide) rhymes with "bead." POS tagging is an important first step toward solving this ambiguity.

Another scenario is when you want to extract pieces of information from a sentence. For example, if you want to extract sub-sequences (phrases) such as noun phrases, verb phrases, and so on, this is also a sequential labeling task. How can you achieve extraction using labeling? The idea is to mark the beginning and the end (or the beginning and the continuation, depending on how you represent) of the desired piece of information using labeling. An example of this is *named entity recognition* (NER), which is a task to identify mentions to real-world entities, such as proper nouns and numerical expressions, from a sentence (figure 5.3).

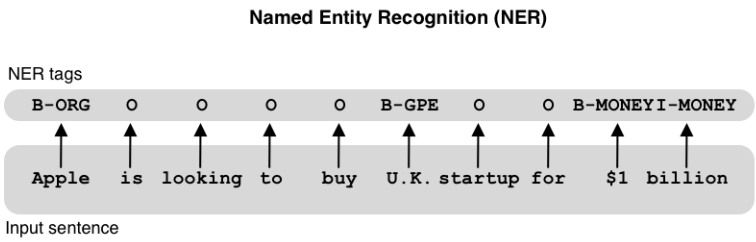


Figure 5.3 Named entity recognition (NER) using sequential labeling.

Notice that all the words that aren't part of any named entities are tagged as O (for "Outside"). For now, you can ignore the cryptic labels in figure 5.3 such as B-GPE and I-MONEY. I'll talk more about how to formulate NER as a sequential labeling problem in section 5.4.

5.1.2 Using RNNs to encode sequences

In sentence classification, we used recurrent neural networks (RNNs) to convert an input of variable length to a fixed-length vector. The fixed-length vector, which gets converted to a set of "scores" by a linear layer, captures the information about the input sentence that's necessary for deriving the sentence label. As a reminder, what this RNN does can be represented by the following pseudo-code and the diagram in figure 5.4.

```
def rnn_vec(words):
    state = init_state()
    for word in words:
        state = update(state, word)
return state
```

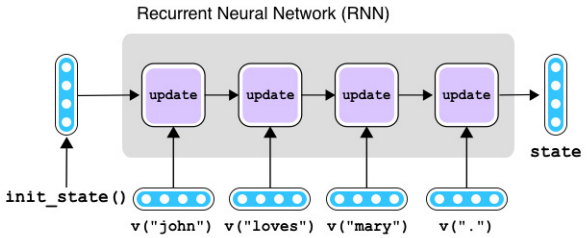


Figure 5.4 Recurrent neural network (RNN) for sentence classification.

Then, what kind of neural network could be used for sequential tagging? We seem to need information for every input `word` in the sentence, not only at the end. If you look at the pseudo-code for `rnn_vec()` carefully, you notice that we already have information for every `word` in the input, which is captured by `state`. The function just happens to only return the final value of `state`, but there's no reason we can't store intermediate values of `state` and return them as a list instead, as in the following function.

```
def rnn_seq(words):
    state = init_state()
    states = []
    for word in words:
        state = update(state, word)
        states.append(state)
    return states
```

If you apply this function to the previous "time flies" example and unroll it, that is, write it without using a loop, it will look like:

```
state = init_state()
states = []
state = update(state, v("time"))
states.append(state)
state = update(state, v("flies"))
states.append(state)
state = update(state, v("like"))
states.append(state)
state = update(state, v("an"))
states.append(state)
state = update(state, v("arrow"))
states.append(state)
state = update(state, v("."))
states.append(state)
```

Note that `v()` here is a function that returns the embedding for the given word. This can be visualized as in figure 5.5. Notice that for each input word `word` the network

produces the corresponding `state` that captures some information about `word`. The length of the list `states` is the same as that of `words`. The final value of `states`, that is, `states[-1]`, is identical to the return value of `rnn_vec()`.

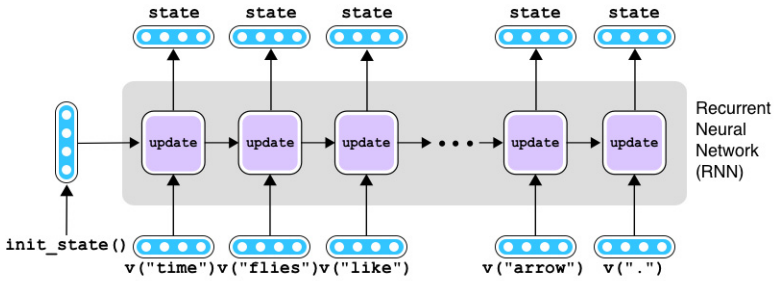


Figure 5.5 Recurrent neural network (RNN) for sequential labeling.

If you think of this RNN as a black box, it takes a sequence of something (for example, word embeddings) and converts it to a sequence of vectors that encode some information about individual words in the input, so this architecture is called Seq2seq (for "sequence-to-sequence") encoder in AllenNLP.

The final step is to apply a linear layer to each state of this RNN to derive a set of scores that correspond to how likely each label is. If this is a part-of-speech tagger, we need one score for the label NOUN, another for VERB, and so on, for each and every word. This conversion is illustrated in figure 5.6. Note that the same linear layer (with the same set of parameters) is applied to every `state`.

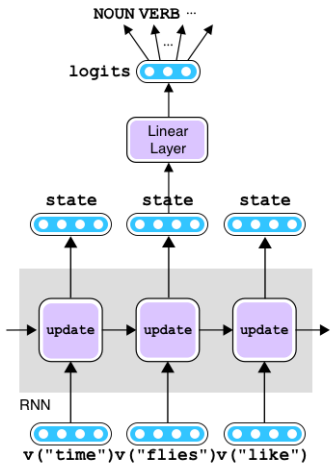


Figure 5.6 Applying linear layer to RNN.

To sum up, we can use almost the same structure for sequential labeling as the one used for sentence classification. The only difference is the former produces hidden

state per each word, not just per sentence. To derive scores used for determining labels, a linear layer is applied to every hidden state.

5.1.3 Implementing Seq2seq encoder in AllenNLP

AllenNLP implements an abstract class called `Seq2SeqEncoder` for abstracting all Seq2seq encoders that take in a sequence of vectors and return another sequence of modified vectors. In theory, you can inherit the class and implement your own Seq2seq encoder. In practice, however, you most likely use one of the off-the-shelf implementations that PyTorch/AllenNLP provide, such as LSTM and GRU. Remember, when we built the encoder for the sentiment analyzer, we used PyTorch's built-in `torch.nn.LSTM` and wrapped it with `PytorchSeq2VecWrapper`, which makes it compatible with AllenNLP's abstraction:

```
encoder = PytorchSeq2VecWrapper(
    torch.nn.LSTM(EMBEDDING_DIM, HIDDEN_DIM, batch_first=True))
```

AllenNLP also implements `PytorchSeq2SeqWrapper`, which takes one of PyTorch's built-in RNN implementations and make it compliant with AllenNLP's `Seq2Seq Encoder`, so there's very little change you need to initialize a Seq2seq encoder:

```
encoder = PytorchSeq2SeqWrapper(
    torch.nn.LSTM(EMBEDDING_DIM, HIDDEN_DIM, batch_first=True))
```

That's it! There are a couple more things to note, but there's surprisingly little change you need to make to the sentence classification code to make it work for sequential labeling. This is thanks to the powerful abstraction of AllenNLP—most of the time you need to worry only about how individual components interact with each other, not about how these components work internally.

5.2 Building a part-of-speech tagger

In this section, we're going to build our first sequential labeling application—a part-of-speech (POS) tagger. You can see the entire code for this section on the Google Colab notebook (http://realworldnlpbook.com/ch5.html#pos-nb).

5.2.1 Reading dataset

As we saw in chapter 1, a *part of speech* (POS) is a category of words that share the similar grammatical properties. Part-of-speech tagging is the process of tagging each word in a sentence with a corresponding part-of-speech tag. A training set for POS tagging follows a tagset, which is a set of pre-defined POS tags for the language.

To train a POS tagger, we need a dataset where every word in every sentence is tagged with its corresponding POS tag. In this experiment, we're going to use the English Universal Dependencies (UD) dataset. Universal Dependencies is a language-independent dependency grammar framework developed by a group of researchers. UD also defines a tagset called the Universal part-of-speech tagset (http://realworldnlpbook.com/ch1.html#universal-pos). The use of UD and the Universal POS tagset

has been very popular in the NLP community, especially for language-independent tasks and models such as POS tagging and parsing.

We're going to use one sub-corpus of UD called A Gold Standard Universal Dependencies Corpus for English, which is built on top of the English Web Treebank (EWT) (http://realworldnlpbook.com/ch5.html#ewt) and can be used under a creative commons license. You can download the entire dataset from the dataset page here (http://realworldnlpbook.com/ch5.html#ewt-data) if needed.

Universal Dependencies datasets are distributed in a format called the CoNLL-U format (http://universaldependencies.org/docs/format.html). AllenNLP already implements a dataset reader called `UniversalDependenciesDatasetReader` that reads datasets in this format and returns a collection of instances that include information like word forms, POS tags, and dependency relationship, so all you need to do is initialize and use it:

```
from allennlp.data.dataset_readers import UniversalDependenciesDatasetReader

reader = UniversalDependenciesDatasetReader()
train_dataset = reader.read(
    'https://s3.amazonaws.com/realworldnlpbook/data/ud-treebanks-v2.3/'
    '/UD_English-EWT/en_ewt-ud-train.conllu')
dev_dataset = reader.read(
    'https://s3.amazonaws.com/realworldnlpbook/data/ud-treebanks-v2.3/'
    '/UD_English-EWT/en_ewt-ud-dev.conllu')
```

Also, don't forget to initialize a `Vocabulary` instance too:

```
from allennlp.data.vocabulary import Vocabulary

vocab = Vocabulary.from_instances(train_dataset + dev_dataset)
```

5.2.2 Defining the model and the loss

The next step for building a POS tagger is to define the model. In the previous section, we already saw that you can initialize a Seq2Seq encoder with little modification using AllenNLP's built-in `PytorchSeq2VecWrapper`. Let's define other components (word embeddings and LSTM) and several variables necessary for the model:

```
import torch
from allennlp.modules.seq2seq_encoders import PytorchSeq2SeqWrapper
from allennlp.modules.text_field_embedders import TextFieldEmbedder,
     BasicTextFieldEmbedder
from allennlp.modules.token_embedders import Embedding

EMBEDDING_SIZE = 128
HIDDEN_SIZE = 128

token_embedding = Embedding(num_embeddings=vocab.get_vocab_size('tokens'),
                            embedding_dim=EMBEDDING_SIZE)
word_embeddings = BasicTextFieldEmbedder({"tokens": token_embedding})

lstm = PytorchSeq2SeqWrapper(
torch.nn.LSTM(EMBEDDING_SIZE, HIDDEN_SIZE, batch_first=True))
```

Now we're ready to define the body of the POS tagger model, as shown in listing 5.1.

Listing 5.1 POS tagger model

```
from allennlp.models import Model
from allennlp.modules.seq2seq_encoders import Seq2SeqEncoder
from allennlp.modules.text_field_embedders import TextFieldEmbedder
from allennlp.nn.util import get_text_field_mask,
     sequence_cross_entropy_with_logits
from allennlp.training.metrics import CategoricalAccuracy

class LstmTagger(Model):
    def __init__(self,
                 embedder: TextFieldEmbedder,
                 encoder: Seq2SeqEncoder,
                 vocab: Vocabulary) -> None:
        super().__init__(vocab)
        self.embedder = embedder
        self.encoder = encoder

        self.linear = torch.nn.Linear(
            in_features=encoder.get_output_dim(),
            out_features=vocab.get_vocab_size('pos'))

        self.accuracy = CategoricalAccuracy()

    def forward(self,
                words: Dict[str, torch.Tensor],
                pos_tags: torch.Tensor = None,
                **args) -> Dict[str, torch.Tensor]:
        mask = get_text_field_mask(words)

        embeddings = self.embedder(words)
        encoder_out = self.encoder(embeddings, mask)
        tag_logits = self.linear(encoder_out)

        output = {"tag_logits": tag_logits}
        if pos_tags is not None:
            self.accuracy(tag_logits, pos_tags, mask)
            output["loss"] = sequence_cross_entropy_with_logits(
                tag_logits, pos_tags, mask)

        return output

    def get_metrics(self, reset: bool = False) -> Dict[str, float]:
        return {"accuracy": self.accuracy.get_metric(reset)}
```

We use accuracy to evaluate the POS tagger.

We need **args to capture unnecessary instance fields that AllenNLP automatically destructures.

Seq2Seq encoder is trained using a sequence cross entropy loss.

Notice that the code shown in listing 5.1 is similar to the code for `LstmClassifier` (listing 4.1), which we used for building a sentiment analyzer. In fact, except for several naming differences, there's only one fundamental difference—the type of loss function.

Recall that we used a loss function called cross entropy for sentence classification tasks, which basically measures how far apart two distributions are. If the model pro-

duces a high probability for the true label, the loss will be low. Otherwise it will be high. But this assumes that there's only one label per sentence. How can we measure how far the prediction is from the true label when there's one label per word?

The answer is: still use the cross entropy but average it over all the elements in the input sequence. For POS tagging, you compute the cross entropy per word as if it were an individual classification task, sum it over all the words in the input sentence, and divide by the length of the sentence. This will give you a number reflecting how well your model is predicting the POS tags for the input sentence on average. See figure 5.7 for the illustration.

As for the evaluation metric, POS taggers are usually evaluated using accuracy, which we're going to use here. Average human performance on POS tagging is around 97%, while the state-of-the-art POS taggers slightly outperform this (http://realworldnlpbook.com/ch5.html#pos-sota). You need to note that accuracy isn't without a problem, however—assume there's a relatively rare POS tag (for example, `SCONJ`, which means subordinating conjugation) that accounts for only 2% of total tokens, and a POS tagger messes it up every time it appears. If the tagger gets the rest of the tokens all correct, it still achieves 98% accuracy.

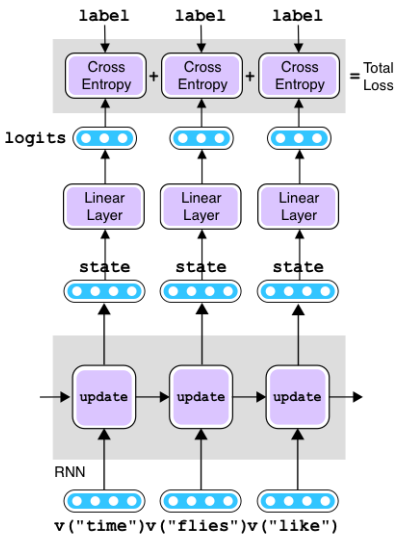


Figure 5.7 Computing loss for sequence.

5.2.3 Building the training pipeline

Now we're ready to move on to building the training pipeline. As with the previous tasks, training pipelines in AllenNLP look similar to each other. See listing 5.2 for the training code.

Listing 5.2 Training pipeline for POS tagger

```
import torch.optim as optim
from allennlp.data.iterators import BucketIterator
from allennlp.training.trainer import Trainer

model = LstmTagger(word_embeddings, encoder, vocab)

optimizer = optim.Adam(model.parameters())

iterator = BucketIterator(batch_size=16,
                          sorting_keys=[("words", "num_tokens")])

iterator.index_with(vocab)

trainer = Trainer(model=model,
                  optimizer=optimizer,
                  iterator=iterator,
                  train_dataset=train_dataset,
                  validation_dataset=dev_dataset,
                  patience=10,
                  num_epochs=10)
trainer.train()
```

When you run this, AllenNLP will alternate between two phases: 1) training the model using the train set, and 2) evaluating it using the validation set for each epoch, while monitoring the loss and accuracy on both sets. Validation set accuracy plateaus around 88% after several epochs. After the training is over, you can run the model for an unseen instance as shown here:

```
import numpy as np
from realworldnlp.predictors import UniversalPOSPredictor

predictor = UniversalPOSPredictor(model, reader)
tokens = ['The', 'dog', 'ate', 'the', 'apple', '.']
logits = predictor.predict(tokens)['tag_logits']
tag_ids = np.argmax(logits, axis=-1)

print([vocab.get_token_from_index(tag_id, 'pos') for tag_id in tag_ids])
```

This code uses `UniversalPOSPredictor`, a predictor that I wrote for this particular POS tagger. Although its detail isn't important, you can look at its code if you're interested (http://realworldnlpbook.com/ch5#upos-predictor). If successful, this will show a list of POS tags: `[DET, NOUN, VERB, DET, NOUN, PUNCT]` which is indeed a correct POS tag sequence for the input sentence.

5.3 Multi-layer and bidirectional RNNs

As we've seen so far, RNNs are a powerful tool for building NLP applications. In this section, I'm going to talk about their structural variants—multi-layer and bidirectional RNNs—which are even more powerful components for building highly accurate NLP applications.

5.3.1 Multi-layer RNNs

If you look at an RNN as a black box, it's a neural network structure that converts a sequence of vectors (word embeddings) to another sequence of vectors (hidden states). The input and output sequences are of the same length, usually the number of input tokens. This means that you can repeat this "encoding" process multiple times by stacking RNNs on top of each other. The output (hidden states) of one RNN becomes the input of another RNN that's just above the previous one. A substructure (such as a single RNN) of a bigger neural network is called a *layer*, because you can stack them together like layers. The structure of a two-layered RNN is shown in figure 5.8.

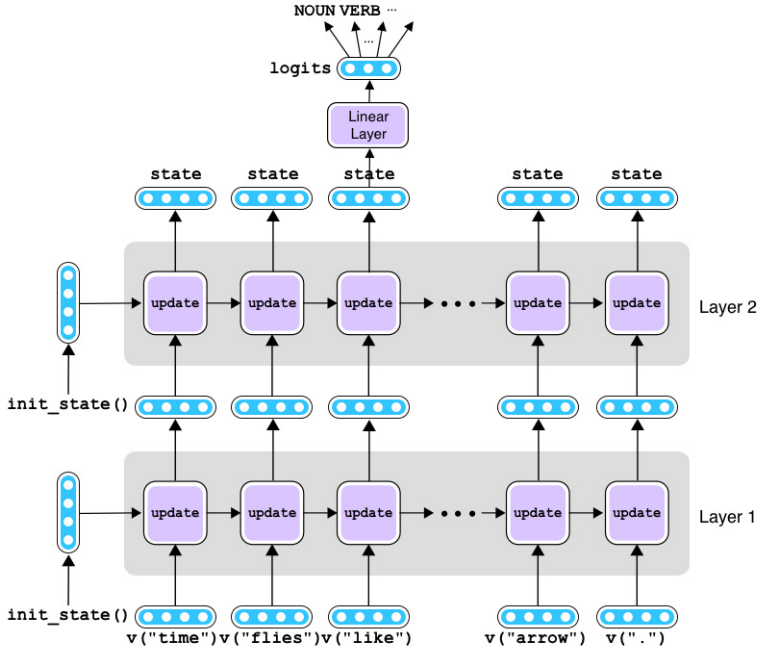


Figure 5.8 Two-layered RNN.

Why is this a good idea? If you think of a layer of RNN as a machine that takes in something concrete (for example, word embeddings) and extracts abstract concepts (for example, scores for POS tags), you can expect that, by repeating this process, RNNs can extract increasingly more abstract concepts as the number of layers increases. Although not fully theoretically proven, many real-world NLP applications use multi-layer RNNs. For example, Google's Neural Machine Translation (NMT) system uses a stacked RNN consisting of eight layers for both the encoder and the decoder (http://realworldnlpbook.com/ch5.html#nmt-paper).

To use multi-layer RNNs in your NLP application, all you need to do is change how the encoder is initialized. Specifically, you only need to specify the number of layers

using the `num_layers` parameter, and AllenNLP makes sure that the rest of the training pipeline works as is:

```
encoder = PytorchSeq2SeqWrapper(
 torch.nn.LSTM(
     EMBEDDING_SIZE, HIDDEN_SIZE, num_layers=2, batch_first=True))
```

If you change this line and re-run the POS tagger training pipeline, you'll notice that accuracy on the validation set is almost unchanged or slightly lower than the previous model with a single-layer RNN. This isn't surprising—information required for POS tagging is mostly superficial, such as the identity of the word being tagged and neighboring words. Rarely does it require deep understanding of the input sentence. On the other hand, adding layers to an RNN isn't without additional cost. It slows down the training and inference and increases the number of parameters, which makes it susceptible to overfitting. For this small experiment, adding layers to the RNN seems to do more harm than good. When you change the structure of the network, always remember to verify its effect on a validation set.

5.3.2 Bidirectional RNN

Up to now, we've been feeding words to RNNs as they come in—from the beginning of the sentence toward the end. This means that when an RNN is processing a word, it can only leverage the information it has encountered so far, which is the word's left context. True, you can get quite a bit of information from a word's left context. For example, if a word is preceded by a modal verb (for example, "can"), it's a strong signal that the next word is a verb. However, there's a lot of information in the right context as well. For example, if you know that the next word is a determiner (for example, "a"), it's a strong signal that "book" on its left is a verb, not a noun.

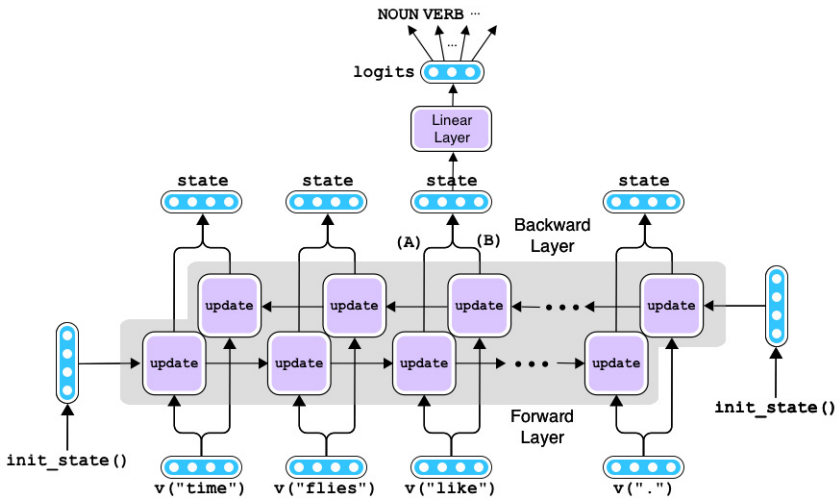


Figure 5.9 Bidirectional RNN.

Bidirectional RNNs (or simply biRNNs) solve this problem by combining two RNNs with opposite directions. A forward RNN is a forward-facing RNN that we've used so far in this book—it scans the input sentence left-to-right and uses the input word and all the information on its left to update the state. A backward RNN, on the other hand, scans the input sentence right-to-left. It uses the input word and all the information on its right to update the state. This is equivalent to flipping the order of the input sentence and feeding it to a forward RNN. The final hidden states produced by biRNNs are concatenations of hidden states from the forward and backward RNNs. See figure 5.9 for an illustration.

Let's use a concrete example to illustrate this. Assume the input sentence is "time flies like an arrow" and you'd like to know the POS tag for the word "like" in the middle of this sentence. The forward RNN processes "time" and "flies", and by the time it reaches "like," its internal state ("A" in figure 5.9) encodes all the information about "time flies like". Similarly, the backward RNN processes "arrow" and "an," and by the time it reaches "like," the internal state ("B" in figure 5.9) has encoded all the information about "like an arrow." The internal state from the biRNN for "like" is the concatenation of these two states (A+B). You literally stitch together two vectors, no mathematical operations involved. As a result, the internal state for "like" encodes all the information from the entire sentence. This is a great improvement over knowing only half the sentence!

Implementing a biRNN is similarly easy—you just need to add the `bidirectional=True` flag when initializing the RNN:

```
encoder = PytorchSeq2SeqWrapper(
torch.nn.LSTM(
        EMBEDDING_SIZE, HIDDEN_SIZE, bidirectional=True, batch_first=True))
```

If you train the POS tagger with this change, the validation set accuracy will jump from ~88% to 91%. This implies that incorporating the information on both sides of the word is effective for POS tagging.

Note that you can combine the two techniques introduced in this section by stacking bidirectional RNNs. The output from one layer of biRNN (concatenation of a forward and a backward layer) becomes the input to another layer of biRNN (see figure 5.10). You can implement this by specifying both flags—`num_layers` and `bidirectional`—when initializing the RNN in PyTorch/AllenNLP.

5.4 Named entity recognition

Sequential labeling can be applied to many information-extraction tasks, not only to part-of-speech tagging. In this section, I'll introduce the task of named entity recognition (NER) and demonstrate how to build an NER tagger using sequential labeling. The code for this section can be viewed and executed via the Google Colab platform (http://realworldnlpbook.com/ch5#ner-nb).

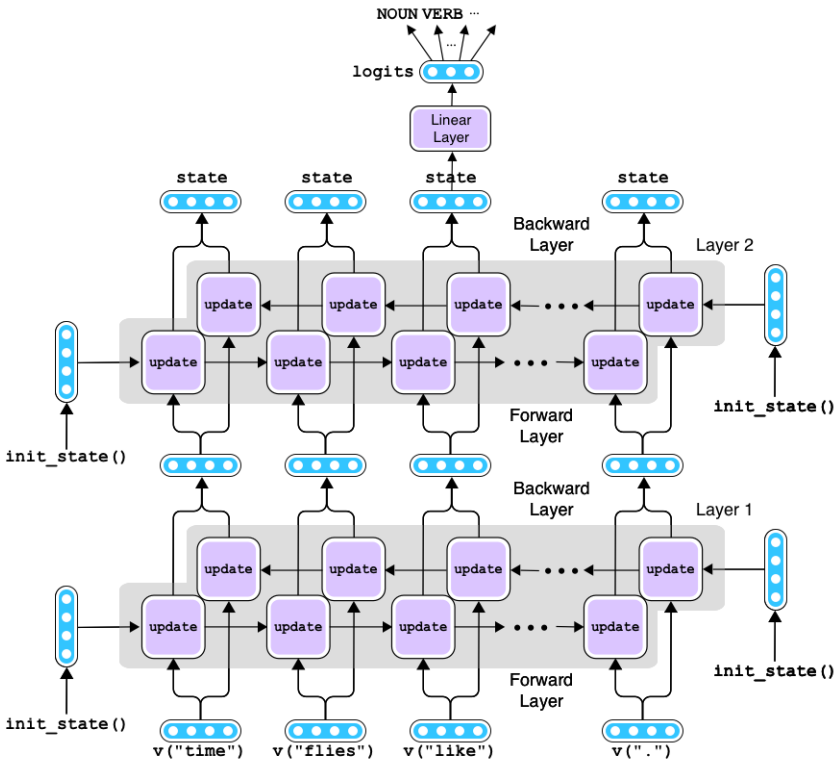


Figure 5.10 Two-layered bidirectional RNN.

5.4.1 What is named entity recognition?

As mentioned before, named entities are mentions of real-world entities such as proper nouns. Common named entities that are usually covered by NER systems include the following:

- Personal name (PER) . . . Alan Turing, Lady Gaga, Elon Musk, . . .
- Organization (ORG) . . . Google, United Nations, Giants, . . .
- Location (LOC) . . . Mount Rainer, Bali Island, Nile, . . .
- Geo-political entity (GPE) . . . UK, San Francisco, Southeast Asia, . . .

However, different NER systems deal with different sets of named entities. The concept of named entities is a bit overloaded in NLP to mean any mentions that are of interest to the application's user. For example, in the medical domain, you may want to extract mentions to names of drugs and chemical compounds. In the financial domain, companies, products, and stock symbols may be of interest. In many domains, numerical and temporal expressions are also considered.

Identifying named entities is in itself important, because named entities (who, what, where, when, and so on) are often what most people are interested in. But NER

is also an important first step for many other NLP applications. One such task is *relation extraction*: extracting all relations between named entities from the given document. For example, given a press release document, you may want to extract the event described in the release, for example, which company has bought which other company for what price. This often assumes that all the parties are already identified via NER. Another task that's closely related to NER is *entity linking*, where mentions of named entities are linked to some knowledge base, such as Wikipedia. When Wikipedia is used as a knowledge base, entity linking is also called Wikification.

But you may be wondering, what's so difficult about simply extracting named entities? If they're just proper nouns, can you simply compile a dictionary of, say, all the celebrities (or all the countries, or whatever you are interested in) and use it? The idea is, whenever the system encounters a noun, it would run the name through this dictionary and tag the mention if it appears in it. Such dictionaries are called *gazetteers*, and many NER systems do use them as a component.

However, relying solely on such dictionaries has one major issue—*ambiguity*. Earlier we saw that a single word type could have multiple parts-of-speech (for example, "book" as a noun and a verb), and named entities are no exception. For example, "Georgia" can be the name of a country, a US state, towns and communities across the US (Georgia, Indiana, Georgia, Nebraska), a film, a number of songs, ships, and a personal name. Simple words such as "book" could also be named entities, including: Book (a community in Louisiana), Book/Books (a surname), The Books (an American band), and so on. Simply matching mentions against dictionaries would tell you nothing about their identities if they're ambiguous.

Fortunately, there are often clues in the sentence that can be used to *disambiguate* the mentions. For example, if the sentence reads "I live in Georgia," it's usually a strong signal that "Georgia" is a name of a place, not a film or a person's name. NER systems use a combination of signals about the mentions themselves (for example, whether they're in a predefined dictionary) and about their context (whether they're preceded or followed by certain words) to determine their tags.

5.4.2 Tagging spans

Unlike POS tagging, where each word is assigned a POS tag, mentions to named entities can span over more than one word, for example, "the United States" and "World Trade Organization." A *span* in NLP is simply a range over one or more contagious words. How can we use the same sequential tagging framework to model spans?

A common practice in NLP is to use a form of encoding to convert spans into per-word tags. The most common encoding scheme used in NER is called IOB2 tagging. It represents spans by a combination of the positional tag and the category tag. There are three types of positional tags:

- B (Beginning) . . . assigned to the first (or the only) token of a span
- I (Inside) . . . assigned to all but the first token of a span
- O (Outside) . . . assigned to all words outside of any spans

Now, let's take a look at the NER example we saw earlier (figure 5.11). The token "Apple" is the first (and the only) token of `ORG` (for "organization") and it is assigned a `B-ORG` tag. Similarly, "UK", the first and the only token of GPE (for "geo-political entity"), is assigned `B-GPE`. For "$1" and "billion", the first and the second tokens of a monetary expression (MONEY), `B-MONEY` and `I-MONEY` are assigned, respectively. All the other tokens are given `O`.

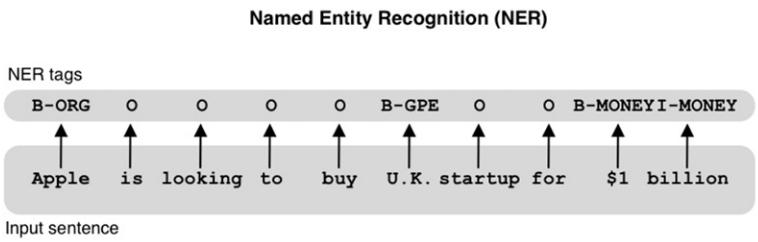


Figure 5.11 Named entity recognition (NER) using sequential labeling.

The rest of the pipeline for solving NER is similar to that of part-of-speech tagging, because both are concerned with assigning an appropriate tag for each word and can be solved by RNNs. In the next section, we're going to build a simple NER system using neural networks.

5.4.3 Implementing a named entity recognizer

In order to build an NER system, we use the Annotated Corpus for Named Entity Recognition prepared by Abhinav Walia published on Kaggle (http://realworldnlpbook.com/ch5.html#ner-data). In what follows, I'm going to assume that you downloaded and expanded the dataset under `data/entity-annotated-corpus`. Alternatively, you can use the copy of the dataset I uploaded to S3 (http://realworldnlpbook.com/ch5.html#ner-data-s3), which is what the following code does.

I went ahead and wrote a dataset reader for this dataset (http://realworldnlpbook.com/ch5.html#ner-reader), so you can simply import (or copy and paste) it to use it:

```
from realworldnlp.examples.ner.ner import NERDatasetReader

reader = NERDatasetReader()
dataset = list(reader.read('https://s3.amazonaws.com/realworldnlpbook/'
                           'data/entity-annotated-corpus/ner_dataset.csv'))
```

Because the dataset isn't separated into train, validation, and test sets, we're going to separate them into train and validation sets based on the index of instances:

```
train_dataset = [inst for i, inst in enumerate(dataset) if i % 10 != 0]
dev_dataset = [inst for i, inst in enumerate(dataset) if i % 10 == 0]
```

Here we're using every tenth instances for the validation (dev) set, and everything else for the train set. Splitting a dataset based on the modulo of the line number (or some ID) is a common practice in NLP and machine learning.

The RNN-based sequential tagging model and the rest of the training pipeline look almost the same as the previous example (POS tagger). The only difference is how we evaluate our NER model. Because most of the tags for a typical NER dataset are simply "O," using tag accuracy is misleading—a stupid system that tags everything "O" would achieve high accuracy. Instead, NER is usually evaluated as an information extraction task, where the goal is to extract named entities from texts, not only to tag them. We'd like to evaluate NER systems based on the "cleanness" of retrieved named entities (how many of them are actual entities) and their "completeness" (how many of actual entities the system was able to retrieve). Does this sound any familiar to you? Yes, these are the definition of recall and precision we talked about in section 4.3. Because there are usually multiple types of named entities, these metrics (precision, recall, and F1 measure) are computed per entity type.

> **NOTE** If these metrics are computed while ignoring entity types, it's called a *micro average*. For example, the micro-averaged precision is the total number of true positives of all types divided by the total number of retrieved named entities regardless of the type. On the other hand, if these metrics are computed per entity type and then get averaged, it's called a *macro average*. For example, if the precision for PER and GPE is 80% and 90%, respectively, its macro average is 85%. What AllenNLP computes in the following is the micro average.

AllenNLP implements `SpanBasedF1Measure`, which computes per-type metrics (precision, recall, and F1 measure) as well as the average. You can define the metric in `__init__()` of your model:

```
self.f1 = SpanBasedF1Measure(vocab, tag_namespace='labels')
```

And use it to get metrics during training and validation:

```
def get_metrics(self, reset: bool = False) -> Dict[str, float]:
    f1_metrics = self.f1.get_metric(reset)
    return {'accuracy': self.accuracy.get_metric(reset),
            'prec': f1_metrics['precision-overall'],
            'rec': f1_metrics['recall-overall'],
            'f1': f1_metrics['f1-measure-overall']}
```

If you run this training pipeline, you get an accuracy around 0.97, and precision, recall, F1 will all hover around 0.83. You can also use the `predict()` method to obtain named entity tags for an unseen sentence:

```
tokens = ['Apple', 'is', 'looking', 'to', 'buy', 'U.K.', 'startup',
          'for', '$1', 'billion', '.']
labels = predict(tokens, model)
print(' '.join('{}/{}'.format(token, label)
               for token, label in zip(tokens, labels)))
```

which produces:

```
Apple/B-org is/O looking/O to/O buy/O U.K./O startup/O for/O $1/O billion/O ./O
```

This isn't perfect—the NER tagger got the first named entity ("Apple") correct but missed two others ("UK" and "$1 billion"). If you look at the training data, the mention "UK" never appears, and no monetary values are tagged. It isn't surprising that the system is struggling to tag entities that it has never seen before. In NLP (and also machine learning in general), the characteristic of the test instances needs to match that of the train data for the model to be fully effective.

5.5 Modeling a language

In this section, I'll switch gears a little bit and introduce *language models*, which is one of the most important concepts in NLP. We'll discuss what they are, why they're important, and how to train them using the neural network components we've introduced so far.

5.5.1 What a language model is

Imagine you're asked to predict what word comes next given a partial sentence: "My trip to the beach was ruined by bad ___." What words could come next? Many things could ruin a trip to a beach, but most likely it's bad weather. Maybe it's bad mannered people at the beach, maybe it's bad food that the person had eaten before the trip, but most would agree that "weather" is a likely word that comes after this partial sentence. Few other nouns (*people*, *food*, *dogs*, . . .) and words of other parts-of-speech (*be*, *the*, *run*, *green*, . . .) are as appropriate as "weather" in this context.

What you did is to assign some belief (or probability) to an English sentence. You compared several alternatives and judged how likely they are as English sentences. Most people would agree that the probability of "My trip to the beach was ruined by bad weather" is a lot higher than "My trip to the beach was ruined by bad dogs."

Formerly, a *language model* is a statistical model that gives a probability to a piece of text. An English language model would assign higher probabilities to sentences that look like English. For example, an English language model would give a higher probability to "My trip to the beach was ruined by bad weather" than it does to "My trip to the beach was ruined by bad dogs" or even "by weather was trip my bad beach the ruined to." The more grammatical and the more "sense" the sentence makes, the higher the probability is.

5.5.2 Why language models are useful

You may wonder what use such a statistical model has. Although predicting the next word might come in handy when you're answering fill-in-the-blank questions for an exam, what particular roles do language models play in NLP?

The answer is it's essential for any systems that generate natural language. For example, machine translation systems, which generate a sentence in a language given a sentence in another language, would benefit greatly from high-quality language

models. Why? Let's say we'd like to translate a Spanish sentence "Está lloviendo fuerte" into English ("It is raining hard"). The last word "fuerte" has several English equivalents—*strong*, *sharp*, *loud*, *heavy*, and so on. How would you determine which English equivalent is the most appropriate in this context? There could be many approaches to solve this problem, but one of the simplest is to use an English language model and re-rank several different translation candidates. Assuming you've finished translating up to "It is raining," you would simply replace the word "fuerte" with all the equivalents you can find in a Spanish-English dictionary, which generates "It is raining strong," "It is raining sharp," "It is raining loud," "It is raining hard." Then all you need to do is ask the language model which one of these candidates has the highest probability.

> **NOTE** In fact, neural machine translation models can be thought of as a variation of a language model which generates sentences in the target language conditioned on its input (sentences in the source language). Such a language model is a called *conditional language model* as opposed to an *unconditional language model*, which we discuss here. We'll discuss machine translation models in chapter 6.

A similar situation arises in speech recognition too, which is another task that generates text given spoken audio input. For example, if somebody uttered "You're right," how would a speech recognition system know it's actually "you're right"? Because "you're" and "your" can have the same pronunciation, and so can "right" and "write" and even "Wright" and "rite," the system output could be any one of "You're write," "You're Wright," "You're rite," "Your right," "Your write," "Your Wright," and so on. Again, the simplest approach to resolving this ambiguity is to use a language model. An English language model would properly re-rank these candidates and determine "you're right" is the most likely transcription.

In fact, humans do this type of disambiguation all the time, though unconsciously. When you're having a conversation with somebody else at a large party, the actual audio signal you receive is often very noisy. Most people can still understand each other without any issues because peoples' language models help them "correct" what you hear and interpolate any missing parts. You'll notice this most if you try to converse in a less proficient, second language—you'd have a lot harder time understanding the other person in a noisy environment, because your language model isn't as good as your first language's.

5.5.3 Training an RNN language model

At this point, you may wonder what the connection is between predicting the next word and assigning a probability to a sentence. These two are actually equivalent. Instead of explaining the theory behind it, which requires you to understand some math (especially probability theory), I'll attempt an intuitive example below without going into mathematical details.

Imagine you want to estimate the chance of tomorrow's weather being rainy and the ground is wet. Let's simplify this and assume there are only two types of weather, sunny and rainy. There are only two outcomes for the ground: dry or wet. This is equivalent to estimating the probably of a sequence: [rain, wet].

Further assume that there's a fifty-fifty chance of rain on a given day. After raining, the ground is wet with a 90% chance. Then, what is the probability of the rain and the ground being wet? It's simply 50% times 90%, which is 45%, or 0.45. If we know the probability of one event happening after another, you can simply multiply two probabilities to get the total probability for the sequence. This is called the *chain rule* in probability theory.

Similarly, if you can correctly estimate the probability of one word occurring after a partial sentence, you can simply multiply it with the probability of the partial sentence. Starting from the first word, you can keep doing this until you reach the end of the sentence. For example, if you'd like to compute the probability for "The trip to the beach was . . .," you can multiply:

- Probability of "The" occurring at the beginning of a sentence
- Probability of "trip" occurring after "The"
- Probability of "to" occurring after "The trip"
- Probability of "the" occurring after "The trip to"
- . . .

To build a language model, you need a model that predicts the probability (or more precisely, the probability distribution) of the next word given the context. You may have noticed that this sounds a little familiar. Indeed, what's done here is similar to the sequential labeling models that we've been talking about in this chapter. For example, a POS tagging model predicts the probability distribution over the possible POS tags given the context. A named entity recognition (NER) model does it for the possible named entity tags. The difference is that a language model does it for the possible next words, given what the model has encountered so far. Hopefully, it's starting to make some sense why I talk about language models in this chapter!

In summary, in order to build a language model, you tweak an RNN-based sequence labeling model a little bit so that it gives the estimates for the next word, instead of POS or NER tags. In chapter 3, I talked about the Skip-gram model, which predicts the words in a context given the target word. Notice the similarity here—both models predict the probability over possible words. The input to the Skip-gram model is a single word, while the input to the language model is the partial sequence. You can use a similar mechanism for converting one vector to another using a linear layer, then converting it to a probability distribution using softmax, as we discussed in chapter 3. The architecture is shown in figure 5.12.

The way RNN-based language models are trained is similar to other sequential labeling models. The loss function we use is the sequential cross-entropy loss, which measures how "off" the predicted words are from actual words. The cross-entropy loss is computed per word and gets averaged over all words in the sentence.

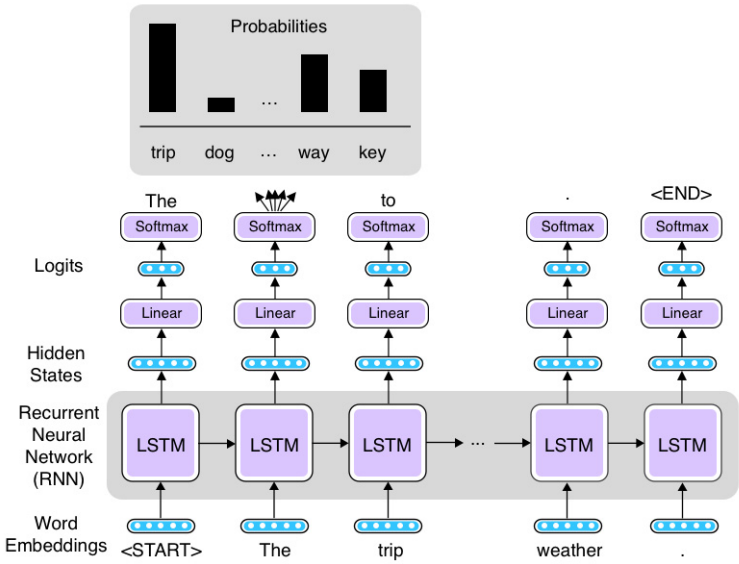


Figure 5.12 Architecture of RNN-based language model.

5.6 Text generation using RNNs

We saw that language models give probabilities to natural language sentences. But the more fun part is that you can generate natural language sentences from scratch using a language model! In the final section of this chapter, we're going to actually build a language model. You can use the trained model to evaluate and generate English sentences. You can find the entire script for this subsection on a Google Colab notebook (http://realworldnlpbook.com/ch5.html#lm-nb).

5.6.1 Feeding characters to an RNN

In the first half of this section, we're going to build an English language model and train it using a generic English corpus. Before we start, we note that the RNN language model we build in this chapter operates on *characters*, not on words or tokens. All the RNN models we've seen so far operate on words, which means the input to the RNN was always sequences of words. On the other hand, the RNN we're going to use in this section takes sequences of characters as the input.

In theory, RNNs can operate on sequences of anything, be it tokens or characters or something completely different (for example, waveform for speech recognition), as long as they're something that can be turned into vectors. In building language models, we often feed characters, even including whitespace and punctuations as the input, treating them words of length one. The rest of the model works exactly the same—individual characters are first embedded (converted to vectors) and then fed into the RNN, which is in turn trained so that it can best predict the distribution over the characters that are likely to come next.

There are a couple of considerations when you're deciding whether you should feed words or characters to an RNN. Using characters will definitely make the RNN less efficient, meaning that it would need more computation to "figure out" the same concept. For example, a word-based RNN can receive the word "dog" at a timestep and update its internal states, while a character-based RNN couldn't do it until it receives three elements *d*, *o*, and *g*, and probably "_" (whitespace). A character-based RNN needs to "learn" that a sequence of these three characters mean something special (the concept of "dog").

On the other hand, by feeding characters to RNNs, you can bypass many issues arising from dealing with tokens. One such issue is related to out of vocabulary (or OOV) words. When training a word-based RNN, you usually fix the entire set of vocabulary, often by enumerating all words that appeared in the train set. But whenever it encounters an OOV word in the test set, it doesn't know what to do with it. Oftentimes, it assigns a special token `<UNK>` to all OOV words and treat them in the same way, which is not ideal. A character-based RNN, on the contrary, can still operate on individual characters, so it may be able to figure out what "doggy" means, for example, based on the rules it has learned by observing "dog" in the train set, even though it has never seen the exact word "doggy" before.

5.6.2 Evaluating text using language model

Let's start building a character-based language model. The first step is to read a plain text dataset file and generate instances for training the model. I'm going to show how to construct an instance without using a dataset reader for a demonstration purpose. Suppose you have a Python string object `text` that you'd like to turn into an instance for training a language model. First you need to segment it into characters using `CharacterTokenizer` as follows:

```
from allennlp.data.tokenizers import CharacterTokenizer

tokenizer = CharacterTokenizer()
tokens = tokenizer.tokenize(text)
```

Note that `tokens` here is a list of `Token` objects, but tokens here contain single characters. Then you insert the `<START>` and `<END>` symbols at the beginning and at the end of the list:

```
from allennlp.common.util import START_SYMBOL, END_SYMBOL

tokens.insert(0, Token(START_SYMBOL))
tokens.append(Token(END_SYMBOL))
```

Inserting special symbols like these at the beginning and the end of each sentence is a common practice in NLP. With these symbols, models can distinguish between occurrences of a token in the middle of a sentence vs at the beginning/end of a sentence. For example, a period is a lot more likely to occur at the end of a sentence (". <END>") than the beginning ("<START> ."), to which a language model can give two very different probabilities, which is impossible to do without the use of these symbols.

Finally, you can construct an instance by specifying individual text fields. Notice that the "output" of a language model is identical to the input, simply shifted by one token:

```
from allennlp.data.fields import TextField
from allennlp.data.instance import Instance

input_field = TextField(tokens[:-1], token_indexers)
output_field = TextField(tokens[1:], token_indexers)
instance = Instance({'input_tokens': input_field,
                     'output_tokens': output_field})
```

Here, `token_indexers` specifies how individual tokens get mapped into IDs. We simply use the `SingleIdTokenIndexer` we've been using so far:

```
from allennlp.data.token_indexers import TokenIndexer

token_indexers = {'tokens': SingleIdTokenIndexer()}
```

Figure 5.13 shows an instance created from this process.

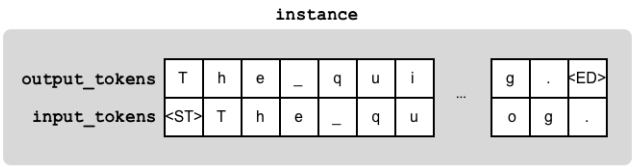


Figure 5.13 Instance for training a language model.

The rest of the training pipeline, as well as the model, is similar to that for sequential labeling mentioned earlier in this chapter. See the Colab notebook for more details. After the model is fully trained, you can construct instances from new texts, turn them into instances, and compute the loss, which basically measures how successful the model was in predicting what comes next:

```
predict('The trip to the beach was ruined by bad weather.', model)
{'loss': 1.3882852}

predict('The trip to the beach was ruined by bad dogs.', model)
{'loss': 1.5099115}

predict('by weather was trip my bad beach the ruined to.', model)
{'loss': 1.8084583}
```

The loss here is the cross-entropy loss between the predicted and the expected characters. The more "unexpected" the characters are, the higher the values will be, so you can use these values to measure how natural the input is as English text. As expected, natural sentences (such as the first one) are given scores that are lower than unnatural sentences (such as the last one).

NOTE If you calculate 2 to the power of the cross-entropy, the value is called *perplexity*. Given a fixed natural language text, perplexity becomes lower as the language model is better at predicting what comes next, so it is commonly used for evaluating the quality of language models in the literature.

5.6.3 Generating text using language model

The most interesting aspect of (fully trained) language models is that it can predict possible characters that may appear next given some context. Specifically, they can give you a probability distribution over possible characters that may come next, from which you choose to determine the next character. For example, if the model has generated *t* and *h*, and the LM is trained on generic English text, it would probably assign a high probability on the letter *e*, generating common English words including *the*, *they*, *them*, and so on. If you start this process from the `<START>` token and keep doing this until you reach the end of the sentence (that is, by generating `<END>`), you can generate an English sentence from scratch. By the way, this is another reason why tokens such as `<START>` and `<END>` are useful—you need something to feed to the RNN to kick off the generation, and you also need to know when the sentence stops.

Let's look at this process in a Python-like pseudo-code:

```
def generate():
    state = init_state()
    token = <START>
    tokens = [<START>]
    while token != <END>:
        state = update(state, token)
        probs = softmax(linear(state))
        token = sample(probs)
        tokens.append(token)
    return tokens
```

This loop looks similar to the one for updating RNNs but with one key difference: here, we aren't receiving any input, but instead generating characters and feeding them as the input. In other words, the RNN operates on the sequence of characters that the RNN itself generated so far. Such models that operate on past sequence they produced are called *autoregressive models*. See figure 5.14 for an illustration of this.

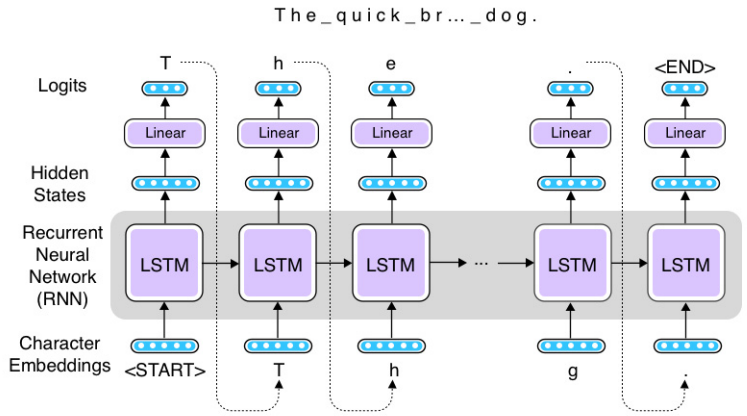


Figure 5.14 Generating text using an RNN.

In this code snippet, `init_state()` and `update()` functions are the ones that initialize and update the hidden states of the RNN, as we've seen earlier. In generating text, we

assume that the model and its parameters are already trained on a large amount of natural language text. `softmax()` is a function to run Softmax on the given vector, and `linear()` is the linear layer to expand/shrink the size of the vector. The function `sample()` returns a character according to the given probability distribution. For example, if the distribution is a:0.6, b:0.3, c:0.1, it will choose a 60% of the time, b 30% of the time, and c 10% of the time. This ensures that the generated string is different every time while every string is likely to look like a real English sentence.

> **NOTE** You can use PyTorch's `torch.multinomial()` for sampling an element from a probability distribution.

If you train this language model using the English sentences from Tatoeba and generate sentences according to this algorithm, the system will produce something similar to the following cherry-picked examples:

```
You can say that you don't know it, and why decided of yourself.
Pike of your value is to talk of hubies.
The meeting despoit from a police?
That's a problem, but us?
The sky as going to send nire into better.
We'll be look of the best ever studented.
There's you seen anything every's redusention day.
How a fail is to go there.
It sad not distaples with money.
What you see him go as famous to eat!
```

This isn't a bad start! If you look at these sentences, there are many words and phrases that make sense as valid English (*You can say that*, *That's a problem*, *to go there*, *see him go*, and so on). Even when the system generates peculiar words (*despoit*, *studented*, *redusention*, *distaples*), they look almost like real English words because they all basically follow morphological and phonological rules of English. This means the language model was successful in learning the basic building blocks of English, such as how to arrange letters (orthography), how to form words (morphology), and how to form basic sentence structures (syntax).

However, if you look at sentences as a whole, few of them make any sense (for example, *What you see him go as famous to eat!*). This means the language model we trained falls short of modeling semantic consistency of sentences. This is potentially due to the fact that our model isn't powerful enough (our LSTM-RNN needs to compress everything about the sentence into 256-dimensional vector) or the training dataset is too small (just 10k sentences), or both. But you can easily imagine that if we keep increasing the model capacity as well as the size of the train set, the model gets incredibly good at producing realistic natural language text. In February 2019, OpenAI announced that it developed a huge language model based on the Transformer model (chapter 8) trained on 40GB of internet text. The model shows that it can produce realistic looking text that shows near-perfect grammar and long-term topical consistency given a prompt. In fact, the model was so good that OpenAI decided not to release the large model they'd trained due to their concerns about malicious use of

the technology. But it's important to keep in mind that, no matter how intelligent the output looks, their model is trained on the same principle as our toy example in this chapter—just trying to predict the next character!

Summary

- Sequential labeling models tag each word in the input with a label, which can be achieved by recurrent neural networks (RNNs).
- Part-of-speech (POS) tagging and named entity recognition (NER) are two instances of sequential labeling tasks.
- Multi-layer RNNs stack multiple layers of RNNs, while bidirectional RNNs combine forward and backward RNNs to encode the entire sentence.
- Language models assign probabilities to natural language text, which is achieved by predicting the next word.
- You can use a trained language model to assess how "natural" a natural language sentence is, or even to generate realistic-looking text from scratch.

Chapter 10 from *Natural Language Processing in Action* by Hobson Lane, Cole Howard, and Hannes Max Hapke

Building on the fixed-dimension word vectors of Chapter 6, Chapter 10 shows you how to teach a machine to handle variable-length sequences of text. This deceptively small broadening of the capability of your Natural Language Processing pipeline opens a world of new applications, from machine translation to conversational chat and question answering. Sequence-to-sequence models enable your machine to say something entirely new and sensible with a few lines of code. These models can even be used to teach a machine to describe the contents of a video clip. This chapter shows you how generative natural language models can be trained on a small portion of the vast amount of freely available natural language text flowing around us all the time. The data and the fun you can have are virtually endless.

Sequence-to-sequence models and attention

This chapter covers

- Mapping one text sequence to another with a neural network
- Understanding sequence-to-sequence tasks and how they're different from the others you've learned about
- Using encoder-decoder model architectures for translation and chat
- Training a model to pay attention to what is important in a sequence

You now know how to create natural language models and use them for everything from sentiment classification to generating novel text (see chapter 9).

Could a neural network translate from English to German? Or even better, would it be possible to predict disease by translating genotype to phenotype (genes to body type)?[34] And what about the chatbot we've been talking about since the

[34] geno2pheno: https://academic.oup.com/nar/article/31/13/3850/2904197.

beginning of the book? Can a neural net carry on an entertaining conversation? These are all sequence-to-sequence problems. They map one sequence of indeterminate length to another sequence whose length is also unknown.

In this chapter, you'll learn how to build sequence-to-sequence models using an encoder-decoder architecture.

10.1 Encoder-decoder architecture

Which of our previous architectures do you think might be useful for sequence-to-sequence problems? The word vector embedding model of chapter 6? The convolutional net of chapter 7 or the recurrent nets of chapter 8 and chapter 9? You guessed it; we're going to build on the LSTM architecture from the last chapter.

LSTMs are great at handling sequences, but it turns out we need two of them rather than only one. We're going to build a modular architecture called an encoder-decoder architecture.

The first half of an encoder-decoder model is the sequence *encoder*, a network which turns a sequence, such as natural language text, into a lower-dimensional representation, such as the thought vector from the end of chapter 9. So you've already built this first half of our sequence-to-sequence model.

The other half of an encoder-decoder architecture is the sequence *decoder*. A sequence decoder can be designed to turn a vector back into human readable text again. But didn't we already do that too? You generated some pretty crazy Shakespearean playscript at the end of chapter 9. That was close, but there are a few more pieces you need to add to get that Shakespearean playwright bot to focus on our new task as a translating scribe.

For example, you might like your model to output the German translation of an English input text. Actually, isn't that just like having our Shakespeare bot translate modern English into Shakespearean? Yes, but in the Shakespeare example we were OK with rolling the dice to let the machine learning algorithm choose any words that matched the probabilities it had learned. That's not going to cut it for a translation service, or for that matter, even a decent playwright bot.

So you already know how to build encoders and decoders; you now need to learn how to make them better, more focused. In fact, the LSTMs from chapter 9 work great as encoders of variable-length text. You built them to capture the meaning and sentiment of natural language text. LSTMs capture that meaning in an internal representation, a thought vector. You just need to extract the thought vector from the state (memory cell) within your LSTM model. You learned how to set `return_state=True` on a Keras LSTM model so that the output includes the hidden layer state. That state vector becomes the output of your encoder and the input to your decoder.

TIP Whenever you train any neural network model, each of the internal layers contains all the information you need to solve the problem you trained it on. That information is usually represented by a fixed-dimensional tensor containing the weights or the activations of that layer. And if your network

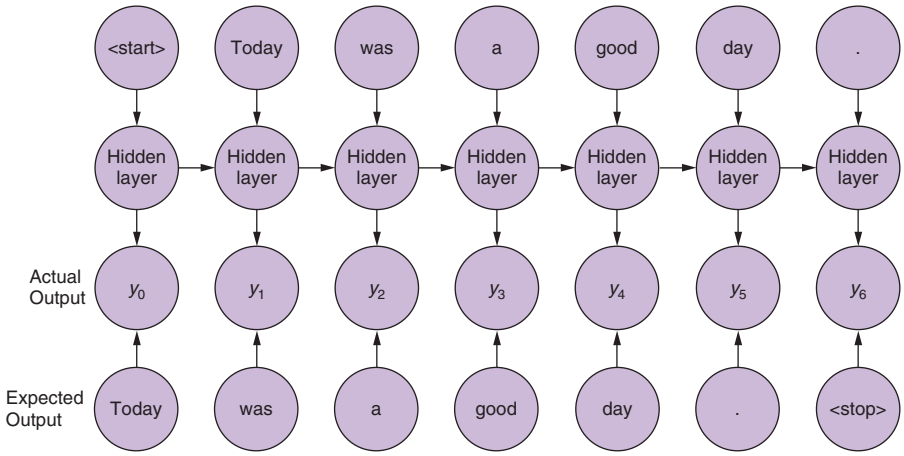
generalizes well, you can be sure that an information bottleneck exists—a layer where the number of dimensions is at a minimum. In Word2vec (see chapter 6), the *weights* of an internal layer were used to compute your vector representation. You can also use the *activations* of an internal network layer directly. That's what the examples in this chapter do. Examine the successful networks you've build in the past to see if you can find this information bottleneck that you can use as an encoded representation of your data.

So all that remains is to improve upon the decoder design. You need to decode a thought vector back into a natural language sequence.

10.1.1 Decoding thought

Imagine you'd like to develop a translation model to translate texts from English to German. You'd like to map sequences of characters or words to another sequence of characters or words. You previously discovered how you can predict a sequence element at time step t based on the previous element at time step t-1. But directly using an LSTM to map from one language to another runs into problems quickly. For a single LSTM to work, you would need input and output sequences to have the same sequence lengths, and for translation they rarely do.

Figure 10.1 demonstrates the problem. The English and the German sentence have different lengths, which complicates the mapping between the English input and the expected output. The English phrase "is playing" (present progressive) is translated to the German present tense "spielt." But "spielt" here would need to be predicted solely on the input of "is;" you haven't gotten to "playing" yet at that time step. Further,



Expected output is the next token in the sample. Shown here on word level.

Figure 10.1 Limitations of language modeling

"playing" would then need to map to "Fußball." Certainly a network could learn these mappings, but the learned representations would have to be hyper-specific to the input, and your dream of a more general language model would go out the window.

Sequence-to-sequence networks, sometimes abbreviated with *seq2seq*, solve this limitation by creating an input representation in the form of a thought vector. Sequence-to-sequence models then use that thought vector, sometimes called a context vector, as a starting point to a second network that receives a different set of inputs to generate the output sequence.

> **THOUGHT VECTOR** Remember when you discovered word vectors? Word vectors are a compression of the meaning of a word into a fixed length vector. Words with similar meaning are close to each other in this vector space of word meanings. A thought vector is very similar. A neural network can compress information from any natural language statement, not just a single word, into a fixed length vector that represents the content of the input text. Thought vectors are this vector. They are used as a numerical representation of the thought within a document to drive some decoder model, usually a translation decoder. The term was coined by Geoffrey Hinton in a talk to the Royal Society in London in 2015.[35]

A sequence-to-sequence network consists of two modular recurrent networks with a thought vector between them (see figure 10.2). The encoder outputs a thought vector at the end of its input sequence. The decoder picks up that thought and outputs a sequence of tokens.

The first network, called the encoder, turns the input text (such as a user message to a chatbot) into the thought vector. The thought vector has two parts, each a vector: the output (activation) of the hidden layer of the encoder and the memory state of the LSTM cell for that input example.

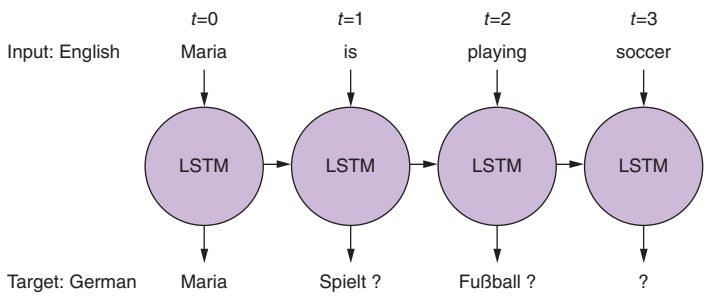


Figure 10.2 Encoder-decoder sandwich with thought vector meat

[35] See the web page titled "Deep Learning," (https://www.evl.uic.edu/creativecoding/courses/cs523/slides/week3/DeepLearning_LeCun.pdf).

TIP As shown in listing 10.1 later in this chapter, the thought vector is captured in the variable names `state_h` (output of the hidden layer) and `state_c` (the memory state).

The thought vector then becomes the input to a second network: the decoder network. As you'll see later in the implementation section, the generated state (thought vector) will serve as the *initial state* of the decoder network. The second network then uses that initial state and a special kind of input, a *start token*. Primed with that information, the second network has to learn to generate the first element of the target sequence (such as a character or word).

The training and inference stages are treated differently in this particular setup. During training, you pass the starting text to the encoder and the *expected* text as the input to the decoder. You're getting the decoder network to learn that, given a primed state and a key to "get started," it should produce a series of tokens. The first direct input to the decoder will be the start token; the second input should be the first expected or predicted token, which should in turn prompt the network to produce the second expected token.

At inference time, however, you don't have the expected text, so what do you use to pass into the decoder other than the state? You use the generic start token and then take the first generated element, which will then become the input to the decoder at the next time step, to generate the next element, and so on. This process repeats until the maximum number of sequence elements is reached or a stop token is generated.

Trained end-to-end this way, the decoder will turn a thought vector into a fully decoded response to the initial input sequence (such as the user question). Splitting the solution into two networks with the thought vector as the binding piece in-between allows you to map input sequences to output sequences of different lengths (see figure 10.3).

10.1.2 Look familiar?

It may seem like you've seen an encoder-decoder approach before. You may have. Autoencoders are a common encoder-decoder architecture for students learning about neural networks. They are a repeat-game-playing neural net that's trained to

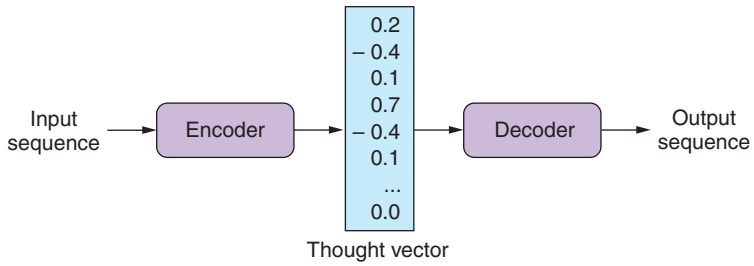


Figure 10.3 Unrolled encoder-decoder

regurgitate its input, which makes finding training data easy. Nearly any large set of high-dimensional tensors, vectors, or sequences will do.

Like any encoder-decoder architecture, autoencoders have a bottleneck of information between the encoder and decoder that you can use as a lower-dimensional representation of the input data. Any network with an information bottleneck can be used as an encoder within an encoder-decoder architecture, even if the network was only trained to paraphrase or restate the input.[36]

Although autoencoders have the same structure as our encoder-decoders in this chapter, they're trained for a different task. Autoencoders are trained to find a vector representation of input data such that the input can be reconstructed by the network's decoder with minimal error. The encoder and decoder are pseudo-inverses of each other. The network's purpose is to find a dense vector representation of the input data (such as an image or text) that allows the decoder to reconstruct it with the smallest error. During the training phase, the input data and the expected output are the same. Therefore, if your goal is finding a dense vector representation of your data—not generating thought vectors for language translation or finding responses for a given question—an autoencoder can be a good option.

What about PCA and t-SNE from chapter 6? Did you use `sklearn.decomposition.PCA` or `sklearn.manifold.TSNE` for visualizing vectors in the other chapters? The t-SNE model produces an embedding as its output, so you can think of it as an encoder, in some sense. The same goes for PCA. However, these models are unsupervised so they can't be targeted at a particular output or task. And these algorithms were developed mainly for feature extraction and visualization. They create very tight bottlenecks to output very low-dimensional vectors, typically two or three. And they aren't designed to take in sequences of arbitrary length. That's what an encoder is all about. And you've learned that LSTMs are the state-of-the-art for extracting features and embeddings from sequences.

> **NOTE** A *variational autoencoder* is a modified version of an autoencoder that is trained to be a good generator as well as encoder-decoder. A variational autoencoder produces a compact vector that not only is a faithful representation of the input but is also Gaussian distributed. This makes it easier to generate a new output by randomly selecting a seed vector and feeding that into the decoder half of the autoencoder.[37]

10.1.3 Sequence-to-sequence conversation

It may not be clear how the dialog engine (conversation) problem is related to machine translation, but they're quite similar. Generating replies in a conversation for

[36] An Autoencoder Approach to Learning Bilingual Word Representations by Chandar and Lauly et al.: https://papers.nips.cc/paper/5270-an-autoencoder-approach-to-learning-bilingual-word-representations.pdf.

[37] See the web page titled "Variational Autoencoders Explained" (http://kvfrans.com/variational-autoencoders-explained).

a chatbot isn't that different from generating a German translation of an English statement in a machine translation system.

Both translation and conversation tasks require your model to map one sequence to another. Mapping sequences of English tokens to German sequences is very similar to mapping natural language statements in a conversation to the expected response by the dialog engine. You can think of the machine translation engine as a schizophrenic, bilingual dialog engine that is playing the childish "echo game,"[38] listening in English and responding in German.

But you want your bot to be responsive, rather than just an echo chamber. So your model needs to bring in any additional information about the world that you want your chatbot to talk about. Your NLP model will have to learn a much more complex mapping from statement to response than echoing or translation. This requires more training data and a higher-dimensional thought vector, because it must contain all the information your dialog engine needs to know about the world. You learned in chapter 9 how to increase the dimensionality, and thus the information capacity, of the thought vector in an LSTM model. You also need to get enough of the right kind of data if you want to turn a translation machine into a conversation machine.

Given a set of tokens, you can train your machine learning pipeline to mimic a conversational response sequence. You need enough of those pairs and enough information capacity in the thought vector to understand all those mappings. Once you have a dataset with enough of these pairs of "translations" from statement to response, you can train a conversation engine using the same network you used for machine translation.

Keras provides modules for building networks for sequence-to-sequence networks with a modular architecture called an encoder-decoder model. And it provides an API to access all the internals of an LSTM network that you need to solve translation, conversation, and even genotype-to-phenotype problems.

10.1.4 LSTM review

In the last chapter, you learned how an LSTM gives recurrent nets a way to selectively remember and forget patterns of tokens they have "seen" within a sample document. The input token for each time step passes through the forget and update gates, is multiplied by weights and masks, and then is stored in a memory cell. The network output at that time step (token) is dictated not solely by the input token, but also by a combination of the input *and* the memory unit's current state.

Importantly, an LSTM shares that token pattern recognizer between documents, because the forget and update gates have weights that are trained as they read many documents. So an LSTM doesn't have to relearn English spelling and grammar with each new document. And you learned how to activate these token patterns stored in

[38] Also called the "repeat game," http://uncyclopedia.wikia.com/wiki/Childish_Repeating_Game.

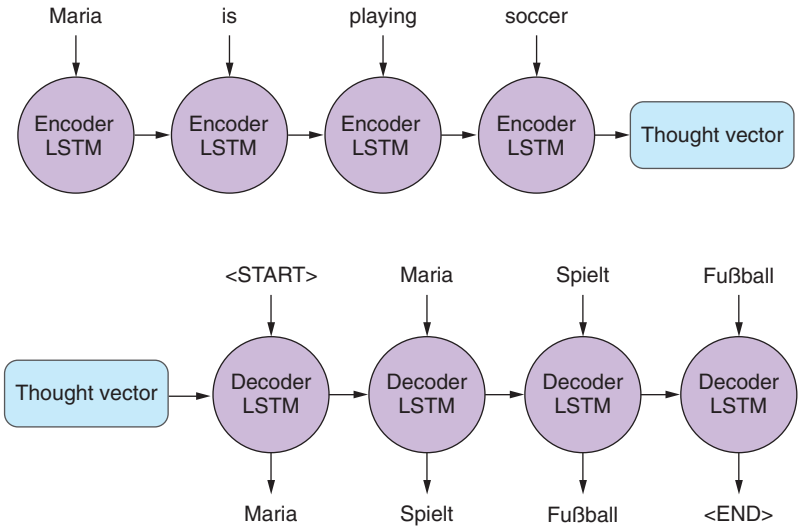


Figure 10.4 Next word prediction

the weights of an LSTM memory cell to predict the tokens that follow based on some seed tokens to trigger the sequence generation (see figure 10.4).

With a token-by-token prediction, you were able to generate some text by selecting the next token based on the probability distribution of likely next tokens suggested by the network. Not perfect by any stretch, but entertaining nonetheless. But you aren't here for mere entertainment; you'd like to have some control over what comes out of a generative model.

Sutskever, Vinyals, and Le came up with a way to bring in a second LSTM model to *decode* the patterns in the memory cell in a less random and more controlled way.[39] They proposed using the classification aspect of the LSTM to create a thought vector and then use that generated vector as the input to a second, *different* LSTM that only tries to predict token by token, which gives you a way to map an input sequence to a distinct output sequence. Let's take a look at how it works.

10.2 Assembling a sequence-to-sequence pipeline

With your knowledge from the previous chapters, you have all the pieces you need to assemble a sequence-to-sequence machine learning pipeline.

10.2.1 Preparing your dataset for the sequence-to-sequence training

As you've seen in previous implementations of convolutional or recurrent neural networks, you need to pad the input data to a fixed length. Usually, you'd extend the

[39] Sutskever, Vinyals, and Le; arXiv:1409.3215, http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf.

input sequences to match the longest input sequence with pad tokens. In the case of the sequence-to-sequence network, you also need to prepare your target data and pad it to match the longest target sequence. Remember, the sequence lengths of the input and target data don't need to be the same (see figure 10.5).

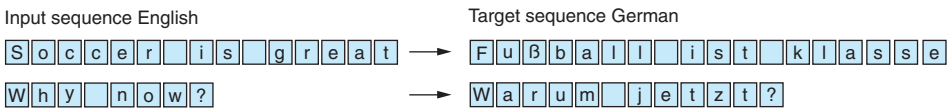


Figure 10.5 **Input and target sequence before preprocessing**

In addition to the required padding, the output sequence should be annotated with the start and stop tokens, to tell the decoder when the job starts and when it's done (see figure 10.6).

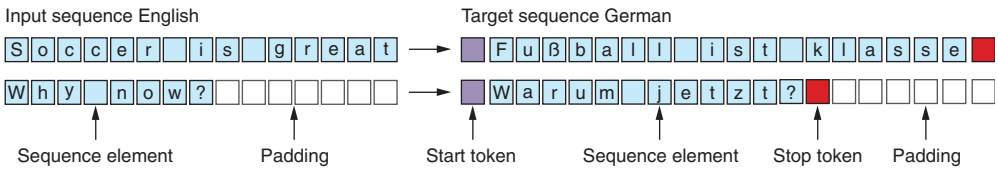


Figure 10.6 **Input and target sequence after preprocessing**

You'll learn how to annotate the target sequences later in the chapter when you build the Keras pipeline. Just keep in mind that you'll need two versions of the target sequence for training: one that starts with the start token (which you'll use for the decoder input), and one that starts without the start token (the target sequence the loss function will score for accuracy).

In earlier chapters, your training sets consisted of pairs: an input and an expected output. Each training example for the sequence-to-sequence model will be a triplet: initial input, expected output (prepended by a start token), and expected output (without the start token).

Before you get into the implementation details, let's recap for a moment. Your sequence-to-sequence network consists of two networks: the encoder, which will generate your thought vector; and a decoder, that you'll pass the thought vector into, as its initial state. With the initialized state and a start token as input to the decoder network, you'll then generate the first sequence element (such as a character or word vector) of the output. Each following element will then be predicted based on the updated state and the next element in the expected sequence. This process will go on

until you either generate a stop token or you reach the maximum number of elements. All sequence elements generated by the decoder will form your predicted output (such as your reply to a user question). With this in mind, let's take a look at the details.

10.2.2 Sequence-to-sequence model in Keras

In the following sections, we guide you through a Keras implementation of a sequence-to-sequence network published by Francois Chollet.[40] Mr. Chollet is also the author of the book *Deep Learning with Python* (Manning, 2017), an invaluable resource for learning neural network architectures and Keras.

During the training phase, you'll train the encoder and decoder network together, end to end, which requires three data points for each sample: a training encoder input sequence, a decoder input sequence, and a decoder output sequence. The training encoder input sequence could be a user question for which you'd like a bot to respond. The decoder input sequence then is the expected reply by the future bot.

You might wonder why you need an input *and* output sequence for the decoder. The reason is that you're training the decoder with a method called *teacher forcing*, where you'll use the initial state provided by the encoder network and train the decoder to produce the expected sequences by showing the input to the decoder and letting it predict the same sequence. Therefore, the decoder's input and output sequences will be identical, except that the sequences have an offset of one time step.

During the execution phase, you'll use the encoder to generate the thought vector of your user input, and the decoder will then generate a reply based on that thought vector. The output of the decoder will then serve as the reply to the user.

> **KERAS FUNCTIONAL API** In the following example, you'll notice a different implementation style of the Keras layers you've seen in previous chapters. Keras introduced an additional way of assembling models by calling each layer and passing the value from the previous layer to it. The functional API can be powerful when you want to build models and reuse portions of the trained models (as you'll demonstrate in the coming sections). For more information about Keras' functional API, we highly recommend the blog post by the Keras core developer team.[41]

10.2.3 Sequence encoder

The encoder's sole purpose is the creation of your thought vector, which then serves as the initial state of the decoder network (see figure 10.7). You can't train an encoder fully in isolation. You have no "target" thought vector for the network to learn to predict. The backpropagation that will train the encoder to create an appropriate

[40] See the web page titled "A ten-minute introduction to sequence-to-sequence learning in Keras" (https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html).

[41] See the web page titled "Getting started with the Keras functional API" (https://keras.io/getting-started/functional-api-guide/).

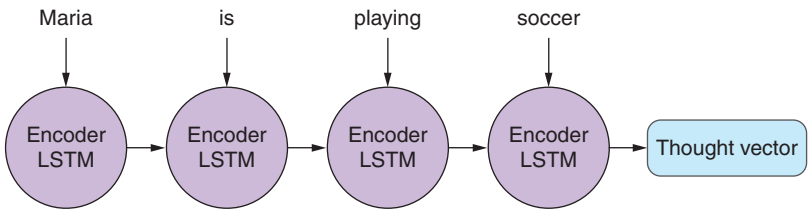


Figure 10.7 Thought encoder

thought vector will come from the error that's generated later downstream in the decoder.

Nonetheless the encoder and decoder are independent modules that are often interchangeable with each other. For example, once your encoder is trained on the English-to-German translation problem, it can be reused with a different encoder for translation from English to Spanish.[42] Listing 10.1 shows what the encoder looks like in isolation.

Conveniently, the RNN layers, provided by Keras, return their internal state when you instantiate the LSTM layer (or layers) with the keyword argument `return_state=True`. In the following snippet, you preserve the final state of the encoder and disregard the actual output of the encoder. The list of the LSTM states is then passed to the decoder.

Listing 10.1 Thought encoder in Keras

```
>>> encoder_inputs = Input(shape=(None, input_vocab_size))
>>> encoder = LSTM(num_neurons, return_state=True)
>>> encoder_outputs, state_h, state_c = encoder(encoder_inputs)
>>> encoder_states = (state_h, state_c)
```

The return_state argument of the LSTM layer needs to be set to True to return the internal states.

The first return value of the LSTM layer is the output of the layer.

Because `return_sequences` defaults to `False`, the first return value is the output from the last time step. `state_h` will be specifically the output of the last time step for this layer. So in this case, `encoder_outputs` and `state_h` will be identical. Either way you can ignore the official output stored in `encoder_outputs`. `state_c` is the current state of the memory unit. `state_h` and `state_c` will make up your thought vector.

[42] Training a multi-task model like this is called "joint training" or "transfer learning" and was described by Luong, Le, Sutskever, Vinyals, and Kaier (Google Brain) at ICLR 2016: https://arxiv.org/pdf/1511.06114.pdf.

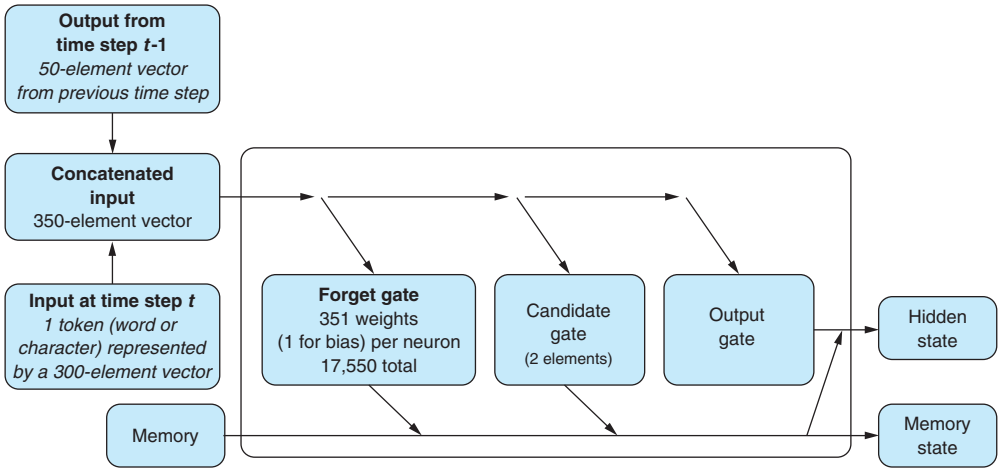


Figure 10.8 **LSTM states used in the sequence-to-sequence encoder**

Figure 10.8 shows how the internal LSTM states are generated. The encoder will update the hidden and memory states with every time step, and pass the final states to the decoder as the initial state.

10.2.4 Thought decoder

Similar to the encoder network setup, the setup of the decoder is pretty straightforward. The major difference is that this time you do want to capture the output of the network at each time step. You want to judge the "correctness" of the output, token by token (see figure 10.9).

This is where you use the second and third pieces of the sample 3-tuple. The decoder has a standard token-by-token input and a token-by-token output. They are almost identical, but off by one time step. You want the decoder to learn to reproduce the tokens of a given input sequence *given* the state generated by the first piece of the 3-tuple fed into the encoder.

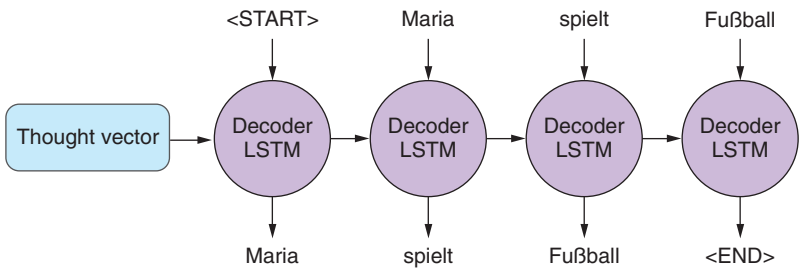


Figure 10.9 **Thought decoder**

NOTE This is the key concept for the decoder, and for sequence-to-sequence models in general; you're training a network to output in the secondary problem space (another language or another being's response to a given question). You form a "thought" about both what was said (the input) and the reply (the output) simultaneously. And this thought defines the response token by token. Eventually, you'll only need the thought (generated by the encoder) and a generic start token to get things going. That's enough to trigger the correct output sequence.

To calculate the error of the training step, you'll pass the output of your LSTM layer into a dense layer. The dense layer will have a number of neurons equal to the number of all possible output tokens. The dense layer will have a softmax activation function across those tokens. So at each time step, the network will provide a probability distribution over all possible tokens for what it thinks is most likely the next sequence element. Take the token whose related neuron has the highest value. You used an output layer with softmax activation functions in earlier chapters, where you wanted to determine a token with the highest likelihood (see chapter 6 for more details). Also note that the `num_encoder_tokens` and the `output_vocab_size` don't need to match, which is one of the great benefits of sequence-to-sequence networks. See the following listing.

Listing 10.2 Thought decoder in Keras

The functional API allows you to pass the initial state to the LSTM layer by assigning the last encoder state to initial_state.

Set up the LSTM layer, similar to the encoder but with an additional argument of return_sequences.

```
>>> decoder_inputs = Input(shape=(None, output_vocab_size))
>>> decoder_lstm = LSTM(
...     num_neurons,return_sequences=True, return_state=True)
>>> decoder_outputs, _, _ = decoder_lstm(
...     decoder_inputs, initial_state=encoder_states)
>>> decoder_dense = Dense(
...     output_vocab_size, activation='softmax')
>>> decoder_outputs = decoder_dense(decoder_outputs)
```

Softmax layer with all possible characters mapped to the softmax output

Passing the output of the LSTM layer to the softmax layer

10.2.5 Assembling the sequence-to-sequence network

The functional API of Keras allows you to assemble a model as object calls. The `Model` object lets you define its input and output parts of the network. For this sequence-to-sequence network, you'll pass a list of your inputs to the model. In listing 10.2, you defined one input layer in the encoder and one in the decoder. These two inputs correspond with the first two elements of each training triplet. As an output layer, you're passing the `decoder_outputs` to the model, which includes the entire model setup you previously defined. The output in `decoder_outputs` corresponds with the final element of each of your training triplets.

NOTE Using the functional API like this, definitions such as `decoder _outputs` are *tensor* representations. This is where you'll notice differences from the sequential model described in earlier chapters. Again refer to the documentation for the nitty-gritty of the Keras API. See the following listing.

Listing 10.3 Keras functional API (`Model()`)

```
>>> model = Model(
...     inputs=[encoder_inputs, decoder_inputs],
...     outputs=decoder_outputs)
```

The inputs and outputs arguments can be defined as lists if you expect multiple inputs or outputs.

10.3 Training the sequence-to-sequence network

The last remaining steps for creating a sequence-to-sequence model in the Keras model are to compile and fit. The only difference compared to earlier chapters is that earlier you were predicting a binary classification: yes or no. But here you have a categorical classification or multiclass classification problem. At each time step you must determine which of many "categories" is correct. And we have many categories here. The model must choose between all possible tokens to "say." Because you're predicting characters or words rather than binary states, you'll optimize your loss based on the `categorical_crossentropy` loss function, rather than the `binary_ crossentropy` used earlier. So that's the only change you need to make to the Keras `model.compile` step, as shown in the following listing.

Listing 10.4 Train a sequence-to-sequence model in Keras

```
>>> model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
>>> model.fit([encoder_input_data, decoder_input_data],
               decoder_target_data,
              batch_size=batch_size, epochs=epochs)
```

Setting the loss function to categorical_ crossentropy.

The model expects the training inputs as a list, where the first list element is passed to the encoder network and the second element is passed to the decoder network during the training.

Congratulations! With the call to `model.fit`, you're training your sequence-to-sequence network, end to end. In the following sections, you'll demonstrate how you can infer an output sequence for a given input sequence.

NOTE The training of sequence-to-sequence networks can be computationally intensive and therefore time-consuming. If your training sequences are long or if you want to train with a large corpus, we highly recommend training these networks on a GPU, which can increase the training speed by 30 times. If you've never trained a neural network on a GPU, don't worry. Check out chapter 13 on how to rent and set up your own GPU on commercial computational cloud services.

LSTMs aren't inherently parallelizable like convolutional neural nets, so to get the full benefit of a GPU you should replace the LSTM layers with `CuDNNLSTM`, which is optimized for training on a GPU enabled with CUDA.

10.3.1 Generate output sequences

Before generating sequences, you need to take the structure of your training layers and reassemble them for generation purposes. At first, you define a model specific to the encoder. This model will then be used to generate the thought vector. See the following listing.

Listing 10.5 Decoder for generating text using the generic Keras `Model`

```
>>> encoder_model = Model(inputs=encoder_inputs, outputs=encoder_states)
```

Here you use the previously defined encoder_inputs and encoder_states; calling the predict method on this model would return the thought vector.

The definition of the decoder can look daunting. But let's untangle the code snippet step by step. First, you'll define your decoder inputs. You are using the Keras input layer, but instead of passing in one-hot vectors, characters, or word embeddings, you'll pass the thought vector generated by the encoder network. Note that the encoder returns a list of two states, which you'll need to pass to the `initial_state` argument when calling your previously defined `decoder_lstm`. The output of the LSTM layer is then passed to the dense layer, which you also previously defined. The output of this layer will then provide the probabilities of all decoder output tokens (in this case, all seen characters during the training phase).

Here is the magic part. The token predicted with the highest probability at each time step will then be returned as the most likely token and passed on to the next decoder iteration step, as the new input. See the following listing.

Listing 10.6 Sequence generator for random thoughts

```
>>> thought_input = [Input(shape=(num_neurons,)),
...     Input(shape=(num_neurons,))]
>>> decoder_outputs, state_h, state_c = decoder_lstm(
...     decoder_inputs, initial_state=thought_input)
>>> decoder_states = [state_h, state_c]
>>> decoder_outputs = decoder_dense(decoder_outputs)

>>> decoder_model = Model(
...     inputs=[decoder_inputs] + thought_input,
...     output=[decoder_outputs] + decoder_states)
```

Define an input layer to take the encoder states.

Pass the encoder state to the LSTM layer as initial state.

The updated LSTM state will then become the new cell state for the next iteration.

The last step is tying the decoder model together.

Pass the output from the LSTM to the dense layer to predict the next token.

The output of the dense layer and the updated states are defined as output.

The decoder_inputs and thought_input become the input to the decoder model.

Once the model is set up, you can generate sequences by predicting the thought vector based on a one-hot encoded input sequence and the last generated token. During the first iteration, the `target_seq` is set to the start token. During all following iterations, `target_seq` is updated with the last generated token. This loop goes on until either you've reached the maximum number of sequence elements or the decoder generates a stop token, at which time the generation is stopped. See the following listing.

Listing 10.7 Simple decoder—next word prediction

```
...
>>> thought = encoder_model.predict(input_seq)
...
>>> while not stop_condition:
...     output_tokens, h, c = decoder_model.predict(
...         [target_seq] + thought)
```

Encode the input sequence into a thought vector (the LSTM memory cell state).

The stop_condition is updated after each iteration and turns True if either the maximum number of output sequence tokens is hit or the decoder generates a stop token.

The decoder returns the token with the highest probability and the internal states, which are reused during the next iteration.

10.4 Building a chatbot using sequence-to-sequence networks

In the previous sections, you learned how to train a sequence-to-sequence network and how to use the trained network to generate sequence responses. In the following section, we guide you through how to apply the various steps to train a chatbot. For the chatbot training, you'll use the Cornell movie dialog corpus.[43] You'll train a sequence-to-sequence network to "adequately" reply to your questions or statements. Our chatbot example is an adopted sequence-to-sequence example from the Keras blog.[44]

10.4.1 Preparing the corpus for your training

First, you need to load the corpus and generate the training sets from it. The training data will determine the set of characters the encoder and decoder will support during the training and during the generation phase. Please note that this implementation doesn't support characters that haven't been included during the training phase. Using the entire Cornell Movie Dialog dataset can be computationally intensive because a few sequences have more than 2,000 tokens—2,000 time steps will take a while to unroll. But the majority of dialog samples are based on less than 100 characters. For this example, you've preprocessed the dialog corpus by limiting samples to those with fewer than 100 characters, removed odd characters, and only allowed

[43] See the web page titled "Cornell Movie-Dialogs Corpus" (https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html).

[44] See the web page titled "keras/examples/lstm_seq2seq.py at master" (https://github.com/fchollet/keras/blob/master/examples/lstm_seq2seq.py).

lowercase characters. With these changes, you limit the variety of characters. You can find the preprocessed corpus in the GitHub repository of *NLP in Action*.[45]

You'll loop over the corpus file and generate the training pairs (technically 3-tuples: input text, target text with start token, and target text). While reading the corpus, you'll also generate a set of input and target characters, which you'll then use to one-hot encode the samples. The input and target characters don't have to match. But characters that aren't included in the sets can't be read or generated during the generation phase. The result of the following listing is two lists of input and target texts (strings), as well as two sets of characters that have been seen in the training corpus.

Listing 10.8 Build character sequence-to-sequence training set

The sets hold the seen characters in the input and target text.

The arrays hold the input and target text read from the corpus file.

The target sequence is annotated with a start (first) and stop (last) token; the characters representing the tokens are defined here. These tokens can't be part of the normal sequence text and should be uniquely used as start and stop tokens.

max_training_samples defines how many lines are used for the training. It's the lower number of either a user-defined maximum or the total number of lines loaded from the file.

```
>>> from nlpia.loaders import get_data
>>> df = get_data('moviedialog')
>>> input_texts, target_texts = [], []
>>> input_vocabulary = set()
>>> output_vocabulary = set()
>>> start_token = '\t'
>>> stop_token = '\n'
>>> max_training_samples = min(25000, len(df) - 1)

>>> for input_text, target_text in zip(df.statement, df.reply):
...     target_text = start_token + target_text \
...         + stop_token
...     input_texts.append(input_text)
...     target_texts.append(target_text)
...     for char in input_text:
...         if char not in input_vocabulary:
...             input_vocabulary.add(char)
...     for char in target_text:
...         if char not in output_vocabulary:
...             output_vocabulary.add(char)
```

The target_text needs to be wrapped with the start and stop tokens.

Compile the vocabulary—set of the unique characters seen in the input_texts.

10.4.2 Building your character dictionary

Similar to the examples from your previous chapters, you need to convert each character of the input and target texts into one-hot vectors that represent each character. In order to generate the one-hot vectors, you generate token dictionaries (for the input and target text), where every character is mapped to an index. You also generate the reverse dictionary (index to character), which you'll use during the generation phase to convert the generated index to a character. See the following listing.

[45] See the web page titled "GitHub - totalgood/nlpia" (https://github.com/totalgood/nlpia).

Listing 10.9 Character sequence-to-sequence model parameters

```
>>> input_vocabulary = sorted(input_vocabulary)
>>> output_vocabulary = sorted(output_vocabulary)

>>> input_vocab_size = len(input_vocabulary)
>>> output_vocab_size = len(output_vocabulary)
>>> max_encoder_seq_length = max(
...     [len(txt) for txt in input_texts])
>>> max_decoder_seq_length = max(
...     [len(txt) for txt in target_texts])

>>> input_token_index = dict([(char, i) for i, char in
...     enumerate(input_vocabulary)])
>>> target_token_index = dict(
...     [(char, i) for i, char in enumerate(output_vocabulary)])
>>> reverse_input_char_index = dict((i, char) for char, i in
...     input_token_index.items())
>>> reverse_target_char_index = dict((i, char) for char, i in
...     target_token_index.items())
```

For the input and target data, you also determine the maximum number of sequence tokens.

You convert the character sets into sorted lists of characters, which you then use to generate the dictionary.

For the input and target data, you determine the maximum number of unique characters, which you use to build the one-hot matrices.

Loop over the input_characters and output_vocabulary to create the lookup dictionaries, which you use to generate the one-hot vectors.

Loop over the newly created dictionaries to create the reverse lookups.

10.4.3 Generate one-hot encoded training sets

In the next step, you're converting the input and target text into one-hot encoded "tensors." In order to do that, you loop over each input and target sample, and over each character of each sample, and one-hot encode each character. Each character is encoded by an *n x 1* vector (with *n* being the number of unique input or target characters). All vectors are then combined to create a matrix for each sample, and all samples are combined to create the training tensor. See the following listing.

Listing 10.10 Construct character sequence encoder-decoder training set

```
>>> import numpy as np

>>> encoder_input_data = np.zeros((len(input_texts),
...     max_encoder_seq_length, input_vocab_size),
...     dtype='float32')
>>> decoder_input_data = np.zeros((len(input_texts),
...     max_decoder_seq_length, output_vocab_size),
...     dtype='float32')
>>> decoder_target_data = np.zeros((len(input_texts),
...     max_decoder_seq_length, output_vocab_size),
...     dtype='float32')

>>> for i, (input_text, target_text) in enumerate(
...             zip(input_texts, target_texts)):
```

You use numpy for the matrix manipulations.

The training tensors are initialized as zero tensors with shape (num_samples, max_len_sequence, num_unique_tokens_in_vocab).

Loop over the training samples; input and target texts need to correspond.

```
...     for t, char in enumerate(input_text):
...         encoder_input_data[
...             i, t, input_token_index[char]] = 1.
...     for t, char in enumerate(target_text):
...         decoder_input_data[
...             i, t, target_token_index[char]] = 1.
...         if t > 0:
...             decoder_target_data[i, t - 1, target_token_index[char]] = 1
```

Loop over each character of each sample.

Set the index for the character at each time step to one; all other indices remain at zero. This creates the one-hot encoded representation of the training samples.

For the training data for the decoder, you create the decoder_input_data and decoder_target_data (which is one time step behind the decoder_input_data).

10.4.4 Train your sequence-to-sequence chatbot

After all the training set preparation—converting the preprocessed corpus into input and target samples, creating index lookup dictionaries, and converting the samples into one-hot tensors—it's time to train the chatbot. The code is identical to the earlier samples. Once the `model.fit` completes the training, you have a fully trained chatbot based on a sequence-to-sequence network. See the following listing.

Listing 10.11 Construct and train a character sequence encoder-decoder network

In this example, you set the batch size to 64 samples. Increasing the batch size can speed up the training; it might also require more memory.

Training a sequence-to-sequence network can be lengthy and easily require 100 epochs.

In this example, you set the number of neuron dimensions to 256.

You withhold 10% of the samples for validation tests after each epoch.

```
>>> from keras.models import Model
>>> from keras.layers import Input, LSTM, Dense

>>> batch_size = 64
>>> epochs = 100
>>> num_neurons = 256

>>> encoder_inputs = Input(shape=(None, input_vocab_size))
>>> encoder = LSTM(num_neurons, return_state=True)
>>> encoder_outputs, state_h, state_c = encoder(encoder_inputs)
>>> encoder_states = [state_h, state_c]

>>> decoder_inputs = Input(shape=(None, output_vocab_size))
>>> decoder_lstm = LSTM(num_neurons, return_sequences=True,
...                     return_state=True)
>>> decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
...     initial_state=encoder_states)
>>> decoder_dense = Dense(output_vocab_size, activation='softmax')
>>> decoder_outputs = decoder_dense(decoder_outputs)
>>> model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

>>> model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
...               metrics=['acc'])
>>> model.fit([encoder_input_data, decoder_input_data],
...     decoder_target_data, batch_size=batch_size, epochs=epochs,
...     validation_split=0.1)
```

10.4.5 Assemble the model for sequence generation

Setting up the model for the sequence generation is very much the same as we discussed in the earlier sections. But you have to make some adjustments, because you don't have a specific target text to feed into the decoder along with the state. All you have is the input, and a start token. See the following listing.

Listing 10.12 Construct response generator model

```
>>> encoder_model = Model(encoder_inputs, encoder_states)
>>> thought_input = [
...     Input(shape=(num_neurons,)), Input(shape=(num_neurons,))]
>>> decoder_outputs, state_h, state_c = decoder_lstm(
...     decoder_inputs, initial_state=thought_input)
>>> decoder_states = [state_h, state_c]
>>> decoder_outputs = decoder_dense(decoder_outputs)

>>> decoder_model = Model(
...     inputs=[decoder_inputs] + thought_input,
...     output=[decoder_outputs] + decoder_states)
```

10.4.6 Predicting a sequence

The `decode_sequence` function is the heart of the response generation of your chatbot. It accepts a one-hot encoded input sequence, generates the thought vector, and uses the thought vector to generate the appropriate response by using the network trained earlier. See the following listing.

Listing 10.13 Build a character-based translator

```
>>> def decode_sequence(input_seq):
...     thought = encoder_model.predict(input_seq)

...     target_seq = np.zeros((1, 1, output_vocab_size))
...     target_seq[0, 0, target_token_index[stop_token]
...         ] = 1.
...     stop_condition = False
...     generated_sequence = ''

...     while not stop_condition:
...         output_tokens, h, c = decoder_model.predict(
...             [target_seq] + thought)

...         generated_token_idx = np.argmax(output_tokens[0, -1, :])
...         generated_char = reverse_target_char_index[generated_token_idx]
...         generated_sequence += generated_char
...         if (generated_char == stop_token or
...                 len(generated_sequence) > max_decoder_seq_length
...                 ):
...             stop_condition = True
```

Generate the thought vector as the input to the decoder.

In contrast to the training, target_seq starts off as a zero tensor.

The first input token to the decoder is the start token.

Passing the already-generated tokens and the latest state to the decoder to predict the next sequence element

Setting the stop_condition to True will stop the loop.

```
...         target_seq = np.zeros((1, 1, output_vocab_size))
...         target_seq[0, 0, generated_token_idx] = 1.
...         thought = [h, c]

...     return generated_sequence
```

Update the target sequence and use the last generated token as the input to the next generation step.

Update the thought vector state.

10.4.7 Generating a response

Now you'll define a helper function, `response()`, to convert an input string (such as a statement from a human user) into a reply for the chatbot to use. This function first converts the user's input text into a sequence of one-hot encoded vectors. That tensor of one-hot vectors is then passed to the previously defined `decode_sequence()` function. It accomplishes the encoding of the input texts into thought vectors and the generation of text from those thought vectors.

> **NOTE** The key is that instead of providing an initial state (thought vector) and an input sequence to the decoder, you're supplying only the thought vector and a start token. The token that the decoder produces given the initial state and the start token becomes the input to the decoder at time step 2. And the output at time step 2 becomes the input at time step 3, and so on. All the while the LSTM memory state is updating the memory and augmenting output as it goes—just like you saw in chapter 9:

Loop over each character of the input text to generate the one-hot tensor for the encoder to generate the thought vector from.

```
>>> def response(input_text):
...     input_seq = np.zeros((1, max_encoder_seq_length, input_vocab_size),
...         dtype='float32')
...     for t, char in enumerate(input_text):
...         input_seq[0, t, input_token_index[char]] = 1.
...     decoded_sentence = decode_sequence(input_seq)
...     print('Bot Reply (Decoded sentence):', decoded_sentence)
```

Use the decode_sequence function to call the trained model and generate the response sequence.

10.4.8 Converse with your chatbot

Voila! You just completed all necessary steps to train and use your own chatbot. Congratulations! Interested in what the chatbot can reply to? After 100 epochs of training, which took approximately seven and a half hours on an NVIDIA GRID K520 GPU, the trained sequence-to-sequence chatbot was still a bit stubborn and short-spoken. A larger and more general training corpus could change that behavior:

```
>>> response("what is the internet?")
Bot Reply (Decoded sentence): it's the best thing i can think of anything.

>>> response("why?")
Bot Reply (Decoded sentence): i don't know. i think it's too late.
```

```
>>> response("do you like coffee?")
Bot Reply (Decoded sentence): yes.

>>> response("do you like football?")
Bot Reply (Decoded sentence): yeah.
```

NOTE If you don't want to set up a GPU and train your own chatbot, no worries. We made the trained chatbot available for you to test it. Head over to the GitHub repository of *NLP in Action*[46] and check out the latest chatbot version. Let the authors know if you come across any funny replies by the chatbot.

10.5 Enhancements

There are two enhancements to the way you train sequence-to-sequence models that can improve their accuracy and scalability. Like human learning, deep learning can benefit from a well-designed curriculum. You need to categorize and order the training material to ensure speedy absorption, and you need to ensure that the instructor highlights the most import parts of any given document.

10.5.1 Reduce training complexity with bucketing

Input sequences can have different lengths, which can add a large number of pad tokens to short sequences in your training data. Too much padding can make the computation expensive, especially when the majority of the sequences are short and only a handful of them use close-to-the-maximum token length. Imagine you train your sequence-to-sequence network with data where almost all samples are 100 tokens long, except for a few outliers that contain 1,000 tokens. Without bucketing, you'd need to pad the majority of your training with 900 pad tokens, and your sequence-to-sequence network would have to loop over them during the training phase. This padding will slow down the training dramatically. Bucketing can reduce the computation in these cases. You can sort the sequences by length and use different sequence lengths during different batch runs. You assign the input sequences to buckets of different lengths, such as all sequences with a length between 5 and 10 tokens, and then use the sequence buckets for your training batches, such as train first with all sequences between 5 and 10 tokens, 10 to 15, and so on. Some deep learning frameworks provide bucketing tools to suggest the optimal buckets for your input data.

As shown in figure 10.10, the sequences were first sorted by length and then only padded to the maximum token length for the particular bucket. That way, you can reduce the number of time steps needed for any particular batch while training the sequence-to-sequence network. You only unroll the network as far as is necessary (to the longest sequence) in a given training batch.

[46] See the web page titled "GitHub - totalgood/nlpia" (https://github.com/totalgood/nlpia).

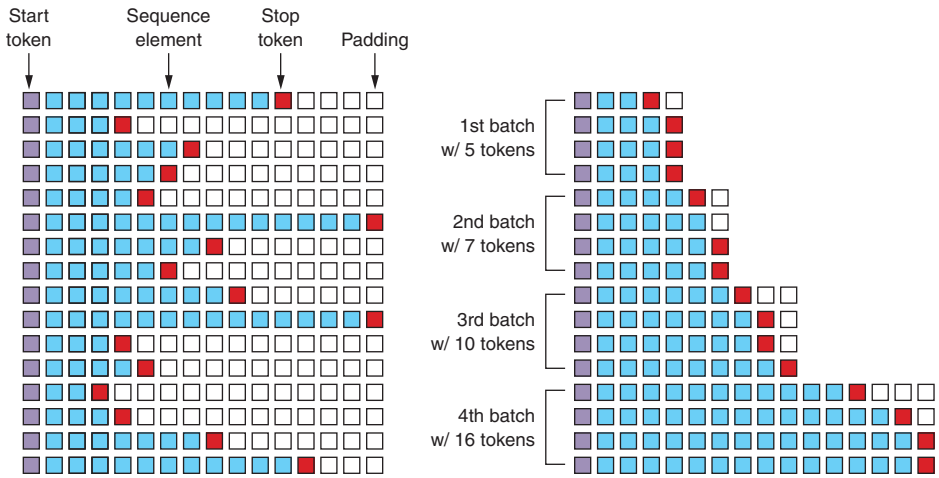


Figure 10.10 **Bucketing applied to target sequences**

10.5.2 Paying attention

As with latent semantic analysis introduced in chapter 4, longer input sequences (documents) tend to produce thought vectors that are less precise representations of those documents. A thought vector is limited by the dimensionality of the LSTM layer (the number of neurons). A single thought vector is sufficient for short input/output sequences, similar to your chatbot example. But imagine the case when you want to train a sequence-to-sequence model to summarize online articles. In this case, your input sequence can be a lengthy article, which should be compressed into a single thought vector to generate such as a headline. As you can imagine, training the network to determine the most relevant information in that longer document is tricky. A headline or summary (and the associated thought vector) must focus on a particular aspect or portion of that document rather than attempt to represent all of the complexity of its meaning.

In 2015, Bahdanau et al. presented their solution to this problem at the International Conference on Learning Representations.[47] The concept the authors developed became known as the *attention mechanism* (see figure 10.11). As the name suggests, the idea is to tell the decoder what to pay attention to in the input sequence. This "sneak preview" is achieved by allowing the decoder to also look all the way back into the states of the encoder network in addition to the thought vector. A version of a "heat map" over the entire input sequence is learned along with the rest of the network. That mapping, different at each time step, is then shared with the decoder. As it decodes any particular part of the sequence, its concept created from the thought vec-

[47] See the web page titled "Neural Machine Translation by Jointly Learning to Align and Translate" (https://arxiv.org/abs/1409.0473).

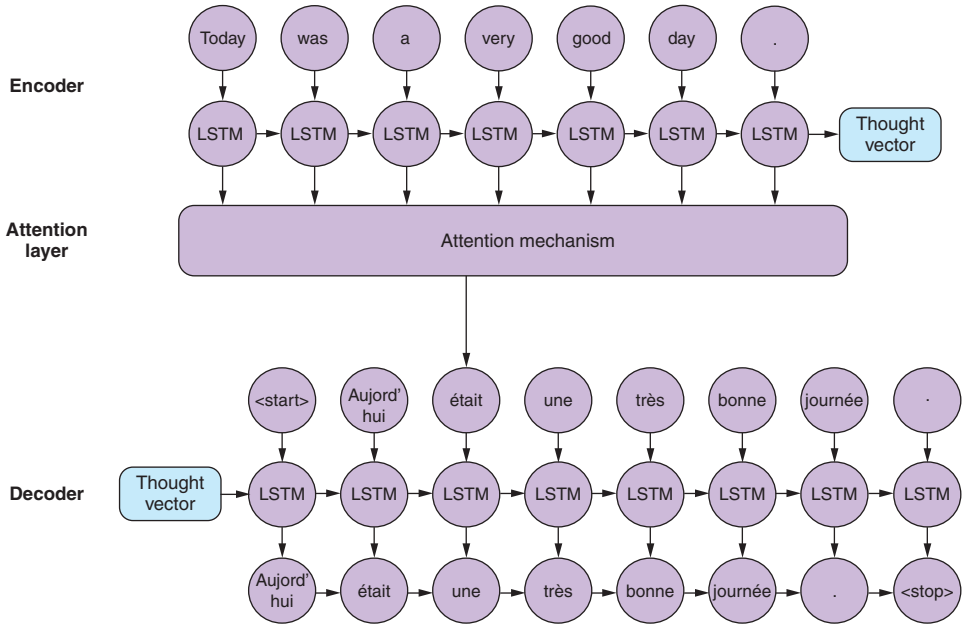


Figure 10.11 Overview of the attention mechanism

tor can be augmented with direct information that it produced. In other words, the attention mechanism allows a direct connection between the output and the input by selecting relevant input pieces. This doesn't mean token-to-token alignment; that would defeat the purpose and send you back to autoencoder land. It does allow for richer representations of concepts wherever they appear in the sequence.

With the attention mechanism, the decoder receives an additional input with every time step representing the one (or many) tokens in the input sequence to pay "attention" to, at this given decoder time step. All sequence positions from the encoder will be represented as a weighted average for each decoder time step.

Configuring and tuning the attention mechanism isn't trivial, but various deep learning frameworks provide implementations to facilitate this. At the time of this writing, a pull request to the Keras package was discussed, but no implementation had yet been accepted.

10.6 In the real world

Sequence-to-sequence networks are well suited for any machine learning application with variable-length input sequences or variable-length output sequences. Since natural language sequences of words almost always have unpredictable length, sequence-to-sequence models can improve the accuracy of most machine learning models.

Key sequence-to-sequence applications are

- Chatbot conversations
- Question answering
- Machine translation
- Image captioning
- Visual question answering
- Document summarization

As you've seen in the previous sections, a dialog system is a common application for NLP. Sequence-to-sequence models are generative, which makes them especially well-suited to conversational dialog systems (chatbots). Sequence-to-sequence chatbots generate more varied, creative, and conversational dialog than information retrieval or knowledge-based chatbot approaches. Conversational dialog systems mimic human conversation on a broad range of topics. Sequence-to-sequence chatbots can generalize from limited-domain corpora and yet respond reasonably on topics not contained in their training set. In contrast, the "grounding" of knowledge-based dialog systems (discussed in chapter 12) can limit their ability to participate in conversations on topics outside their training domains. Chapter 12 compares the performance of chatbot architectures in greater detail.

Besides the Cornell Movie Dialog Corpus, various free and open source training sets are available, such as Deep Mind's Q&A datasets.[48, 49] When you need your dialog system to respond reliably in a specific domain, you'll need to train it on a corpora of statements from that domain. The thought vector has a limited amount of information capacity and that capacity needs to be filled with information on the topics you want your chatbot to be conversant in.

Another common application for sequence-to-sequence networks is machine translation. The concept of the thought vector allows a translation application to incorporate the *context* of the input data, and words with multiple meanings can be translated in the correct context. If you want to build translation applications, the ManyThings website (http://www.manythings.org/anki/) provides sentence pairs that can be used as training sets. We've provided these pairs for you in the `nlpia` package. In listing 10.8 you can replace `get_data('moviedialog')` with `get_data('deu-eng')` for English-German statement pairs, for example.

Sequence-to-sequence models are also well-suited to text summarization, due to the difference in string length between input and output. In this case, the input to the encoder network is, for example, news articles (or any other length document) and the decoder can be trained to generate a headline or abstract or any other summary sequence associated with the document. Sequence-to-sequence networks can provide

[48] Q&A dataset: https://cs.nyu.edu/~kcho/DMQA/.
[49] List of dialog corpora in the NLPIA package docs: https://github.com/totalgood/nlpia/blob/master/docs/notes/nlp–data.md#dialog-corpora.

more natural-sounding text summaries than summarization methods based on bag-of-words vector statistics. If you're interested in developing such an application, the Kaggle news summary challenge[50] provides a good training set.

Sequence-to-sequence networks aren't limited to natural language applications. Two other applications are automated speech recognition and image captioning. Current, state-of-the-art automated speech recognition systems[51] use sequence-to-sequence networks to turn voice input amplitude sample sequences into the thought vector that a sequence-to-sequence decoder can turn into a text transcription of the speech. The same concept applies to image captioning. The sequence of image pixels (regardless of image resolution) can be used as an input to the encoder, and a decoder can be trained to generate an appropriate description. In fact, you can find a combined application of image captioning and Q&A system called visual question answering at https://vqa.cloudcv.org/.

Summary

- Sequence-to-sequence networks can be built with a modular, reusable encoder-decoder architecture.
- The encoder model generates a thought vector, a dense, fixed-dimension vector representation of the information in a variable-length input sequence.
- A decoder can use thought vectors to predict (generate) output sequences, including the replies of a chatbot.
- Due to the thought vector representation, the input and the output sequence lengths don't have to match.
- Thought vectors can only hold a limited amount of information. If you need a thought vector to encode more complex concepts, the attention mechanism can help selectively encode what is important in the thought vector.

[50] See the web page titled "NEWS SUMMARY: Kaggle" (https://www.kaggle.com/sunnysai12345/news-summary/data).

[51] State of the art speech recognition system: https://arxiv.org/pdf/1610.03022.pdf.

index

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W


manning