16_Modeling_Sequential_Data_Using_Recurrent_Neural_Networks

June 29, 2018

1 Modeling Sequential Data Using Recurrent Neural Networks

In the previous chapter, we focused on **Convolutional Neural Networks (CNNs)** for image classification. In this chapter, we will explore **Recurrent Neural Networks (RNNs)** and see their application in modeling sequencial data and a specific subset of sequential data - time-series data. As an overview, in this chapter, we will cover the following topics:

- Introducing sequential data
- RNNs for modeling sequences
- **Long Short-Term Memory (LSTM)**
- **Truncated Backpropagation Through Time (T-BPTT)**
- Implementing a multilayer RNN for sequence modeling in TensorFlow
- Project one - RNN sentiment analysis of the IMDb movie review dataset
- Project two - RNN character-level language modeling with LSTM cells, using text data from Shakespeare's Hamlet
- Using gradient clipping to avoid exploring gradients.

Since this chapter is the last in our *Python Machine Learning* journey, we will conclude with a summary of what we have learned about RNNs, and an overview of all the machine learning and deep learning topics that led us to RNNs across the journey of the book. We will then sign off by sharing with you links to some of our favorite people and initiatives in this wonderful field so that you can continue your journey into machine learning and deep learning.

2 Introducing sequential data

Let's begin our discussion of RNNs by looking at the nature of sequential data, more commonly known as **sequences**. We will take a look at the unique properties of sequences that make them different from other kind of data. We will then see how we can represent sequential data, and explore the various categories of model ffor sequential data, which are based on the input and output of a model. This will help us explore relationship between RNNs and sequences a little bit later on in the chapter.

2.1 Modeling sequential data - order matters

What makes sequences unique, from other data types, is that elements in a sequence appear in a certain order, and are not independent of each other.

If you recall from Chapter 6, we discussed that typical machine learning algorithms for supervised learning assume that the input data is **Independent and Identically Distributed (IID)**. For example, if we have n data samples, $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$, the order in which we use the data for training our machine learning algorithm does not matter.

However, this assumption is not valid anymore when we deal with sequences - by definition, order matters.

2.2 Representing sequences

We have estabilished that sequences are a nonindependent order in our input data; we next need to find ways to leverage this valuable information in our machine learning model.

Throughout this chapter, we will represent sequences as $(x^{(1)}, x^{(2)}, \ldots, x^{(T)})$. The superscript indices indicate the order of the instances, and the length of the sequence is T. For a sensible example of sequences, consider time-series data, where each sample point $x^{(t)}$ belongs to a particular time t.

The following figure shows an example of time-series data where both x's and y's naturally follow the order according to their time axis; therefore, both x's and y's are sequences:

The standard neural network models that we covered so far, such as MLP and CNNs, are not capable of handling *the order* of input samples. Intuitively, one can say that such models do not have a *memory* of the past seen samples. For instance, the samples are passed through the feedforward and backpropagation steps, and the weights are updated independent of the order in which the sample is processed.

RNNs, by contrast, are designed for modeling sequences and are capable of remembering past information and processing new events accordingly.

2.3 The different categories of sequence modeling

Sequence modeling has many fascinating applications, such as language translation (perhaps from English to German), image captioning, and text generation.

However, we need to understand the different types of sequence modeling tasks to develop an appropriate model. The following figure shows several different relationship categories of input and output data:

So, let's consider the input and output data here. If neither the input or output data represents sequences, then we are dealing with standard data, and we can use any of the previous methods to model such data. But if either the input or output is a sequence, the data will form one of the following three different categories:

- **Many-to-one**: The input data is a sequence, but the output is a fixed-size vector, not a sequence. For example, in sentiment analysis, the input is a text-based and the output is a class label.
- **One-to-many**: The input data is in standard format, not a sequence, but the output is a sequence. An example of this category is image captioning - the input is an image; the output is an English phrase.
- **Many-to-many**: Both the input and output arrays are sequences. This category can be further divided based on whether the input and output are synchronized or not. An example of a **synchronized** many-to-many modeling task is video classification, where each frame in a video is labeled. An example of a **delayed** many-to-many would be translating a language into another. For instance, an entire English sentence must be read and processed by a machine before producing its translation into German.

Now, since we know the categories of sequence modeling, we can move forward to discuss the structure of an RNN.

3 RNNs for modeling sequences

In this section, now that we understand sequences, we can look at the foundations of RNNs. We will start by introducing the typical structure of an RNN, and we will see how the data flows through it with one or more hidden layers. We will then examine how the neuron activations are computed in a typical RNN. This will create a context for us to discuss the common challenges in training RNNs, and explore the modern solution to these challenges - LSTM.

3.1 Understanding the structure and flow of an RNN

Let's start by introducing the architecture of an RNN. The following figure shows a standard feedforward neural network and an RNN, in a side by side for comparison:

Both of these networks have only one hidden layer. In this representation, the units are not displayed, but we assume that the input layer (x), hidden layer (h), and output layer (y) are vectors which contain many units.

This generic RNN architecture could correspond to the two sequence modeling categories where the input is a sequence. Thus, it could be either many-to-many if we consider $y^{(t)}$ as teh final output, or it could be many-to-one if, for example, we only use the last element of $y^{(t)}$ as the final output.

Later, we will see how the output sequence $y^{(t)}$ can be converted into standard, nonsequential output.

In a standard feedforward network, information flows from the input to the hidden layer, and then from the hidden layer to the output layer. On the other hand, in a recurrent network, the hidden layer gets its input from both the input layer and the hidden layer from the previous time step.

The flow of information in adjacent time steps in the hidden layer allows the network to have a memory of past events. This flow of information is usually displayed as a loop, also known as a **recurrent edge** in graph notation, which is how this general architecture got its name.

In the following figure, the single hidden layer network and the multilayer network illustrate two contrasting architectures:

In order to examine the architecture of RNNs and the flow of information, a compact representation with a recurrent edge can be unfolded, which you can see in the preceding figure.

As we know, each hidden unit in a standard neural network receives only one input - the net preactivation associated with the input layer. Now, in contrast, each hidden layer unit in an RNN receives two *distinct* sets of input - the preactivation from the input layer and the activation of the same hidden layer from the previous time step $t-1$.

At the first time step $t = 0$, the hidden units are initialized to zeros or small random values. Then, at a time step where $t > 0$, the hidden units get their input from the data point at the current time $x^{(t)}$ and the previous values of hidden units at $t-1$, indicated as $h^{(t-1)}$.

Similarly, in the case of a multilayer RNN, we can summarize the information flow as follows:

- *layer=1*: Here, the hidden layer is represented as $h_1^{(t)}$ and gets its input from the data point $x^{(t)}$ and the hidden values in the same layer, but the previous time step h_1^{t-1}.

- *layer=2*: The second hidden layer, $h_2^{(t)}$ receives its inputs from the hidden units from the layer below at the current time step ($h_1^{(t)}$) and its own hidden values from the previous time step $h_2^{(t-1)}$.

3.2 Computing activations in an RNN

Now that we understand the structure and general flow of information in an RNN, let's get more specific and compute the actual activations of the hidden layers as well as the output layer. For simplicity, we will consider just a single hidden layer; however, the same concept applies to multilayer RNNs.

Each directed edge (the connections between boxes) in the representation of an RNN that we just looked at is associated with a weight matrix. Those weights do not depend on time t; therefore, they are shared across the time axis. The different weight matrices in a single layer RNN are as follows:

- W_{xh}: The weight matrix between the input $x^{(t)}$ and the hidden layer h.
- W_{hh}: The weight matrix associated with the recurrent edge.
- W_{hy}: The weight matrix between the hidden layer and output layer.

You can see these weights matrices in the following figure:

In certain implementations, you may observe that weight matrices W_{xh} and W_{hh} are concatenated to a combined matrix $W_h = [W_{xh}; W_{hh}]$. Later on, we will make use of this notation as well.

Computing the activations is very similar to standard multilayer perceptrons and other types of feedforward neural networks. For the hidden layer, the net input Z_h (preactivation) is computed through a linear combination. That is, we compute the sum of the multiplications of the weight matrices with the corresponding vectors and add the bias unit - $Z_h = W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h$. Then, the activations of the hidden units at the time step t are calculated as follows:

$$h^{(t)} = \phi(Z_h^{(t)}) = \phi_h(W_{xh}x^{(t)} + W)hhh^{(t-1)} + b_h)$$

Here, b_h is the bias vector for the hidden units and ϕ_h is the activation function of the hidden layer.

In case you want to use the concatenated weight matrix $W_h = [W_{xh}; W_{hh}]$, the formula for computing hidden units will change as follows:

$h^{(t)} = \phi_h \left([W_{xh}; W_{hh}][x^{(t)}; h^{(t-1)}]^{-1} + b_h \right)$

Once the activations of hidden units at the current time step are computed, then the activations of output units will be computed as follows:

$$y^{(t)} = \phi_y(w_{hy}h^{(t)} + b_y)$$

To help clarify this further, the following figure shows the process of computing these activations with both formulations:

Training RNNs using BPTT

The learning algorithm for RNNs was introduced in 1990s *Backpropagation Through Time: What it Does and How to Do It*.

The derivation of the gradients might be a bit complicated, but the basic idea is that the overall loss L is the sum of all the loss functions at time $t = 1$ to $t = T$:

$$L = \sum_{t=1}^{T} L^{(t)}$$

3.3 The challenges of learning long-range interactions

Backpropagation through time, or BPTT, which we briefly mentioned in the previous information box, introduces some challenges. Because of the multiplicative factor $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ in the computing gradients of a loss function, the so-called **vanishing** or **exploding** gradient problem arises. This problem is explained through the examples in the following figure, which shows an RNN with only one hidden unit for simplicity:

Basically, $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ has $t-k$ multiplications; therefore, multiplying the w weight $t-k$ times results in a factor w^{t-k}. As a result, if $|w| < 1$, this factor becomes very small when $t-k$ is large. On the other hand, if the weight of the recurrent edge is $|w| > 1$, then w^{t-k} becomes very large when $t-k$ is large. Note that large $t-k$ refers to long-range dependencies.

Intuitively, we can see that a naive solution to avoid vanishing or exploring gradient can be accomplished by ensuring $|w| = 1$.

In practice, there are two solutions to this problem: * Truncated backpropagation through time (TBPTT). * Long short-term memory (LSTM).

TBPTT clips the gradients above a given threshold. While TBPTT can solve the exploding gradient problem, the truncation limits the number of steps that the gradient can effectively flow back and properly update the weights.

On the other hand, LSTM, designed in 1997 by Hochreiter and Schmidhuber, has been more successful in modeling long-range sequences by overcoming the vanishing gradient problem. Let's discuss LSTM in more detail.

3.4 LSTM units

LSTM were first introduced to overcome the vanishing gradient problem. The building block of an LSTM is a **memory cell**, which essentially represents the hidden layer.

In each memory cell, there is a recurrent edge that has the desirable weight $w = 1$, as we discussed previously, to overcome the vanishing and exploding gradient problems. The values associated with this recurrent edge is called **cell state**. The unfolded structure of a modern LSTM cell is shown in the following figure:

Notice that the cell state from the previous time step, $C^{(t-1)}$, is modified to get the cell state at the current time step, $C^{(t)}$, without being multiplied directly with any weight factor.

The flow of information in this memory cell is controlled by some units of computation that we will describe here. In the previous figure, $\odot$ refers to the **element-wise product** (element-wise multiplication) and $\oplus$ means **element-wise summation** (element-wise addition). Furthermore, $x^{(t)}$ refers to the input data at time t, and $h^{(t-1)}$ indicates the hidden units at time $t-1$.

Four boxes are indicated with an activation function, either the sigmoid function (σ) or hyperbolic tangent (tanh), and a set of weights; these boxes apply linear combination by performing matrix-vector multiplications on their input. These units of computation with sigmoid activation functions, whose output units are passed through $\odot$, are called **gates**.

In an LSTM cell, there are three different types of gates, known as the forget gate, the input gate, and the output gate:

- The **forget gate (f_t)** allows the memory cell to reset the cell state without growing indefinitely. In fact, the forget gate decides which information is allowed to go through and which information to suppress. Now, f_t is computed as follows: $f_t = \sigma(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f)$. Note that the forget gate was not part of the original LSTM cell; it was added a few years later to improve the original model.
- The **input gate (i_t)** and input nodes (g_t) are responsible for updating the cell state. They are computed as follows: $i_t = \sigma(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i)$ and $g_i = tanh(W_{xg}x^{(t)} + W_{hg}h^{(t-1)} + b_g)$. The cell state at time t is computed as follows: $C^{(t)} = (C^{t-1} \odot f_t) \oplus (i_t \odot g_t)$.
- The **output gate (o_t)** decides how to update the values of hidden units: $o_t = \sigma(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o)$. Given this, the hidden units at the current time step are computed as follows: $h^{(t)} = o_t \odot tanh(C^{(t)})$.

The structure of an LSTM cell and its underlying computations might seem too complex. However, the good news is that TensorFlow has already implemented everything in wrapper functions that allows us to define our LSTM cell easily. We will see the real application of LSTMs in action when we use TensorFlow later in this chapter.

We have introduced LSTMs in this section, which provide a basic approach for modeling long-range dependencies in sequences. Yet, it is important to note that there are many variations of LSTMs described in literature.

Also, worth noting is a more recent approach, called **Gated Recurrent Unit (GRU)**, which was proposed in 2014. GRUs have a simpler architecture than LSTMs; therefore, they are computationally more efficient while their performance in some tasks, such as polyphonic music modeling, is comparable to LSTMs.

4 Implementing a multilayer RNN for sequence modeling in TensorFlow

Now that we introduced the underlying theory behind RNNs, we are ready to move on to the more practical part to implement RNNs in TensorFlow. During the rest of this chapter, we will apply RNNs to two common problems tasks:

1. Sentiment analysis
2. Language modeling

These two projects, which we will build together in the following pages, are both fascinating but also quite involved. Thus, instead of providing all the code all at once, we will break the implementation up into several steps and discuss the code in detail.

Note, before we start coding in this chapter, that since we are using a very modern build of TensorFlow, we will be using code from the *contrib* submodule of TensorFlow's Python API, in the latest version of TensorFlow (1.3.0) form August 2017. These *contrib* functions and classes, as well as their documentation references used in this chapter, may change in the future versions of TensorFlow, or they may be integrated into the *tf.nn* submodule. We therefore advise you to keep and eye on the TensorFlow API documentation to be updated with the latest version details, in particular, if you have any problems using the *tf.contrib* code described in this chapter.

5 Project one - performing sentiment analysis of IMDb movie reviews using multilayer RNNs

You may recall from Chapter 8 that sentiment analysis is concerned with analyzing the expressed opinion of a sentence or text document. In this section and the following subsections, we will implement a multilayer RNN for sentiment analysis using a many-to-one architecture.

In the next section, we will implement many-to-many RNN for an application language modeling. While the chosen examples are purposefully simple to introduce the main concepts of RNNs, language modeling has a wide range of interesting applications such as building chatbot - giving computers the ability to directly talk and interact with a human.

5.1 Preparing the data

In the preprocessing steps in Chapter 8, we created a clean dataset named *movie_data.csv*, which we will use again now. So, first let's import the necessary modules and read the data into a *DataFrame* pandas, as follows:

```
In [1]: import pyprind
        import pandas as pd
        from string import punctuation
        import re
        import numpy as np

        df = pd.read_csv('movie_data.csv', encoding='utf-8')
```

Recall that this *df* data frame has two columns, namely *'review'* and *'sentiment'*, where *'review'* contains the text of movie reviews and *'sentiment'* contains the 0 or 1 labels. The text component of these movie reviews are sequences of words; therefore, we want to build an RNN model to process the words in each sequence, and at the end, classify the entire sequence to 0 or 1 classes.

To prepare the data for input to a neural network, we need to encode it into numeric values. To do this, we first find the unique words in the entire dataset, which can be using sets in Python. However, I found that using sets for finding unique words in such a large dataset is not efficient. A more efficient way is to use *Counter* from the collections package.

In the following code, we will define a *counts* objects from the *Counter* class that collects the counts of occurrence of each unique word in the text. Note that in this particular application (and in contrast to the bag-of-words models), we are only interested in the set of unique words and won't require the word counts, which are created as a side product.

Then, we create a mapping in the form of a dictionary that maps each unique words, in our dataset, to a unique integer number. We call this dictionary *word_to_int*, which can be used to convert the entire text of a review into a list of numbers. The unique words are sorted based on their counts, but any arbitrary order can be used without affecting the final results. This process of converting a text into a list of integers is performed using the following code:

```
In [2]: # Preprocessing the data:
        # Separate words and
        # count each word's occurrence

        from collections import Counter
```

```
counts = Counter()
pbar = pyprind.ProgBar(len(df['review']), title='Counting words occurrences')
for i, review in enumerate(df['review']):
    text = ''.join([c if c not in punctuation else ' '+c+' ' for c in review]).lower()
    df.loc[i, 'review'] = text
    pbar.update()
    counts.update(text.split())

# Create a mapping
# Map each unique word to an integer
word_counts = sorted(counts, key=counts.get, reverse=True)
print(word_counts[:5])
word_to_int = {word: ii for ii, word in enumerate(word_counts, 1)}

mapped_reviews = []
pbar = pyprind.ProgBar(len(df['review']), title='Map reviews to ints')
for review in df['review']:
    mapped_reviews.append([word_to_int[word] for word in review.split()])
    pbar.update()
```

```
Counting words occurrences
0% [##############################] 100% | ETA: 00:00:00
Total time elapsed: 00:03:35
Map reviews to ints


['the', '.', ',', 'and', 'a']


0% [##############################] 100% | ETA: 00:00:00
Total time elapsed: 00:00:02
```

So far, we have converted sequences of words into sequences of integers. However, there is one issue that we still need to solve - the sequences currently have different lenghts. In order to generate input data that is compatible with our RNNs architecture, we will need to make sure that all the sequences have the same length.

For this purpose, we define a parameter called *sequence_length* that we set to 200. Sequences that have fewer than 200 words will be left-padded with zeros. Vice versa, sequences that are longer than 200 words are cut such that only the last 200 corresponding words will be used. We can implement this preprocessing step in two steps:

1. Create a matrix of zeros, where each row correspond to a sequence of size 200.
2. Fill the index of words in each sequence from the right-hand side of the matrix. Thus, if a sequence has a length of 150, the first 50 elements of the corresponding row will stay zero.

These two steps are shown in the following figure, for a small example with eight sequences of sizes 4, 12, 8, 11, 7, 3, 10, and 13:

Note that *sequence_length* is, in fact, a hyperparameter and can be used for optimal performance. Due to page limitations, we did not optimize this hyperparameter further, but we encourage you to try this with different values for *sequence_length*, such as 50, 100, 200, 250, and 300.

Check out the following code for the implementation for these steps to create sequences of the same length:

```
In [3]: # Define same-length sequences
        # if sequence length < 200: left-pad with zeros
        # if sequence length > 200: use the last 200 elements

        sequence_length = 200 # (Known as T in our RNN formulas)
        sequences = np.zeros((len(mapped_reviews), sequence_length), dtype=int)

        for i, row in enumerate(mapped_reviews):
            review_arr = np.array(row)
            sequences[i, -len(row):] = review_arr[-sequence_length:]
```

After we preprocess the dataset, we can proceed with splitting the data into separate training and test sets. Since the dataset was already shuffled, we can simply take the first half of the dataset for training and the second half for testing, as follows:

```
In [4]: X_train = sequences[:25000, :]
        y_train = df.loc[:25000, 'sentiment'].values
        X_test = sequences[25000:, :]
        y_test = df.loc[25000:, 'sentiment'].values
```

Now if we want to separate the dataset for cross-validation, we can further split the second half of the data further to generate a smaller test set and a validation set for hyperparameter optimization.

Finally, we define a helper function that breaks a given dataset (which could be a training set or test set) into chunks and returns a generator to iterate through these chunks (also known as **mini-batches**):

```
In [5]: np.random.seed(123) # for reproducibility

        # Define a function to generate mini-batches:
        def create_batch_generator(x, y=None, batch_size=64):
            n_batches = len(x)//batch_size
            x = x[:n_batches*batch_size]
            if y is not None:
                y = y[:n_batches*batch_size]
            for ii in range(0, len(x), batch_size):
                if y is not None:
                    yield x[ii:ii+batch_size], y[ii:ii+batch_size]
                else:
                    yield x[ii:ii+batch_size]
```

Using generators, as we have done in this code, is a very useful technique for handling memory limitations. This is the recommended approach for splitting the dataset into mini-batches for training a neural network, rather than creating all the data splits upfront and keeping them in memory during training.

5.2 Embedding

During the data preparation in the previous step, we generated sequences of the same length. The elements of these sequences were integer numbers that corresponded to the *indices* of unique words.

These word indices can be converted into input features in several different ways. One naive way is to apply one-hot encoding to convert indices into vectors of zeros and ones. Then, each word will be mapped to a vector whose size is the number of unique words in the entire dataset. Given that the number of unique words (the size of the vocabulary) can be in the order of 20,000, which will also be the number of our input features, a model trained on such features may suffer from the **curse of dimensionality**. Furthermore, these features are very sparse, since all are zero except one.

A more elegant way is to map each word to a vector of fixed size with real-valued elements (not necessarily integers). In contrast to the one-hot encoded vectors, we can use finite-sized vectors to represent an infinite number of real numbers (in theory, we can extract infinite real numbers from a given interval, for example [-1, 1]).

This is the idea behind the so-called **embedding**, which is a feature-learning technique that we can utilize here to automatically learn the salient features to represent the words in our dataset. Given the number of unique words *unique_words*, we can choose the size of the embedding vectors to be much smaller than the number of unique words (*embedding_size* << *unique_words*) to represent the entire vocabulary as input features.

The advantages of embedding over one-hot encoding are as follows:

- A reduction in the dimensionality of the feature space to decrease the effect of effect of the curse of dimensionality.
- The extraction of salient features since the embedding layer in a neural network is trainable.

The following schematic representation shows how embedding works by mapping vocabulary indices to a trainable embedding matrix:

TensorFlow implements an efficient function, *tf.nn.embedding_lookup*, that maps each integer that corresponds to a unique word, to a row of this trainable matrix. For example, integer 1 is mapped to the first row, integer 2 is mapped to the second row, and so on. Then, given a sequence of integers, such as <0, 5, 3, 4, 19, 2, ...>, we need to look up the corresponding rows for each element in this sequence.

Now, let's see how we can create an embedding layer in practice. If we have *tf_x* as the input layer where the corresponding vocabulary indices are fed with type *tf.int32*, then creating an embedding layer can be done in two steps, as follows:

1. We start by creating a matrix of size $[n_words \times embedding_size]$ as a tensor variable, which we call *embedding* and we initialize its elements randomly with floats between $[-1, 1]$.
2. Then, we use the *tf.nn.embedding_lookup* function to look up the row in the embedding matrix associated with each element of *tf_x*.

As you may have observed in these steps, to create an embedding layer, the *tf.nn.embedding_lookup* function requires two arguments, the embedding tensor and the lookup IDs.

The *tf.nn.embedding_lookup* function has a few optional arguments that allow you to tweak the behavior of the embedding layer, such as applying L2 normalization.

5.3 Building an RNN model

Now we are ready to build an RNN model. We will implement a *SentimentRNN* class that has the following methods:

- A constructor to set all the model parameters and then create a computation graph and call the *self.build* method to build the multilayer RNN model.
- A *build* method that declares three placeholders for input data, input labels, and the keep-probability for the dropout configuration of the hidden layer. After declaring these, it creates an embedding layer, and builds the multilayer RNN using the embedded representation as input.
- A *train* method that creates a TensorFlow session for launching the computation graph, iterates through the mini-batches of data, and runs for a fixed number of epochs, to minimize the cost function defined in the graph. This method also saves the model after 10 epochs for checkpointing.
- A *predict* method that creates a new session, restores the last checkpoint saved during the training process, and carries out the predictions for the test data.

In the following code, we will see the implementations of this class and its methods broken into separate code sections.

5.4 The SentimentRNN class constructor

Let's start with the constructor of our *SentimentRNN* class, which we will code as follows:

```
In [6]: import tensorflow as tf

        class SentimentRNN(object):

            def __init__(self, n_words, seq_len=200,
                         lstm_size=256, num_layers=1, batch_size=64,
                         learning_rate=0.0001, embed_size=200):
                self.n_words = n_words
                self.seq_len = seq_len
                self.lstm_size = lstm_size # number of hidden units
                self.num_layers = num_layers
                self.batch_size = batch_size
                self.learning_rate = learning_rate
                self.embed_size = embed_size

                self.g = tf.Graph()
                with self.g.as_default():
                    tf.set_random_seed(123)
                    self.build()
                    self.saver = tf.train.Saver()
                    self.init_op = tf.global_variables_initializer()

            def build(self):
                ## Define the placeholders
                tf_x = tf.placeholder(tf.int32, shape=(self.batch_size, self.seq_len), name='tf_
                tf_y = tf.placeholder(tf.float32, shape=(self.batch_size), name='tf_y')
                tf_keepprob = tf.placeholder(tf.float32, name='tf_keepprob')

                ## Create the embedding layer
                embedding = tf.Variable(tf.random_uniform((self.n_words, self.embed_size), minva
                embed_x = tf.nn.embedding_lookup(embedding, tf_x, name='embeded_x')

                ## Define LSMT cell and stack them together
                cells = tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.DropoutWrapper(tf.contrib.rn

                ## Define the initial state
                self.initial_state = cells.zero_state(self.batch_size, tf.float32)
                print(' << initial state >> ', self.initial_state)

                lstm_outputs, self.final_state = tf.nn.dynamic_rnn(cells, embed_x, initial_state

                # Note: lstm_outputs shape:
                #[batch_size, max_time, cells.output_size]
                print('\n << lstm_output >> ', lstm_outputs)
                print('\n << final state >> ', self.final_state)

                logits = tf.layers.dense(inputs=lstm_outputs[:, -1], units=1, activation=None, n

                logits = tf.squeeze(logits, name='logits_squeezed')
                print('\n << logits >> ', logits)

                y_proba = tf.nn.sigmoid(logits, name='probabilities')
                predictions = {
                    'probabilities': y_proba,
                    'labels': tf.cast(tf.round(y_proba), tf.int32, name='labels')
                }
                print('\n << predictions >> ', predictions)

                ## Define the cost function
                cost = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf_y, logit

                ## Define the optimizer
                optimizer = tf.train.AdamOptimizer(self.learning_rate)
                train_op = optimizer.minimize(cost, name='train_op')

            def train(self, X_train, y_train, num_epochs):
                with tf.Session(graph=self.g) as sess:
                    sess.run(self.init_op)
                    iteration = 1
                    for epoch in range(num_epochs):
                        state = sess.run(self.initial_state)
                        for batch_x, batch_y in create_batch_generator(X_train, y_train, self.ba
                            feed = {'tf_x:0': batch_x, 'tf_y:0': batch_y,
                                    'tf_keepprob:0': 0.5, self.initial_state: state}
                            loss, _, state = sess.run(['cost:0', 'train_op', self.final_state],

                            if iteration % 20 == 0:
                                print('Epoch %d/%d Iteration: %d | Train loss: %.5f' % (epoch+1,

                            iteration += 1
                        if (epoch+1)%10 == 0:
                            self.saver.save(sess, 'model/sentiment-%d.ckpt' % epoch)

            def predict(self, X_data, return_proba=False):
                preds = []
                with tf.Session(graph=self.g) as sess:
                    self.saver.restore(sess, tf.train.latest_checkpoint('./model/'))
                    test_state = sess.run(self.initial_state)
                    for ii, batch_x in enumerate(create_batch_generator(X_data, None, batch_size
                        feed = {'tf_x:0': batch_x, 'tf_keepprob:0': 1.0, self.initial_state: tes
                        if return_proba:
                            pred, test_state = sess.run(['probabilities:0', self.final_state], f
                        else:
                            pred, test_state = sess.run(['labels:0', self.final_state], feed_dic

                        preds.append(pred)

                return np.concatenate(preds)
```

Here, the *n_words* parameter must be set equal to the number of unique words (plus 1 since we use zero to fill sequences whose size is less than 200) and it is used while creating the embedding layer along with the *embed_size* hyperparameter. Meanwhile, the *seq_len* variable must be set according to the length of the sequences that were created in the preprocessing steps we went through previously. Note that *lstm_size* is another hyperparameter that we have used here, and it determines that number of hidden units in each RNN layer.

5.5 The build method

Next, let's discuss the *build* method for our *SentimentRNN* class. This is the longest and most critical method in our sequence, so we will be going through it in plenty of detail. First, we will look at the code in full, so we can see everything together, and then we will analyze each of its main parts:

So first of all in our *build* method here, we created three placeholders, namely *tf_x*, *tf_y*, and *tf_keepprob*, which we need for feeding the input data. Then we added the embedding layer, which builds the embedded representation *embed_x*, as we discussed earlier.

Next, in our *build* method, we built the RNN network with LSTM cells. We did this in three steps:

1. First, we defined the multilayer RNN cells.

2. Next, we defined the initial state of these cells.
3. Finally, we created an RNN specified by the RNN cells and their initial states.

Let's break these three steps out in detail in the following three sections, so we can examine in depth how we built the RNN network in our *build* method.

5.5.1 Step 1 - defining multilayer RNN cells

To examine how we encoded our *build* method to build the RNN network, the first step was to define our multilayer RNN cells.

Fortunately, TensorFlow has a very nice wrapper class to define LSTM cells - the *BasicLSTMCell* class - which can be stacked together to form a multilayer RNN using the *MultiRNNCell* wrapper class. The process of stacking RNN cells with a dropout has three nested steps; these three nested steps can be described from inside out as follows:

1. First, create the RNN cells using *tf.contrib.rnn.BasicLSTMCell*.
2. Apply the dropout to the RNN cells using *tf.contrib.rnn.DropoutWrapper*.
3. Make a list of such cells according to the desired number of RNN layers and pass this list to *tf.contrib.rnn.MultiRNNCell*.

In our *build* method code, this list is created using Python list comprehension. Note that for a single layer, this list has only once cell.

5.5.2 Step 2 - defining the initial states for the RNN cells

The second step that our *build* method takes to build the RNN network was to define the initial states for the RNN cells.

You will recall from the architecture of LSTM cells, there are three types of inputs in an LSTM cell - input data $x^{(t)}$, activations of hidden units from the previous time step $h^{(t-1)}$, and the cell state from the previous time step $C^{(t-1)}$.

So, in our *build* method implementation, $x^{(t)}$ is the embedded *embed_x* data tensor. However, when we evaluate the *cells*, we also need to specify the previous state of the cells. So, when we start processing a new input sequence, we initialize the cell states to zero state; then after each time step, we need to store the updated state of the cells to use for the next time step.

Once our multilayer RNN object is defined (*cells* in our implementation), we define its initial state in our *build* method using the *cells.zero_state* method.

5.5.3 Step 3 - creating the RNN using the RNN cells and their states

The third step to creating the RNN in our *build* method, used the *tf.nn.dynamic_rnn* function to pull together all our components.

The *tf.dynamic_rnn* function therefore pulls the embedded data, the RNN cells, and their initial states, and creates a pipeline for them according to the unrolled architecture of LSTM cells.

The *tf.dynamic_rnn* function returns a tuple containing the activations of the RNN cells, *outputs*; and their final states, *state*. The output is a three-dimensional tensor with this shape *(batch_size, num_steps, lstm_size)*. We pass *outputs* to a fully connected layer to get *logits* and we store the final state to use as the initial state of the next mini-batch of data.

Finally, in our *build* method, after setting up the RNN components of the network, the cost function and optimization schemes can be defined like any other neural network.

5.6 The train method

The next method in our *SentimentRNN* class is *train*. This method is quite similar to the train methods we created in Chapters 14 and 15, except that we have an additional tensor, *state*, that we feed into our network.

The following code shows the implementation of the *train* method:

In this implementation of our *train* method, at the beginning of each epoch, we start from the zero states of RNN cells as our current state. Running each mini-batch of data is performed by feeding the current state along with the data *batch_x* and their labels *batch_y*. Upon finishing the executing of a mini-batch, we update the state to be the final state, which is returned by the *tf.nn.dynamic_rnn* function. This updated state will be used toward executing of the next mini-batch. This process is repeated and the current state is updated throughout the epoch.

5.7 The predict method

Finally, the last method in our *SentimentRNN* class is the *predict* method, which keeps updating the current state similar to the *train* method, shown in the following code:

5.8 Instantiating the SentimentRNN class

We have now coded and examined all four parts of our *SentimentRNN* class, which were the class constructor, the *build* mehtod, the *train* method, and *predict* method.

We are now ready to create an object of the class *SentimentRNN*, with parameters as follows:

```
In [7]: n_words = max(list(word_to_int.values())) + 1

        rnn = SentimentRNN(n_words=n_words,
                           seq_len=sequence_length,
                           embed_size=256,
                           lstm_size=128,
                           num_layers=1,
                           batch_size=100,
                           learning_rate=0.001)
```

```
WARNING:tensorflow:From /usr/lib/python3.6/site-packages/tensorflow/contrib/learn/python/learn/d
Instructions for updating:
Use the retry module or similar alternatives.
 << initial state >>  (LSTMStateTuple(c=<tf.Tensor 'MultiRNNCellZeroState/DropoutWrapperZeroStat

 << lstm_output >>  Tensor("rnn/transpose_1:0", shape=(100, 200, 128), dtype=float32)

 << final state >>  (LSTMStateTuple(c=<tf.Tensor 'rnn/while/Exit_3:0' shape=(100, 128) dtype=flo

 << logits >>  Tensor("logits_squeezed:0", shape=(100,), dtype=float32)

 << predictions >>  {'probabilities': <tf.Tensor 'probabilities:0' shape=(100,) dtype=float32>,
```

Notice here that we use *num_layers=1* to use a single RNN layer. Although our implementation allows us to create a multilayer RNNs, by setting *num_layers* greater than 1. Here we should consider the small size of our dataset, and that a single RNN layer may generalize better to unseen data, since it is less likely to overfit the training data.

5.9 Training and optimizing the sentiment analysis RNN model

Next, we can train the RNN model by calling the *rnn.train* function. In the following code, we train the model for 40 epochs using the input from *X_train* and the corresponding class labels stored in *y_train*:

```
In [8]: rnn.train(X_train, y_train, num_epochs=40)
```

```
Epoch 1/40 Iteration: 20 | Train loss: 0.69958
Epoch 1/40 Iteration: 40 | Train loss: 0.62890
Epoch 1/40 Iteration: 60 | Train loss: 0.64267
Epoch 1/40 Iteration: 80 | Train loss: 0.57872
Epoch 1/40 Iteration: 100 | Train loss: 0.56441
Epoch 1/40 Iteration: 120 | Train loss: 0.56481
Epoch 1/40 Iteration: 140 | Train loss: 0.50480
Epoch 1/40 Iteration: 160 | Train loss: 0.51108
Epoch 1/40 Iteration: 180 | Train loss: 0.47452
Epoch 1/40 Iteration: 200 | Train loss: 0.42639
Epoch 1/40 Iteration: 220 | Train loss: 0.41352
Epoch 1/40 Iteration: 240 | Train loss: 0.44784
Epoch 2/40 Iteration: 260 | Train loss: 0.39764
Epoch 2/40 Iteration: 280 | Train loss: 0.37825
Epoch 2/40 Iteration: 300 | Train loss: 0.30951
Epoch 2/40 Iteration: 320 | Train loss: 0.25323
Epoch 2/40 Iteration: 340 | Train loss: 0.33181
Epoch 2/40 Iteration: 360 | Train loss: 0.25260
Epoch 2/40 Iteration: 380 | Train loss: 0.37994
Epoch 2/40 Iteration: 400 | Train loss: 0.32391
Epoch 2/40 Iteration: 420 | Train loss: 0.28590
Epoch 2/40 Iteration: 440 | Train loss: 0.28629
Epoch 2/40 Iteration: 460 | Train loss: 0.49977
Epoch 2/40 Iteration: 480 | Train loss: 0.26288
Epoch 2/40 Iteration: 500 | Train loss: 0.24810
Epoch 3/40 Iteration: 520 | Train loss: 0.40307
Epoch 3/40 Iteration: 540 | Train loss: 0.25136
Epoch 3/40 Iteration: 560 | Train loss: 0.41164
Epoch 3/40 Iteration: 580 | Train loss: 0.27206
Epoch 3/40 Iteration: 600 | Train loss: 0.22466
Epoch 3/40 Iteration: 620 | Train loss: 0.26403
Epoch 3/40 Iteration: 640 | Train loss: 0.19858
Epoch 3/40 Iteration: 660 | Train loss: 0.18231
Epoch 3/40 Iteration: 680 | Train loss: 0.29750
Epoch 3/40 Iteration: 700 | Train loss: 0.16059
Epoch 3/40 Iteration: 720 | Train loss: 0.24814
Epoch 3/40 Iteration: 740 | Train loss: 0.26981
Epoch 4/40 Iteration: 760 | Train loss: 0.22786
Epoch 4/40 Iteration: 780 | Train loss: 0.22482
Epoch 4/40 Iteration: 800 | Train loss: 0.11130
Epoch 4/40 Iteration: 820 | Train loss: 0.31478
Epoch 4/40 Iteration: 840 | Train loss: 0.19561
Epoch 4/40 Iteration: 860 | Train loss: 0.14709
Epoch 4/40 Iteration: 880 | Train loss: 0.24212
Epoch 4/40 Iteration: 900 | Train loss: 0.17968
Epoch 4/40 Iteration: 920 | Train loss: 0.20790
Epoch 4/40 Iteration: 940 | Train loss: 0.15241
Epoch 4/40 Iteration: 960 | Train loss: 0.15897
Epoch 4/40 Iteration: 980 | Train loss: 0.15400
Epoch 4/40 Iteration: 1000 | Train loss: 0.31969
Epoch 5/40 Iteration: 1020 | Train loss: 0.14744
Epoch 5/40 Iteration: 1040 | Train loss: 0.11334
Epoch 5/40 Iteration: 1060 | Train loss: 0.20580
Epoch 5/40 Iteration: 1080 | Train loss: 0.14594
Epoch 5/40 Iteration: 1100 | Train loss: 0.06017
Epoch 5/40 Iteration: 1120 | Train loss: 0.19716
Epoch 5/40 Iteration: 1140 | Train loss: 0.12153
Epoch 5/40 Iteration: 1160 | Train loss: 0.07483
Epoch 5/40 Iteration: 1180 | Train loss: 0.11106
Epoch 5/40 Iteration: 1200 | Train loss: 0.07324
Epoch 5/40 Iteration: 1220 | Train loss: 0.09906
Epoch 5/40 Iteration: 1240 | Train loss: 0.19839
Epoch 6/40 Iteration: 1260 | Train loss: 0.07519
Epoch 6/40 Iteration: 1280 | Train loss: 0.02650
Epoch 6/40 Iteration: 1300 | Train loss: 0.07117
Epoch 6/40 Iteration: 1320 | Train loss: 0.09500
Epoch 6/40 Iteration: 1340 | Train loss: 0.05786
Epoch 6/40 Iteration: 1360 | Train loss: 0.08426
Epoch 6/40 Iteration: 1380 | Train loss: 0.08544
Epoch 6/40 Iteration: 1400 | Train loss: 0.11900
Epoch 6/40 Iteration: 1420 | Train loss: 0.09035
Epoch 6/40 Iteration: 1440 | Train loss: 0.12220
Epoch 6/40 Iteration: 1460 | Train loss: 0.32966
Epoch 6/40 Iteration: 1480 | Train loss: 0.13025
Epoch 6/40 Iteration: 1500 | Train loss: 0.10458
Epoch 7/40 Iteration: 1520 | Train loss: 0.11861
Epoch 7/40 Iteration: 1540 | Train loss: 0.07473
Epoch 7/40 Iteration: 1560 | Train loss: 0.20360
Epoch 7/40 Iteration: 1580 | Train loss: 0.05547
Epoch 7/40 Iteration: 1600 | Train loss: 0.04844
Epoch 7/40 Iteration: 1620 | Train loss: 0.12609
Epoch 7/40 Iteration: 1640 | Train loss: 0.01784
Epoch 7/40 Iteration: 1660 | Train loss: 0.02601
Epoch 7/40 Iteration: 1680 | Train loss: 0.02967
Epoch 7/40 Iteration: 1700 | Train loss: 0.02946
Epoch 7/40 Iteration: 1720 | Train loss: 0.05913
Epoch 7/40 Iteration: 1740 | Train loss: 0.05434
Epoch 8/40 Iteration: 1760 | Train loss: 0.02269
Epoch 8/40 Iteration: 1780 | Train loss: 0.02750
Epoch 8/40 Iteration: 1800 | Train loss: 0.02006
Epoch 8/40 Iteration: 1820 | Train loss: 0.02765
Epoch 8/40 Iteration: 1840 | Train loss: 0.01369
Epoch 8/40 Iteration: 1860 | Train loss: 0.03151
Epoch 8/40 Iteration: 1880 | Train loss: 0.08518
Epoch 8/40 Iteration: 1900 | Train loss: 0.06939
Epoch 8/40 Iteration: 1920 | Train loss: 0.08638
Epoch 8/40 Iteration: 1940 | Train loss: 0.04884
Epoch 8/40 Iteration: 1960 | Train loss: 0.07556
Epoch 8/40 Iteration: 1980 | Train loss: 0.14354
Epoch 8/40 Iteration: 2000 | Train loss: 0.04755
Epoch 9/40 Iteration: 2020 | Train loss: 0.05910
Epoch 9/40 Iteration: 2040 | Train loss: 0.03710
Epoch 9/40 Iteration: 2060 | Train loss: 0.06637
Epoch 9/40 Iteration: 2080 | Train loss: 0.09210
Epoch 9/40 Iteration: 2100 | Train loss: 0.03585
Epoch 9/40 Iteration: 2120 | Train loss: 0.08653
Epoch 9/40 Iteration: 2140 | Train loss: 0.12724
Epoch 9/40 Iteration: 2160 | Train loss: 0.02180
Epoch 9/40 Iteration: 2180 | Train loss: 0.04479
Epoch 9/40 Iteration: 2200 | Train loss: 0.02537
Epoch 9/40 Iteration: 2220 | Train loss: 0.06662
Epoch 9/40 Iteration: 2240 | Train loss: 0.04517
Epoch 10/40 Iteration: 2260 | Train loss: 0.01583
Epoch 10/40 Iteration: 2280 | Train loss: 0.01302
Epoch 10/40 Iteration: 2300 | Train loss: 0.01250
Epoch 10/40 Iteration: 2320 | Train loss: 0.03017
Epoch 10/40 Iteration: 2340 | Train loss: 0.01923
Epoch 10/40 Iteration: 2360 | Train loss: 0.00690
Epoch 10/40 Iteration: 2380 | Train loss: 0.03661
Epoch 10/40 Iteration: 2400 | Train loss: 0.01338
Epoch 10/40 Iteration: 2420 | Train loss: 0.09246
Epoch 10/40 Iteration: 2440 | Train loss: 0.08763
Epoch 10/40 Iteration: 2460 | Train loss: 0.03130
Epoch 10/40 Iteration: 2480 | Train loss: 0.04457
Epoch 10/40 Iteration: 2500 | Train loss: 0.02749
Epoch 11/40 Iteration: 2520 | Train loss: 0.05788
Epoch 11/40 Iteration: 2540 | Train loss: 0.01309
Epoch 11/40 Iteration: 2560 | Train loss: 0.02433
Epoch 11/40 Iteration: 2580 | Train loss: 0.01018
Epoch 11/40 Iteration: 2600 | Train loss: 0.02860
Epoch 11/40 Iteration: 2620 | Train loss: 0.04188
Epoch 11/40 Iteration: 2640 | Train loss: 0.00633
Epoch 11/40 Iteration: 2660 | Train loss: 0.01208
Epoch 11/40 Iteration: 2680 | Train loss: 0.02004
Epoch 11/40 Iteration: 2700 | Train loss: 0.00899
Epoch 11/40 Iteration: 2720 | Train loss: 0.01879
Epoch 11/40 Iteration: 2740 | Train loss: 0.05585
Epoch 12/40 Iteration: 2760 | Train loss: 0.15768
Epoch 12/40 Iteration: 2780 | Train loss: 0.00239
Epoch 12/40 Iteration: 2800 | Train loss: 0.04917
Epoch 12/40 Iteration: 2820 | Train loss: 0.05850
Epoch 12/40 Iteration: 2840 | Train loss: 0.01792
Epoch 12/40 Iteration: 2860 | Train loss: 0.00755
Epoch 12/40 Iteration: 2880 | Train loss: 0.01838
Epoch 12/40 Iteration: 2900 | Train loss: 0.02314
Epoch 12/40 Iteration: 2920 | Train loss: 0.04700
Epoch 12/40 Iteration: 2940 | Train loss: 0.00268
Epoch 12/40 Iteration: 2960 | Train loss: 0.00674
Epoch 12/40 Iteration: 2980 | Train loss: 0.00976
Epoch 12/40 Iteration: 3000 | Train loss: 0.01709
Epoch 13/40 Iteration: 3020 | Train loss: 0.03490
Epoch 13/40 Iteration: 3040 | Train loss: 0.00312
Epoch 13/40 Iteration: 3060 | Train loss: 0.00823
Epoch 13/40 Iteration: 3080 | Train loss: 0.00156
Epoch 13/40 Iteration: 3100 | Train loss: 0.04337
Epoch 13/40 Iteration: 3120 | Train loss: 0.00398
Epoch 13/40 Iteration: 3140 | Train loss: 0.00146
Epoch 13/40 Iteration: 3160 | Train loss: 0.00721
Epoch 13/40 Iteration: 3180 | Train loss: 0.00946
Epoch 13/40 Iteration: 3200 | Train loss: 0.00099
Epoch 13/40 Iteration: 3220 | Train loss: 0.00106
Epoch 13/40 Iteration: 3240 | Train loss: 0.00315
Epoch 14/40 Iteration: 3260 | Train loss: 0.00652
Epoch 14/40 Iteration: 3280 | Train loss: 0.00096
Epoch 14/40 Iteration: 3300 | Train loss: 0.00814
Epoch 14/40 Iteration: 3320 | Train loss: 0.06354
Epoch 14/40 Iteration: 3340 | Train loss: 0.01688
Epoch 14/40 Iteration: 3360 | Train loss: 0.02321
Epoch 14/40 Iteration: 3380 | Train loss: 0.01933
Epoch 14/40 Iteration: 3400 | Train loss: 0.02442
Epoch 14/40 Iteration: 3420 | Train loss: 0.01667
Epoch 14/40 Iteration: 3440 | Train loss: 0.00360
Epoch 14/40 Iteration: 3460 | Train loss: 0.01194
Epoch 14/40 Iteration: 3480 | Train loss: 0.01263
Epoch 14/40 Iteration: 3500 | Train loss: 0.03146
Epoch 15/40 Iteration: 3520 | Train loss: 0.01771
Epoch 15/40 Iteration: 3540 | Train loss: 0.01262
Epoch 15/40 Iteration: 3560 | Train loss: 0.00821
Epoch 15/40 Iteration: 3580 | Train loss: 0.00374
Epoch 15/40 Iteration: 3600 | Train loss: 0.00602
Epoch 15/40 Iteration: 3620 | Train loss: 0.03432
Epoch 15/40 Iteration: 3640 | Train loss: 0.05936
Epoch 15/40 Iteration: 3660 | Train loss: 0.01452
Epoch 15/40 Iteration: 3680 | Train loss: 0.00446
Epoch 15/40 Iteration: 3700 | Train loss: 0.00493
Epoch 15/40 Iteration: 3720 | Train loss: 0.00211
Epoch 15/40 Iteration: 3740 | Train loss: 0.00135
Epoch 16/40 Iteration: 3760 | Train loss: 0.05862
Epoch 16/40 Iteration: 3780 | Train loss: 0.05829
Epoch 16/40 Iteration: 3800 | Train loss: 0.00303
Epoch 16/40 Iteration: 3820 | Train loss: 0.02846
Epoch 16/40 Iteration: 3840 | Train loss: 0.00167
Epoch 16/40 Iteration: 3860 | Train loss: 0.00099
Epoch 16/40 Iteration: 3880 | Train loss: 0.00229
Epoch 16/40 Iteration: 3900 | Train loss: 0.00661
Epoch 16/40 Iteration: 3920 | Train loss: 0.00192
Epoch 16/40 Iteration: 3940 | Train loss: 0.00803
Epoch 16/40 Iteration: 3960 | Train loss: 0.00058
Epoch 16/40 Iteration: 3980 | Train loss: 0.00127
Epoch 16/40 Iteration: 4000 | Train loss: 0.00253
Epoch 17/40 Iteration: 4020 | Train loss: 0.01097
Epoch 17/40 Iteration: 4040 | Train loss: 0.00050
Epoch 17/40 Iteration: 4060 | Train loss: 0.00105
Epoch 17/40 Iteration: 4080 | Train loss: 0.01823
Epoch 17/40 Iteration: 4100 | Train loss: 0.00426
Epoch 17/40 Iteration: 4120 | Train loss: 0.00056
Epoch 17/40 Iteration: 4140 | Train loss: 0.00078
Epoch 17/40 Iteration: 4160 | Train loss: 0.00436
Epoch 17/40 Iteration: 4180 | Train loss: 0.00032
Epoch 17/40 Iteration: 4200 | Train loss: 0.00167
Epoch 17/40 Iteration: 4220 | Train loss: 0.00582
Epoch 17/40 Iteration: 4240 | Train loss: 0.00052
Epoch 18/40 Iteration: 4260 | Train loss: 0.00166
Epoch 18/40 Iteration: 4280 | Train loss: 0.00031
Epoch 18/40 Iteration: 4300 | Train loss: 0.00048
Epoch 18/40 Iteration: 4320 | Train loss: 0.00024
Epoch 18/40 Iteration: 4340 | Train loss: 0.00027
Epoch 18/40 Iteration: 4360 | Train loss: 0.00045
Epoch 18/40 Iteration: 4380 | Train loss: 0.00315
Epoch 18/40 Iteration: 4400 | Train loss: 0.00172
Epoch 18/40 Iteration: 4420 | Train loss: 0.00097
Epoch 18/40 Iteration: 4440 | Train loss: 0.00177
Epoch 18/40 Iteration: 4460 | Train loss: 0.00016
Epoch 18/40 Iteration: 4480 | Train loss: 0.00140
Epoch 18/40 Iteration: 4500 | Train loss: 0.00065
Epoch 19/40 Iteration: 4520 | Train loss: 0.00054
Epoch 19/40 Iteration: 4540 | Train loss: 0.00013
Epoch 19/40 Iteration: 4560 | Train loss: 0.00155
Epoch 19/40 Iteration: 4580 | Train loss: 0.00126
Epoch 19/40 Iteration: 4600 | Train loss: 0.00081
Epoch 19/40 Iteration: 4620 | Train loss: 0.00749
Epoch 19/40 Iteration: 4640 | Train loss: 0.00091
Epoch 19/40 Iteration: 4660 | Train loss: 0.00047
Epoch 19/40 Iteration: 4680 | Train loss: 0.00175
Epoch 19/40 Iteration: 4700 | Train loss: 0.00025
Epoch 19/40 Iteration: 4720 | Train loss: 0.00011
Epoch 19/40 Iteration: 4740 | Train loss: 0.00831
Epoch 20/40 Iteration: 4760 | Train loss: 0.01487
Epoch 20/40 Iteration: 4780 | Train loss: 0.00603
Epoch 20/40 Iteration: 4800 | Train loss: 0.01694
Epoch 20/40 Iteration: 4820 | Train loss: 0.01253
Epoch 20/40 Iteration: 4840 | Train loss: 0.00288
Epoch 20/40 Iteration: 4860 | Train loss: 0.00451
Epoch 20/40 Iteration: 4880 | Train loss: 0.02055
Epoch 20/40 Iteration: 4900 | Train loss: 0.01078
Epoch 20/40 Iteration: 4920 | Train loss: 0.00487
Epoch 20/40 Iteration: 4940 | Train loss: 0.00294
Epoch 20/40 Iteration: 4960 | Train loss: 0.00856
Epoch 20/40 Iteration: 4980 | Train loss: 0.00181
Epoch 20/40 Iteration: 5000 | Train loss: 0.00139
Epoch 21/40 Iteration: 5020 | Train loss: 0.00090
Epoch 21/40 Iteration: 5040 | Train loss: 0.00117
Epoch 21/40 Iteration: 5060 | Train loss: 0.00143
Epoch 21/40 Iteration: 5080 | Train loss: 0.03607
Epoch 21/40 Iteration: 5100 | Train loss: 0.00123
Epoch 21/40 Iteration: 5120 | Train loss: 0.00812
Epoch 21/40 Iteration: 5140 | Train loss: 0.00473
Epoch 21/40 Iteration: 5160 | Train loss: 0.00131
Epoch 21/40 Iteration: 5180 | Train loss: 0.00670
Epoch 21/40 Iteration: 5200 | Train loss: 0.00061
Epoch 21/40 Iteration: 5220 | Train loss: 0.01438
Epoch 21/40 Iteration: 5240 | Train loss: 0.00527
Epoch 22/40 Iteration: 5260 | Train loss: 0.00090
Epoch 22/40 Iteration: 5280 | Train loss: 0.00009
Epoch 22/40 Iteration: 5300 | Train loss: 0.00148
Epoch 22/40 Iteration: 5320 | Train loss: 0.00101
Epoch 22/40 Iteration: 5340 | Train loss: 0.00213
Epoch 22/40 Iteration: 5360 | Train loss: 0.00734
Epoch 22/40 Iteration: 5380 | Train loss: 0.01303
Epoch 22/40 Iteration: 5400 | Train loss: 0.00182
Epoch 22/40 Iteration: 5420 | Train loss: 0.00582
Epoch 22/40 Iteration: 5440 | Train loss: 0.00060
Epoch 22/40 Iteration: 5460 | Train loss: 0.01637
Epoch 22/40 Iteration: 5480 | Train loss: 0.00085
Epoch 22/40 Iteration: 5500 | Train loss: 0.00254
Epoch 23/40 Iteration: 5520 | Train loss: 0.00182
Epoch 23/40 Iteration: 5540 | Train loss: 0.00027
Epoch 23/40 Iteration: 5560 | Train loss: 0.00682
Epoch 23/40 Iteration: 5580 | Train loss: 0.00063
Epoch 23/40 Iteration: 5600 | Train loss: 0.00036
Epoch 23/40 Iteration: 5620 | Train loss: 0.00112
Epoch 23/40 Iteration: 5640 | Train loss: 0.00180
Epoch 23/40 Iteration: 5660 | Train loss: 0.00068
Epoch 23/40 Iteration: 5680 | Train loss: 0.00581
Epoch 23/40 Iteration: 5700 | Train loss: 0.00029
Epoch 23/40 Iteration: 5720 | Train loss: 0.00029
Epoch 23/40 Iteration: 5740 | Train loss: 0.00034
Epoch 24/40 Iteration: 5760 | Train loss: 0.00078
Epoch 24/40 Iteration: 5780 | Train loss: 0.00044
Epoch 24/40 Iteration: 5800 | Train loss: 0.00037
Epoch 24/40 Iteration: 5820 | Train loss: 0.00046
Epoch 24/40 Iteration: 5840 | Train loss: 0.00139
Epoch 24/40 Iteration: 5860 | Train loss: 0.00100
Epoch 24/40 Iteration: 5880 | Train loss: 0.00453
Epoch 24/40 Iteration: 5900 | Train loss: 0.00488
Epoch 24/40 Iteration: 5920 | Train loss: 0.09039
Epoch 24/40 Iteration: 5940 | Train loss: 0.00176
Epoch 24/40 Iteration: 5960 | Train loss: 0.00269
Epoch 24/40 Iteration: 5980 | Train loss: 0.00051
Epoch 24/40 Iteration: 6000 | Train loss: 0.00550
Epoch 25/40 Iteration: 6020 | Train loss: 0.00161
Epoch 25/40 Iteration: 6040 | Train loss: 0.00051
Epoch 25/40 Iteration: 6060 | Train loss: 0.00130
Epoch 25/40 Iteration: 6080 | Train loss: 0.00112
Epoch 25/40 Iteration: 6100 | Train loss: 0.00041
Epoch 25/40 Iteration: 6120 | Train loss: 0.00255
Epoch 25/40 Iteration: 6140 | Train loss: 0.00083
Epoch 25/40 Iteration: 6160 | Train loss: 0.00149
Epoch 25/40 Iteration: 6180 | Train loss: 0.00144
Epoch 25/40 Iteration: 6200 | Train loss: 0.00058
Epoch 25/40 Iteration: 6220 | Train loss: 0.00027
Epoch 25/40 Iteration: 6240 | Train loss: 0.00043
Epoch 26/40 Iteration: 6260 | Train loss: 0.00058
Epoch 26/40 Iteration: 6280 | Train loss: 0.00026
Epoch 26/40 Iteration: 6300 | Train loss: 0.00025
Epoch 26/40 Iteration: 6320 | Train loss: 0.00235
Epoch 26/40 Iteration: 6340 | Train loss: 0.00308
Epoch 26/40 Iteration: 6360 | Train loss: 0.00037
Epoch 26/40 Iteration: 6380 | Train loss: 0.00193
Epoch 26/40 Iteration: 6400 | Train loss: 0.00073
Epoch 26/40 Iteration: 6420 | Train loss: 0.00053
Epoch 26/40 Iteration: 6440 | Train loss: 0.00106
Epoch 26/40 Iteration: 6460 | Train loss: 0.00497
Epoch 26/40 Iteration: 6480 | Train loss: 0.00177
Epoch 26/40 Iteration: 6500 | Train loss: 0.00059
Epoch 27/40 Iteration: 6520 | Train loss: 0.00135
Epoch 27/40 Iteration: 6540 | Train loss: 0.00138
Epoch 27/40 Iteration: 6560 | Train loss: 0.00247
Epoch 27/40 Iteration: 6580 | Train loss: 0.00159
Epoch 27/40 Iteration: 6600 | Train loss: 0.08248
Epoch 27/40 Iteration: 6620 | Train loss: 0.00895
Epoch 27/40 Iteration: 6640 | Train loss: 0.00539
Epoch 27/40 Iteration: 6660 | Train loss: 0.01875
Epoch 27/40 Iteration: 6680 | Train loss: 0.00133
Epoch 27/40 Iteration: 6700 | Train loss: 0.00087
Epoch 27/40 Iteration: 6720 | Train loss: 0.00988
Epoch 27/40 Iteration: 6740 | Train loss: 0.00096
Epoch 28/40 Iteration: 6760 | Train loss: 0.00093
Epoch 28/40 Iteration: 6780 | Train loss: 0.00045
Epoch 28/40 Iteration: 6800 | Train loss: 0.00052
Epoch 28/40 Iteration: 6820 | Train loss: 0.00032
Epoch 28/40 Iteration: 6840 | Train loss: 0.00090
Epoch 28/40 Iteration: 6860 | Train loss: 0.00033
Epoch 28/40 Iteration: 6880 | Train loss: 0.00033
Epoch 28/40 Iteration: 6900 | Train loss: 0.00080
Epoch 28/40 Iteration: 6920 | Train loss: 0.00079
Epoch 28/40 Iteration: 6940 | Train loss: 0.00043
Epoch 28/40 Iteration: 6960 | Train loss: 0.00114
Epoch 28/40 Iteration: 6980 | Train loss: 0.00023
Epoch 28/40 Iteration: 7000 | Train loss: 0.00060
Epoch 29/40 Iteration: 7020 | Train loss: 0.00055
Epoch 29/40 Iteration: 7040 | Train loss: 0.00014
Epoch 29/40 Iteration: 7060 | Train loss: 0.00038
Epoch 29/40 Iteration: 7080 | Train loss: 0.00022
Epoch 29/40 Iteration: 7100 | Train loss: 0.00041
Epoch 29/40 Iteration: 7120 | Train loss: 0.00012
Epoch 29/40 Iteration: 7140 | Train loss: 0.00007
Epoch 29/40 Iteration: 7160 | Train loss: 0.00032
Epoch 29/40 Iteration: 7180 | Train loss: 0.00058
Epoch 29/40 Iteration: 7200 | Train loss: 0.00017
Epoch 29/40 Iteration: 7220 | Train loss: 0.00005
Epoch 29/40 Iteration: 7240 | Train loss: 0.00008
Epoch 30/40 Iteration: 7260 | Train loss: 0.00009
Epoch 30/40 Iteration: 7280 | Train loss: 0.00019
Epoch 30/40 Iteration: 7300 | Train loss: 0.00014
Epoch 30/40 Iteration: 7320 | Train loss: 0.00010
Epoch 30/40 Iteration: 7340 | Train loss: 0.00023
Epoch 30/40 Iteration: 7360 | Train loss: 0.00007
Epoch 30/40 Iteration: 7380 | Train loss: 0.00013
Epoch 30/40 Iteration: 7400 | Train loss: 0.00062
Epoch 30/40 Iteration: 7420 | Train loss: 0.00005
Epoch 30/40 Iteration: 7440 | Train loss: 0.00016
Epoch 30/40 Iteration: 7460 | Train loss: 0.00012
Epoch 30/40 Iteration: 7480 | Train loss: 0.00008
Epoch 30/40 Iteration: 7500 | Train loss: 0.00011
Epoch 31/40 Iteration: 7520 | Train loss: 0.00013
Epoch 31/40 Iteration: 7540 | Train loss: 0.00002
Epoch 31/40 Iteration: 7560 | Train loss: 0.00009
Epoch 31/40 Iteration: 7580 | Train loss: 0.00005
Epoch 31/40 Iteration: 7600 | Train loss: 0.00006
Epoch 31/40 Iteration: 7620 | Train loss: 0.00037
Epoch 31/40 Iteration: 7640 | Train loss: 0.00006
Epoch 31/40 Iteration: 7660 | Train loss: 0.00022
Epoch 31/40 Iteration: 7680 | Train loss: 0.00011
Epoch 31/40 Iteration: 7700 | Train loss: 0.00016
Epoch 31/40 Iteration: 7720 | Train loss: 0.00004
Epoch 31/40 Iteration: 7740 | Train loss: 0.00004
Epoch 32/40 Iteration: 7760 | Train loss: 0.00008
Epoch 32/40 Iteration: 7780 | Train loss: 0.00004
Epoch 32/40 Iteration: 7800 | Train loss: 0.00002
Epoch 32/40 Iteration: 7820 | Train loss: 0.00002
Epoch 32/40 Iteration: 7840 | Train loss: 0.00025
Epoch 32/40 Iteration: 7860 | Train loss: 0.00005
Epoch 32/40 Iteration: 7880 | Train loss: 0.00004
Epoch 32/40 Iteration: 7900 | Train loss: 0.00005
Epoch 32/40 Iteration: 7920 | Train loss: 0.00001
Epoch 32/40 Iteration: 7940 | Train loss: 0.00009
Epoch 32/40 Iteration: 7960 | Train loss: 0.00002
Epoch 32/40 Iteration: 7980 | Train loss: 0.00010
Epoch 32/40 Iteration: 8000 | Train loss: 0.00003
Epoch 33/40 Iteration: 8020 | Train loss: 0.00009
Epoch 33/40 Iteration: 8040 | Train loss: 0.00006
Epoch 33/40 Iteration: 8060 | Train loss: 0.00007
Epoch 33/40 Iteration: 8080 | Train loss: 0.00006
Epoch 33/40 Iteration: 8100 | Train loss: 0.00004
Epoch 33/40 Iteration: 8120 | Train loss: 0.00114
Epoch 33/40 Iteration: 8140 | Train loss: 0.00001
Epoch 33/40 Iteration: 8160 | Train loss: 0.00013
Epoch 33/40 Iteration: 8180 | Train loss: 0.00003
Epoch 33/40 Iteration: 8200 | Train loss: 0.00002
Epoch 33/40 Iteration: 8220 | Train loss: 0.00002
Epoch 33/40 Iteration: 8240 | Train loss: 0.00003
Epoch 34/40 Iteration: 8260 | Train loss: 0.00004
Epoch 34/40 Iteration: 8280 | Train loss: 0.00008
Epoch 34/40 Iteration: 8300 | Train loss: 0.00003
Epoch 34/40 Iteration: 8320 | Train loss: 0.00001
Epoch 34/40 Iteration: 8340 | Train loss: 0.00004
Epoch 34/40 Iteration: 8360 | Train loss: 0.00005
Epoch 34/40 Iteration: 8380 | Train loss: 0.00002
Epoch 34/40 Iteration: 8400 | Train loss: 0.00031
Epoch 34/40 Iteration: 8420 | Train loss: 0.00003
Epoch 34/40 Iteration: 8440 | Train loss: 0.00007
Epoch 34/40 Iteration: 8460 | Train loss: 0.00006
Epoch 34/40 Iteration: 8480 | Train loss: 0.00002
Epoch 34/40 Iteration: 8500 | Train loss: 0.00001
Epoch 35/40 Iteration: 8520 | Train loss: 0.00002
Epoch 35/40 Iteration: 8540 | Train loss: 0.00007
Epoch 35/40 Iteration: 8560 | Train loss: 0.00010
Epoch 35/40 Iteration: 8580 | Train loss: 0.00024
Epoch 35/40 Iteration: 8600 | Train loss: 0.00002
Epoch 35/40 Iteration: 8620 | Train loss: 0.00002
Epoch 35/40 Iteration: 8640 | Train loss: 0.00001
Epoch 35/40 Iteration: 8660 | Train loss: 0.00002
Epoch 35/40 Iteration: 8680 | Train loss: 0.00002
Epoch 35/40 Iteration: 8700 | Train loss: 0.00002
Epoch 35/40 Iteration: 8720 | Train loss: 0.00001
Epoch 35/40 Iteration: 8740 | Train loss: 0.00003
Epoch 36/40 Iteration: 8760 | Train loss: 0.00002
Epoch 36/40 Iteration: 8780 | Train loss: 0.00026
Epoch 36/40 Iteration: 8800 | Train loss: 0.00002
Epoch 36/40 Iteration: 8820 | Train loss: 0.00000
Epoch 36/40 Iteration: 8840 | Train loss: 0.00006
Epoch 36/40 Iteration: 8860 | Train loss: 0.00005
Epoch 36/40 Iteration: 8880 | Train loss: 0.00005
Epoch 36/40 Iteration: 8900 | Train loss: 0.00009
Epoch 36/40 Iteration: 8920 | Train loss: 0.00003
Epoch 36/40 Iteration: 8940 | Train loss: 0.00002
Epoch 36/40 Iteration: 8960 | Train loss: 0.00002
Epoch 36/40 Iteration: 8980 | Train loss: 0.00001
Epoch 36/40 Iteration: 9000 | Train loss: 0.00001
Epoch 37/40 Iteration: 9020 | Train loss: 0.00006
Epoch 37/40 Iteration: 9040 | Train loss: 0.00001
Epoch 37/40 Iteration: 9060 | Train loss: 0.00007
Epoch 37/40 Iteration: 9080 | Train loss: 0.00003
Epoch 37/40 Iteration: 9100 | Train loss: 0.00001
Epoch 37/40 Iteration: 9120 | Train loss: 0.00012
Epoch 37/40 Iteration: 9140 | Train loss: 0.00007
Epoch 37/40 Iteration: 9160 | Train loss: 0.00005
Epoch 37/40 Iteration: 9180 | Train loss: 0.00001
Epoch 37/40 Iteration: 9200 | Train loss: 0.00001
Epoch 37/40 Iteration: 9220 | Train loss: 0.00004
Epoch 37/40 Iteration: 9240 | Train loss: 0.00001
Epoch 38/40 Iteration: 9260 | Train loss: 0.00002
Epoch 38/40 Iteration: 9280 | Train loss: 0.00001
Epoch 38/40 Iteration: 9300 | Train loss: 0.00001
Epoch 38/40 Iteration: 9320 | Train loss: 0.00001
Epoch 38/40 Iteration: 9340 | Train loss: 0.00003
Epoch 38/40 Iteration: 9360 | Train loss: 0.00001
Epoch 38/40 Iteration: 9380 | Train loss: 0.00001
Epoch 38/40 Iteration: 9400 | Train loss: 0.00001
Epoch 38/40 Iteration: 9420 | Train loss: 0.00001
Epoch 38/40 Iteration: 9440 | Train loss: 0.00002
Epoch 38/40 Iteration: 9460 | Train loss: 0.00002
Epoch 38/40 Iteration: 9480 | Train loss: 0.00002
Epoch 38/40 Iteration: 9500 | Train loss: 0.00006
Epoch 39/40 Iteration: 9520 | Train loss: 0.00092
Epoch 39/40 Iteration: 9540 | Train loss: 0.00306
Epoch 39/40 Iteration: 9560 | Train loss: 0.01709
Epoch 39/40 Iteration: 9580 | Train loss: 0.01798
Epoch 39/40 Iteration: 9600 | Train loss: 0.12664
Epoch 39/40 Iteration: 9620 | Train loss: 0.01687
Epoch 39/40 Iteration: 9640 | Train loss: 0.00727
Epoch 39/40 Iteration: 9660 | Train loss: 0.00462
Epoch 39/40 Iteration: 9680 | Train loss: 0.05224
Epoch 39/40 Iteration: 9700 | Train loss: 0.00544
Epoch 39/40 Iteration: 9720 | Train loss: 0.00282
Epoch 39/40 Iteration: 9740 | Train loss: 0.00656
Epoch 40/40 Iteration: 9760 | Train loss: 0.09288
Epoch 40/40 Iteration: 9780 | Train loss: 0.00726
Epoch 40/40 Iteration: 9800 | Train loss: 0.00167
Epoch 40/40 Iteration: 9820 | Train loss: 0.00115
Epoch 40/40 Iteration: 9840 | Train loss: 0.00141
Epoch 40/40 Iteration: 9860 | Train loss: 0.00069
Epoch 40/40 Iteration: 9880 | Train loss: 0.00128
Epoch 40/40 Iteration: 9900 | Train loss: 0.03402
Epoch 40/40 Iteration: 9920 | Train loss: 0.01177
Epoch 40/40 Iteration: 9940 | Train loss: 0.00268
Epoch 40/40 Iteration: 9960 | Train loss: 0.00092
Epoch 40/40 Iteration: 9980 | Train loss: 0.00989
Epoch 40/40 Iteration: 10000 | Train loss: 0.01256
```

The trained model is saved using TensorFlow's checkpointing system. Now, we can use the trained model for predicting the class labels on the test set, as follows:

```
In [9]: preds = rnn.predict(X_test)
        y_true = y_test[:len(preds)]
        print('Test Acc.: %.3f' % (np.sum(preds==y_true) / len(y_true)))
```

```
INFO:tensorflow:Restoring parameters from ./model/sentiment-39.ckpt
Test Acc.: 0.853
```

The result will show an accuracy of 86 percent. Given the small size of this dataset, this is comparable to the test prediction accuracy obtained in Chapter 8.

We can optimize this further by changing the hyperparameters of the model, such as *lstm_size*, *seq_len*, and *embed_size*, to achieve better generalization performance. However, for hyperparameter tuning, it is recommended that we create a separate validation set and that we do not repeatedly use the test set for evaluation to avoid introducing bias through test data leakage.

Also, if you are interested in the prediction probabilities on the test set rather than the class labels, then you can set *return_proba=True* as follows:

```
In [10]: proba = rnn.predict(X_test, return_proba=True)
         print(proba)

INFO:tensorflow:Restoring parameters from ./model/sentiment-39.ckpt
[1.7920318e-06 9.9999547e-01 6.9740318e-02 ... 6.9026237e-06 1.4025514e-01
 9.9837083e-01]
```

So this was our first RNN model for sentiment analysis. We will now go further and create an RNN for character-by-character language modeling in TensorFlow, as another popular application of sequence modeling.

6 Project two - implementing an RNN for character-level language modeling in TensorFlow

Language modeling is a fascinating application that enables machines to perform human-language-related tasks, such as generating English sentences.

In the model that we will build now, the input is a text document, and our goal is to develop a model that can generate new text similar to the input document. Examples of such an input can be a book or a computer program in a specific programming language.

In character-level language modeling, the input is broken down into a sequence of characters that are fed into our network one character at time. The network will process each new character in conjunction with the memory of the previously seen characters to predict the new character. The following figure shows an example of character-level language modeling:

We can break this implementation down into three separate steps - preparing the data, building the RNN model, and performing next-character prediction and sampling to generate new text.

If you recall from the previous sections of this chapter, we mentioned the exploding gradient problem. In this application, we will also get a chance to play with a gradient clipping technique to avoid this exploding gradient problem.

6.1 Preparing the data

In this section, we prepare the data for character-level language modeling.

To get the input data, visit the Project Gutenberg website at https://www.gutenberg.org, which provides thousands of free e-books. For our example, we can get the book *The Tragedie of Hamlet* by William Shakespeare in plain text format from http://www.gutenberg.org/cache/epub/2265/pg2265.txt.

Once we have some data, we can read it into a Python session as plain text. In the following code, the Python variable *chars* represents the set of *unique* characters observed in this text. We then create a dictionary that maps each character to an integer, *char2int*, and a dictionary that performs reverse mapping, for instance, mapping integers to those unique characters - *int2char*. Using the *char2int* dictionary, we convert the text into a NumPy array of integers. The following figure shows an example of converting characters into integers and the reverse for the words *"Hello"* and *"world"*:

This code reads the text from the downloaded link, removes the beginning portion of the text that contains some legal description of the Gutenberg project, and then constructs the dictionaries based on the text:

```
In [11]: import numpy as np

         # Reading and processing text
         with open('pg2265.txt', 'r', encoding='utf-8') as f:
             text = f.read()
         text = text[15858:]
         chars = set(text)
         char2int = {ch:i for i, ch in enumerate(chars)}
         int2char = dict(enumerate(chars))
         text_ints = np.array([char2int[ch] for ch in text], dtype=np.int32)
```

Now, we should reshape the data into batches of sequences, the most important step in preparing the data. As we know, the goal is to predict the next character based on the sequence of characters that we have observed so far. Therefore, we shift the input *x* and output *y* of the neural network by one character. The following figure shows the preprocessing steps, starting from a text corpus to generating data arrays for *x* and *y*:

As you can see in this figure, the training arrays *x* and *y* have the same shapes or dimensions, where the number of rows is equal to the *batch size* and the number of columns is *number of batches x number of steps*.

Given the input array *data* that contains the integers that correspond to the characters in the text corpus, the following function will generate *x* and *y* with the same structure shown in the previous figure:

```
In [12]: def reshape_data(sequence, batch_size, num_steps):
             tot_batch_length = batch_size * num_steps
             num_batches = int(len(sequence) / tot_batch_length)
             if num_batches*tot_batch_length + 1 > len(sequence):
                 num_batches = num_batches - 1
             # Truncate the sequence at the end to get rid of
             # remaining characters that do not make a full batch
             x = sequence[0:num_batches*tot_batch_length]
             y = sequence[1:num_batches*tot_batch_length+1]
             # Split x and y into a list batches of sequences
             x_batch_splits = np.split(x, batch_size)
             y_batch_splits = np.split(y, batch_size)
             # Stack the batches together
             # batch size x tot_bach_length
             x = np.stack(x_batch_splits)
             y = np.stack(y_batch_splits)

             return x, y
```

The next step is to split the arrays *x* and *y* into mini-batches where each row is a sequence with length equal to the *number of steps*. The process of splitting the data array *x* is shown in the following figure:

In the following code, we define a function named *create_batch_generator* that splits the data arrays *x* and *y*, as shown in the previous figure, and outputs a batch generator. Later, we will use this generator to iterate through the mini-batches during the training of our network:

```
In [16]: def create_batch_generator(data_x, data_y, num_steps):
             batch_size, tot_batch_length = data_x.shape
             num_batches = int(tot_batch_length/num_steps)
             for b in range(num_batches):
                 yield (data_x[:, b*num_steps:(b+1)*num_steps],
                        data_y[:, b*num_steps:(b+1)*num_steps])
```

At this points, we have now completed the data preprocessing steps, and we have the data in the proper format. In the next section, we will implement the RNN model for character-level language modeling.

6.2 Building a character-level RNN model

To build a character-level neural network, we will implement a class called *CharRNN* that constructs the graph of the RNN in order to predict the next character, after observing a given sequence of characters. From the classification perspective, the number of classes is the total number of unique characters that exists in the text corpus. The *CharRNN* class has four methods, as follows:

- A constructor that sets up the learning parameters, creates a computation graph, and calls the *build* method to construct the graph based on the sampling mode versus the training mode.
- A *build* method that defines the placeholders for feeding the data, constructs the RNN using LSTM cells, and defines the output of the network, the cost function, and the optimizer.
- A *train* method to iterate through the mini-batches and train the network for the specified number of epochs.
- A *sample* method to start from a given string, calculate the probabilities for the next character, and choose a character randomly according to these probabilities. This process will be repeated, and the sampled characters will be concatenated together to form a string. Once the size of this string reaches the specified length, it will return the string.

We will break these four methods into separate code sections and explain each one. Note that implementing the RNN part of this model is very similar to the implementation in the *Project One*. So, we will skip the description of building the RNN components here.

6.3 The constructor

In contrast to our previous implementation for sentiment analysis, where the same computation graph was used for both training and prediction modes, this time our computation graph is going to be different for the training versus the sampling mode.

Therefore, we need to add a new Boolean type argument to the constructor, to determine whether we are building the model for the training mode or the sampling mode. The following code shows the implementation of the constructor enclosed in the class definition:

```
In [26]: import tensorflow as tf
         import os

         def get_top_char(probas, char_size, top_n=5):
             p = np.squeeze(probas)
             p[np.argsort(p)[:-top_n]] = 0.0
             p = p / np.sum(p)
             ch_id = np.random.choice(char_size, 1, p=p)[0]
             return ch_id

         class CharRNN(object):

             def __init__(self, num_classes, batch_size=64,
                          num_steps=100, lstm_size=128,
                          num_layers=1, learning_rate=0.001,
                          keep_prob=0.5, grad_clip=5,
                          sampling=False):
                 self.num_classes = num_classes
                 self.batch_size = batch_size
                 self.num_steps = num_steps
                 self.lstm_size = lstm_size
                 self.num_layers = num_layers
                 self.learning_rate = learning_rate
                 self.keep_prob = keep_prob
                 self.grad_clip = grad_clip

                 self.g = tf.Graph()
                 with self.g.as_default():
                     tf.set_random_seed(123)

                     self.build(sampling=sampling)

                     self.saver = tf.train.Saver()

                     self.init_op = tf.global_variables_initializer()

             def build(self, sampling):
                 if sampling == True:
                     batch_size, num_steps = 1, 1
                 else:
                     batch_size = self.batch_size
                     num_steps = self.num_steps

                 ## Define the placeholders
                 tf_x = tf.placeholder(tf.int32, shape=(batch_size, num_steps), name='tf_x')
                 tf_y = tf.placeholder(tf.int32, shape=(batch_size, num_steps), name='tf_y')
                 tf_keepprob = tf.placeholder(tf.float32, name='tf_keepprob')
```

```
        ## One-hot encoding
        x_onehot = tf.one_hot(tf_x, depth=self.num_classes)
        y_onehot = tf.one_hot(tf_y, depth=self.num_classes)

        ## Build the multi-layer RNN cells
        cells = tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.DropoutWrapper(tf.contrib.r

        ## Define the initial state
        self.initial_state = cells.zero_state(batch_size, tf.float32)

        ## Run each sequence step through the RNN
        lstm_outputs, self.final_state = tf.nn.dynamic_rnn(cells, x_onehot, initial_sta

        seq_output_reshaped = tf.reshape(lstm_outputs, shape=[-1, self.lstm_size], name

        logits = tf.layers.dense(inputs=seq_output_reshaped, units=self.num_classes, ac

        proba = tf.nn.softmax(logits, name='probabilities')

        y_reshaped = tf.reshape(y_onehot, shape=[-1, self.num_classes], name='y_reshape

        cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_reshaped

        # Gradient clipping to avoid "exploding gradients"
        tvars = tf.trainable_variables()
        grads, _ = tf.clip_by_global_norm(tf.gradients(cost, tvars), self.grad_clip)

        optimizer = tf.train.AdamOptimizer(self.learning_rate)
        train_op = optimizer.apply_gradients(zip(grads, tvars), name='train_op')

    def train(self, train_x, train_y, num_epochs,
              ckpt_dir='./model/'):
        ## Create the checkpoint directory
        ## if it does not exist
        if not os.path.exists(ckpt_dir):
            os.mkdir(ckpt_dir)

        with tf.Session(graph=self.g) as sess:
            sess.run(self.init_op)

            n_batches = int(train_x.shape[1] / self.num_steps)
            iterations = n_batches * num_epochs

            for epoch in range(num_epochs):
                ## Train network
                new_state = sess.run(self.initial_state)
                loss = 0
                ## Mini-batch generator
                bgen = create_batch_generator(train_x, train_y, self.num_steps)

                for b, (batch_x, batch_y) in enumerate(bgen, 1):
                    iteration = epoch*n_batches + b

                    feed = {'tf_x:0': batch_x,
                            'tf_y:0': batch_y,
                            'tf_keepprob:0': self.keep_prob,
                            self.initial_state: new_state}
                    batch_cost, _, new_state = sess.run(['cost:0', 'train_op', self.fin

                    if iteration % 10 == 0:
                        print('Epoch %d/%d Iteration: %d | Training loss: %.4f' % (epoc

                ## Save the trained model
                self.saver.save(sess, os.path.join(ckpt_dir, 'language_modeling.ckpt'))

    def sample(self, output_length, ckpt_dir,
               starter_seq="The "):
        observed_seq = [ch for ch in starter_seq]
        with tf.Session(graph=self.g) as sess:
            self.saver.restore(sess, tf.train.latest_checkpoint(ckpt_dir))

            ## 1: run the model using the starter sequence
            new_state = sess.run(self.initial_state)
            for ch in starter_seq:
                x = np.zeros((1, 1))
                x[0, 0] = char2int[ch]
                feed = {'tf_x:0': x,
                        'tf_keepprob:0': 1.0,
                        self.initial_state: new_state}
                proba, new_state = sess.run(['probabilities:0', self.final_state], feed

            ch_id = get_top_char(proba, len(chars))
            observed_seq.append(int2char[ch_id])

            ## 2: run the model using the updated observed_seq
            for i in range(output_length):
                x[0,0] = ch_id
                feed = {'tf_x:0': x,
                        'tf_keepprob:0': 1.0,
                        self.initial_state: new_state}
                proba, new_state = sess.run(['probabilities:0', self.final_state], feed

                ch_id = get_top_char(proba, len(chars))
                observed_seq.append(int2char[ch_id])

        return ''.join(observed_seq)
```

As we planned earlier, the Boolean *sampling* argument is used to determine whether the instance of *CharRNN* is for building the graph in the training mode (*sampling=False*) or the sampling mode (*sampling=True*).

In addition to the *sampling* argument, we have introduced a new argument called *grad_clip*, which is used for clipping the gradients to avoid the exploding gradient problem that we mentioned earlier.

Then, similar to the previous implementation, the constructor creates a computation graph, sets the graph-level random seed for consistent output, and builds the graph by calling the *build* method.

6.4 The build method

The next method of the *CharRNN* class is *build*, which is very similar to the *build* method in the *Project One*, except for some minor differences. The *build* method first defines two local variables, *batch_size* and *num_steps*, based on the mode, as follows:

- In sampling mode: batch_size = 1 and num_steps = 1
- In training mode: batch_size = self.batch_size and num_steps = self.num_steps

Recall that in the sentiment analysis implementation, we used an embedding layer to create a salient representation of the unique words in the dataset. In contrast, here we are using the one-hot encoding scheme for both x and y with *depth=num_classes*, where *num_classes* is in fact the total number of characters in the text corpus.

Building a multilayer RNN component of the model is exactly the same as in our sentiment analysis implementation, using the *tf.nn.dynamic_rnn* function. However, *outputs* form the *tf.nn.dynamic_rnn* function is a three-dimensional tensor with this shape - *batch_size, num_steps, lstm_size*. Next, this tensor will be reshaped into a two-dimensional tensor with the *batch_size*num_steps, lstm_size* shape, which is passed to the *tf.layers.dense* function to make a fully connected layer and obtain *logits* (net inputs). Finally, the probabilities for the next batch of characters are obtained and the cost function is defined. In addition, here, we apply gradient clipping using the *tf.clip_by_global_norm* function to avoid the exploding gradient problem.

The following code shows the implementation of what we have just described for our new *build* method:

6.5 The train method

The next method of the *CharRNN* class is the *train* method, which is very similar to the *train* method described in the *Project One*. Here is the *train* method code, which will look very familiar to the sentiment analysis version we built earlier in this chapter:

6.6 The sample method

The final method in our *CharRNN* class is the *sample* method. The behavior of this *sample* method is similar to that of the *predict* method that we implemented in the *Project One*. However, the difference here is that we calculate the probabilities for the next character from an observed sequence - *observed_seq*. Then, these probabilities are passed to a function named *get_top_char*, which randomly selects one character according to the obtained probabilities.

Initially, the observed sequence starts from *starter_seq*, which is provided as an argument. When new characters are sampled according to their predicted probabilities, they are appended to the observed sequence, and the new observed sequence is used for predicting the next character.

The implementation of the *sample* method is as follows:

So here, the *sample* method calls the *get_top_char* function to choose a character ID randomly (*ch_id*) according to the obtained probabilities.

In this *get_top_char* function, the probabilities are first sorted, then the *top_n* probabilities are passed to the *numpy.random.choice* function to randomly select one out of these top probabilities. The implementation of the *get_top_char* function is as follows:

6.7 Creating and training the CharRNN Model

Now we are ready to create an instance of the *CharRNN* class to build the RNN model, and to train it with the following configurations:

```
In [21]: batch_size = 64
         num_steps = 100

         train_x, train_y = reshape_data(text_ints, batch_size,
                                         num_steps)

         rnn = CharRNN(num_classes=len(chars), batch_size=batch_size)
         rnn.train(train_x, train_y, num_epochs=100,
                   ckpt_dir='./model-100/')
```

```
WARNING:tensorflow:From <ipython-input-20-784f6e96ca40>:70: softmax_cross_entropy_with_logits (f
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See tf.nn.softmax_cross_entropy_with_logits_v2.

Epoch 1/100 Iteration: 10 | Training loss: 3.7287
Epoch 1/100 Iteration: 20 | Training loss: 3.3645
Epoch 2/100 Iteration: 30 | Training loss: 3.2881
Epoch 2/100 Iteration: 40 | Training loss: 3.2401
Epoch 2/100 Iteration: 50 | Training loss: 3.2364
Epoch 3/100 Iteration: 60 | Training loss: 3.2156
Epoch 3/100 Iteration: 70 | Training loss: 3.1875
Epoch 4/100 Iteration: 80 | Training loss: 3.1801
Epoch 4/100 Iteration: 90 | Training loss: 3.1576
Epoch 4/100 Iteration: 100 | Training loss: 3.1440
Epoch 5/100 Iteration: 110 | Training loss: 3.1313
Epoch 5/100 Iteration: 120 | Training loss: 3.0920
Epoch 6/100 Iteration: 130 | Training loss: 3.0538
Epoch 6/100 Iteration: 140 | Training loss: 3.0104
Epoch 6/100 Iteration: 150 | Training loss: 2.9652
Epoch 7/100 Iteration: 160 | Training loss: 2.9453
Epoch 7/100 Iteration: 170 | Training loss: 2.8769
Epoch 8/100 Iteration: 180 | Training loss: 2.8283
Epoch 8/100 Iteration: 190 | Training loss: 2.8002
Epoch 8/100 Iteration: 200 | Training loss: 2.7327
Epoch 9/100 Iteration: 210 | Training loss: 2.7317
Epoch 9/100 Iteration: 220 | Training loss: 2.6803
Epoch 10/100 Iteration: 230 | Training loss: 2.6414
Epoch 10/100 Iteration: 240 | Training loss: 2.6480
Epoch 10/100 Iteration: 250 | Training loss: 2.5841
Epoch 11/100 Iteration: 260 | Training loss: 2.5952
Epoch 11/100 Iteration: 270 | Training loss: 2.5511
Epoch 12/100 Iteration: 280 | Training loss: 2.5217
Epoch 12/100 Iteration: 290 | Training loss: 2.5313
Epoch 12/100 Iteration: 300 | Training loss: 2.4686
Epoch 13/100 Iteration: 310 | Training loss: 2.5037
Epoch 13/100 Iteration: 320 | Training loss: 2.4569
Epoch 14/100 Iteration: 330 | Training loss: 2.4390
Epoch 14/100 Iteration: 340 | Training loss: 2.4585
Epoch 14/100 Iteration: 350 | Training loss: 2.4061
Epoch 15/100 Iteration: 360 | Training loss: 2.4376
Epoch 15/100 Iteration: 370 | Training loss: 2.4012
Epoch 16/100 Iteration: 380 | Training loss: 2.3963
Epoch 16/100 Iteration: 390 | Training loss: 2.4088
Epoch 16/100 Iteration: 400 | Training loss: 2.3552
Epoch 17/100 Iteration: 410 | Training loss: 2.3895
Epoch 17/100 Iteration: 420 | Training loss: 2.3597
Epoch 18/100 Iteration: 430 | Training loss: 2.3523
Epoch 18/100 Iteration: 440 | Training loss: 2.3666
Epoch 18/100 Iteration: 450 | Training loss: 2.3031
Epoch 19/100 Iteration: 460 | Training loss: 2.3420
Epoch 19/100 Iteration: 470 | Training loss: 2.3299
Epoch 20/100 Iteration: 480 | Training loss: 2.3133
Epoch 20/100 Iteration: 490 | Training loss: 2.3320
Epoch 20/100 Iteration: 500 | Training loss: 2.2781
Epoch 21/100 Iteration: 510 | Training loss: 2.3243
Epoch 21/100 Iteration: 520 | Training loss: 2.3000
Epoch 22/100 Iteration: 530 | Training loss: 2.2866
Epoch 22/100 Iteration: 540 | Training loss: 2.3140
Epoch 22/100 Iteration: 550 | Training loss: 2.2470
Epoch 23/100 Iteration: 560 | Training loss: 2.2953
Epoch 23/100 Iteration: 570 | Training loss: 2.2581
Epoch 24/100 Iteration: 580 | Training loss: 2.2660
Epoch 24/100 Iteration: 590 | Training loss: 2.2801
Epoch 24/100 Iteration: 600 | Training loss: 2.2256
Epoch 25/100 Iteration: 610 | Training loss: 2.2753
Epoch 25/100 Iteration: 620 | Training loss: 2.2498
Epoch 26/100 Iteration: 630 | Training loss: 2.2367
Epoch 26/100 Iteration: 640 | Training loss: 2.2552
Epoch 26/100 Iteration: 650 | Training loss: 2.2047
Epoch 27/100 Iteration: 660 | Training loss: 2.2586
Epoch 27/100 Iteration: 670 | Training loss: 2.2261
Epoch 28/100 Iteration: 680 | Training loss: 2.2188
Epoch 28/100 Iteration: 690 | Training loss: 2.2539
Epoch 28/100 Iteration: 700 | Training loss: 2.1885
Epoch 29/100 Iteration: 710 | Training loss: 2.2420
Epoch 29/100 Iteration: 720 | Training loss: 2.1960
Epoch 30/100 Iteration: 730 | Training loss: 2.2118
Epoch 30/100 Iteration: 740 | Training loss: 2.2268
Epoch 30/100 Iteration: 750 | Training loss: 2.1524
Epoch 31/100 Iteration: 760 | Training loss: 2.2101
Epoch 31/100 Iteration: 770 | Training loss: 2.1898
Epoch 32/100 Iteration: 780 | Training loss: 2.1900
Epoch 32/100 Iteration: 790 | Training loss: 2.2180
Epoch 32/100 Iteration: 800 | Training loss: 2.1540
Epoch 33/100 Iteration: 810 | Training loss: 2.1993
Epoch 33/100 Iteration: 820 | Training loss: 2.1651
Epoch 34/100 Iteration: 830 | Training loss: 2.1770
Epoch 34/100 Iteration: 840 | Training loss: 2.1941
Epoch 34/100 Iteration: 850 | Training loss: 2.1325
Epoch 35/100 Iteration: 860 | Training loss: 2.1719
Epoch 35/100 Iteration: 870 | Training loss: 2.1503
Epoch 36/100 Iteration: 880 | Training loss: 2.1569
Epoch 36/100 Iteration: 890 | Training loss: 2.1768
Epoch 36/100 Iteration: 900 | Training loss: 2.1054
Epoch 37/100 Iteration: 910 | Training loss: 2.1583
Epoch 37/100 Iteration: 920 | Training loss: 2.1370
Epoch 38/100 Iteration: 930 | Training loss: 2.1483
Epoch 38/100 Iteration: 940 | Training loss: 2.1717
Epoch 38/100 Iteration: 950 | Training loss: 2.1090
Epoch 39/100 Iteration: 960 | Training loss: 2.1658
Epoch 39/100 Iteration: 970 | Training loss: 2.1236
Epoch 40/100 Iteration: 980 | Training loss: 2.1266
Epoch 40/100 Iteration: 990 | Training loss: 2.1457
Epoch 40/100 Iteration: 1000 | Training loss: 2.0978
Epoch 41/100 Iteration: 1010 | Training loss: 2.1495
Epoch 41/100 Iteration: 1020 | Training loss: 2.1010
Epoch 42/100 Iteration: 1030 | Training loss: 2.1175
Epoch 42/100 Iteration: 1040 | Training loss: 2.1498
Epoch 42/100 Iteration: 1050 | Training loss: 2.0769
Epoch 43/100 Iteration: 1060 | Training loss: 2.1378
Epoch 43/100 Iteration: 1070 | Training loss: 2.0973
Epoch 44/100 Iteration: 1080 | Training loss: 2.1011
Epoch 44/100 Iteration: 1090 | Training loss: 2.1266
Epoch 44/100 Iteration: 1100 | Training loss: 2.0645
Epoch 45/100 Iteration: 1110 | Training loss: 2.1236
Epoch 45/100 Iteration: 1120 | Training loss: 2.0859
Epoch 46/100 Iteration: 1130 | Training loss: 2.0816
Epoch 46/100 Iteration: 1140 | Training loss: 2.1206
Epoch 46/100 Iteration: 1150 | Training loss: 2.0473
Epoch 47/100 Iteration: 1160 | Training loss: 2.1034
Epoch 47/100 Iteration: 1170 | Training loss: 2.0759
Epoch 48/100 Iteration: 1180 | Training loss: 2.0780
Epoch 48/100 Iteration: 1190 | Training loss: 2.1055
Epoch 48/100 Iteration: 1200 | Training loss: 2.0366
Epoch 49/100 Iteration: 1210 | Training loss: 2.0856
Epoch 49/100 Iteration: 1220 | Training loss: 2.0537
Epoch 50/100 Iteration: 1230 | Training loss: 2.0632
Epoch 50/100 Iteration: 1240 | Training loss: 2.0967
Epoch 50/100 Iteration: 1250 | Training loss: 2.0340
Epoch 51/100 Iteration: 1260 | Training loss: 2.0845
Epoch 51/100 Iteration: 1270 | Training loss: 2.0371
Epoch 52/100 Iteration: 1280 | Training loss: 2.0349
Epoch 52/100 Iteration: 1290 | Training loss: 2.0874
Epoch 52/100 Iteration: 1300 | Training loss: 2.0174
Epoch 53/100 Iteration: 1310 | Training loss: 2.0649
Epoch 53/100 Iteration: 1320 | Training loss: 2.0341
Epoch 54/100 Iteration: 1330 | Training loss: 2.0379
Epoch 54/100 Iteration: 1340 | Training loss: 2.0683
Epoch 54/100 Iteration: 1350 | Training loss: 2.0049
Epoch 55/100 Iteration: 1360 | Training loss: 2.0462
Epoch 55/100 Iteration: 1370 | Training loss: 2.0259
Epoch 56/100 Iteration: 1380 | Training loss: 2.0299
Epoch 56/100 Iteration: 1390 | Training loss: 2.0668
Epoch 56/100 Iteration: 1400 | Training loss: 1.9945
Epoch 57/100 Iteration: 1410 | Training loss: 2.0448
Epoch 57/100 Iteration: 1420 | Training loss: 2.0284
Epoch 58/100 Iteration: 1430 | Training loss: 2.0231
Epoch 58/100 Iteration: 1440 | Training loss: 2.0512
Epoch 58/100 Iteration: 1450 | Training loss: 1.9832
Epoch 59/100 Iteration: 1460 | Training loss: 2.0277
Epoch 59/100 Iteration: 1470 | Training loss: 2.0039
Epoch 60/100 Iteration: 1480 | Training loss: 2.0046
Epoch 60/100 Iteration: 1490 | Training loss: 2.0537
Epoch 60/100 Iteration: 1500 | Training loss: 1.9773
Epoch 61/100 Iteration: 1510 | Training loss: 2.0235
Epoch 61/100 Iteration: 1520 | Training loss: 1.9941
Epoch 62/100 Iteration: 1530 | Training loss: 1.9986
Epoch 62/100 Iteration: 1540 | Training loss: 2.0255
Epoch 62/100 Iteration: 1550 | Training loss: 1.9739
Epoch 63/100 Iteration: 1560 | Training loss: 2.0103
Epoch 63/100 Iteration: 1570 | Training loss: 1.9979
Epoch 64/100 Iteration: 1580 | Training loss: 1.9969
Epoch 64/100 Iteration: 1590 | Training loss: 2.0227
Epoch 64/100 Iteration: 1600 | Training loss: 1.9530
Epoch 65/100 Iteration: 1610 | Training loss: 2.0092
Epoch 65/100 Iteration: 1620 | Training loss: 1.9868
Epoch 66/100 Iteration: 1630 | Training loss: 1.9812
Epoch 66/100 Iteration: 1640 | Training loss: 2.0193
Epoch 66/100 Iteration: 1650 | Training loss: 1.9446
Epoch 67/100 Iteration: 1660 | Training loss: 1.9951
Epoch 67/100 Iteration: 1670 | Training loss: 1.9750
Epoch 68/100 Iteration: 1680 | Training loss: 1.9604
Epoch 68/100 Iteration: 1690 | Training loss: 2.0105
Epoch 68/100 Iteration: 1700 | Training loss: 1.9360
Epoch 69/100 Iteration: 1710 | Training loss: 1.9816
Epoch 69/100 Iteration: 1720 | Training loss: 1.9631
Epoch 70/100 Iteration: 1730 | Training loss: 1.9588
Epoch 70/100 Iteration: 1740 | Training loss: 2.0093
Epoch 70/100 Iteration: 1750 | Training loss: 1.9281
Epoch 71/100 Iteration: 1760 | Training loss: 1.9703
Epoch 71/100 Iteration: 1770 | Training loss: 1.9591
Epoch 72/100 Iteration: 1780 | Training loss: 1.9504
Epoch 72/100 Iteration: 1790 | Training loss: 1.9990
Epoch 72/100 Iteration: 1800 | Training loss: 1.9055
Epoch 73/100 Iteration: 1810 | Training loss: 1.9748
Epoch 73/100 Iteration: 1820 | Training loss: 1.9503
Epoch 74/100 Iteration: 1830 | Training loss: 1.9516
Epoch 74/100 Iteration: 1840 | Training loss: 1.9871
Epoch 74/100 Iteration: 1850 | Training loss: 1.9077
Epoch 75/100 Iteration: 1860 | Training loss: 1.9720
Epoch 75/100 Iteration: 1870 | Training loss: 1.9495
Epoch 76/100 Iteration: 1880 | Training loss: 1.9331
Epoch 76/100 Iteration: 1890 | Training loss: 1.9587
Epoch 76/100 Iteration: 1900 | Training loss: 1.9137
Epoch 77/100 Iteration: 1910 | Training loss: 1.9579
Epoch 77/100 Iteration: 1920 | Training loss: 1.9313
Epoch 78/100 Iteration: 1930 | Training loss: 1.9291
Epoch 78/100 Iteration: 1940 | Training loss: 1.9677
Epoch 78/100 Iteration: 1950 | Training loss: 1.8985
Epoch 79/100 Iteration: 1960 | Training loss: 1.9579
Epoch 79/100 Iteration: 1970 | Training loss: 1.9348
Epoch 80/100 Iteration: 1980 | Training loss: 1.9300
Epoch 80/100 Iteration: 1990 | Training loss: 1.9650
Epoch 80/100 Iteration: 2000 | Training loss: 1.9060
Epoch 81/100 Iteration: 2010 | Training loss: 1.9508
Epoch 81/100 Iteration: 2020 | Training loss: 1.9254
Epoch 82/100 Iteration: 2030 | Training loss: 1.9186
Epoch 82/100 Iteration: 2040 | Training loss: 1.9547
Epoch 82/100 Iteration: 2050 | Training loss: 1.9006
Epoch 83/100 Iteration: 2060 | Training loss: 1.9368
Epoch 83/100 Iteration: 2070 | Training loss: 1.9200
Epoch 84/100 Iteration: 2080 | Training loss: 1.9243
Epoch 84/100 Iteration: 2090 | Training loss: 1.9619
Epoch 84/100 Iteration: 2100 | Training loss: 1.8790
Epoch 85/100 Iteration: 2110 | Training loss: 1.9302
Epoch 85/100 Iteration: 2120 | Training loss: 1.9224
Epoch 86/100 Iteration: 2130 | Training loss: 1.9097
Epoch 86/100 Iteration: 2140 | Training loss: 1.9423
Epoch 86/100 Iteration: 2150 | Training loss: 1.8794
Epoch 87/100 Iteration: 2160 | Training loss: 1.9215
Epoch 87/100 Iteration: 2170 | Training loss: 1.8930
Epoch 88/100 Iteration: 2180 | Training loss: 1.8956
Epoch 88/100 Iteration: 2190 | Training loss: 1.9315
Epoch 88/100 Iteration: 2200 | Training loss: 1.8665
Epoch 89/100 Iteration: 2210 | Training loss: 1.9205
Epoch 89/100 Iteration: 2220 | Training loss: 1.8885
Epoch 90/100 Iteration: 2230 | Training loss: 1.8877
Epoch 90/100 Iteration: 2240 | Training loss: 1.9299
Epoch 90/100 Iteration: 2250 | Training loss: 1.8701
Epoch 91/100 Iteration: 2260 | Training loss: 1.9057
Epoch 91/100 Iteration: 2270 | Training loss: 1.8868
Epoch 92/100 Iteration: 2280 | Training loss: 1.9071
Epoch 92/100 Iteration: 2290 | Training loss: 1.9250
Epoch 92/100 Iteration: 2300 | Training loss: 1.8551
Epoch 93/100 Iteration: 2310 | Training loss: 1.8961
Epoch 93/100 Iteration: 2320 | Training loss: 1.8781
Epoch 94/100 Iteration: 2330 | Training loss: 1.8918
Epoch 94/100 Iteration: 2340 | Training loss: 1.9246
Epoch 94/100 Iteration: 2350 | Training loss: 1.8591
Epoch 95/100 Iteration: 2360 | Training loss: 1.9072
Epoch 95/100 Iteration: 2370 | Training loss: 1.8782
Epoch 96/100 Iteration: 2380 | Training loss: 1.8703
Epoch 96/100 Iteration: 2390 | Training loss: 1.9196
Epoch 96/100 Iteration: 2400 | Training loss: 1.8436
Epoch 97/100 Iteration: 2410 | Training loss: 1.8911
Epoch 97/100 Iteration: 2420 | Training loss: 1.8705
Epoch 98/100 Iteration: 2430 | Training loss: 1.8792
Epoch 98/100 Iteration: 2440 | Training loss: 1.9072
Epoch 98/100 Iteration: 2450 | Training loss: 1.8423
Epoch 99/100 Iteration: 2460 | Training loss: 1.8873
Epoch 99/100 Iteration: 2470 | Training loss: 1.8705
Epoch 100/100 Iteration: 2480 | Training loss: 1.8644
Epoch 100/100 Iteration: 2490 | Training loss: 1.9047
Epoch 100/100 Iteration: 2500 | Training loss: 1.8314
```

The trained model will be saved in a directory called *./model-100/* so that we can reload it later for prediction or for continuing the training.

6.8 The CharRNN model in the sampling mode

Next up, we can create a new instance of the *CharRNN* class in the sampling mode by specifying that *sampling=True*. We will call the *sample* method to load the saved model in the *./model-100/* folder, and generate a sequence of 500 characters:

```
In [27]: del rnn

         np.random.seed(123)
         rnn = CharRNN(len(chars), sampling=True)
         print(rnn.sample(ckpt_dir='./model-100/', output_length=500))
```

```
INFO:tensorflow:Restoring parameters from ./model-100/language_modeling.ckpt
The seall, in his, in the bese me and mestine:
I worcout in his sollous and shall wish with

   Ham. The same thould to heart, wire thes times in the part
And a the soull the withen, what hiu that in a but
Are a farth mess ant that same that hount,
It the beare andore all thee hash aray astitinnesers,
And, at is the prosing tone tersine,
I mart the morithousse of the pestion to him
Whore than the menthers as oul the bants, thith,
To beere heere, these whilge the hourd, and to my Lord:
If astind the m
```

The generated text will look like the following:

You can see that in the resulting output, that some English words are mostly preserved. It's also important to note that this is from an old English text; therefore, some words in the original text may by unfamiliar. To get a better result, we would need to train the model for higher number of epochs. Feel free to repeat this with a much larger document and train the model for more epochs.

7 Chapter and book summary

We hope you enjoyed this last chapter of *Python Machine Learning* and our exciting tour of machine learning and deep learning. Through the journey of this book, we have covered the essential topics that this field has to offer, and you should now be well equipped to put those techniques into action to solve real-world problems.

We started our journey with a brief overview of the different types of learning tasks: supervised learning, reinforcement learning, and unsupervised learning. We then discussed several different learning algorithms that you can use for classification, starting with simple single-layer neural networks in Chapter 2.

We continued to discuss advanced classification algorithms in Chapter 3, and we learned about the most important aspects of a machine learning pipeline in Chapter 4.

Remember that even the most advanced algorithm is limited by the information in the training data that it gets to learn from. So in Chapter 6, we learned about the best practices to build and evaluate predictive models, which is another important aspect in machine learning applications.

If one single learning algorithm does not achieve the performance we desire, it can be sometimes helpful to create an ensemble of experts to make a prediction. We explored this in Chapter 7.

THen in Chapter 8, we applied machine learning to analyze one of the most popular and interesting forms of data in the modern age that's dominated by social media plataform on the internet - text documents.

Next, we reminded ourselves that machine learning techniques are not limited to offline data analysis, and in Chapter 9, we saw how to embed a machine learning model into a web application to share it with the outside world.

For the most part, our focus was on algorithms for classification, which is probably the most popular application of machine learning. However, this is not where our journey ended! In Chapter 10, we explored several algorithms for regression analysis to predict continuous valued output values.

Another exciting subfield of machine learning is clustering analysis, which can help us find hidden structures in the data, even if our training data does not come with the right answers to learn from. We worked with this in Chapter 11.

We then shifted our attention to one of the most exciting algorithms in the whole machine learning field - artificial neural networks. We started by implementing a multilayer perceptron from scratch with NumPy in Chapter 12.

The power of TensorFlow became obvious in Chapter 13, where we used TensorFlow to facilitate the process of build neural network models and make use of GPUs to make the training of multilayer neural networks more efficient.

We delved deeper into the mechanics of TensorFlow in Chapter 14, and discussed the different aspects and mechanics of TensorFlow, including variables and operators in a TensorFlow computation graph, variables scopes, launching graphs, and different ways of executing nodes.

In Chapter 15, we dived into convolutional neural networks, which are widely usde in computer vision at the moment, due to their great performance in image classification tasks.

Finally, here in Chapter 16, we learned about sequence modeling using RNNs. While a comprehensive study of deep learning is well beyond the scope of this book, we hope that we have kindled your interest enough to follow the most recent advancements in this field of deep learning.