16
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 sequential 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 exploding gradients

Since this chapter is the last in our *Python Machine Learning* journey, we'll conclude with a summary of what we've 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'll 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.

Introducing sequential data

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

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, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, 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.

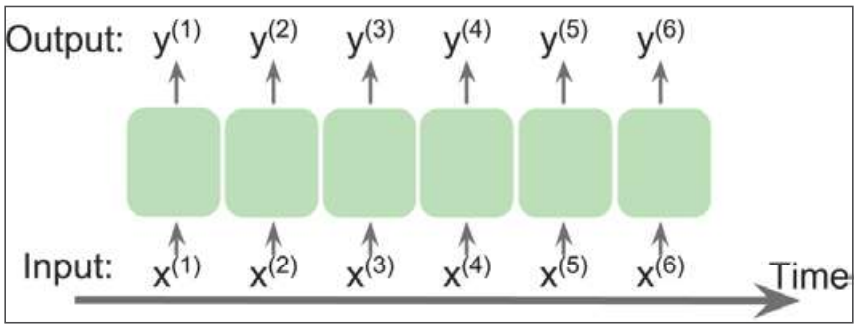
However, this assumption is not valid anymore when we deal with sequences—by definition, order matters.

Representing sequences

We've established 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 $\left(x^{(1)}, x^{(2)}, \ldots, x^{(T)}\right)$. 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 $\boldsymbol{x}^{(t)}$ 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:



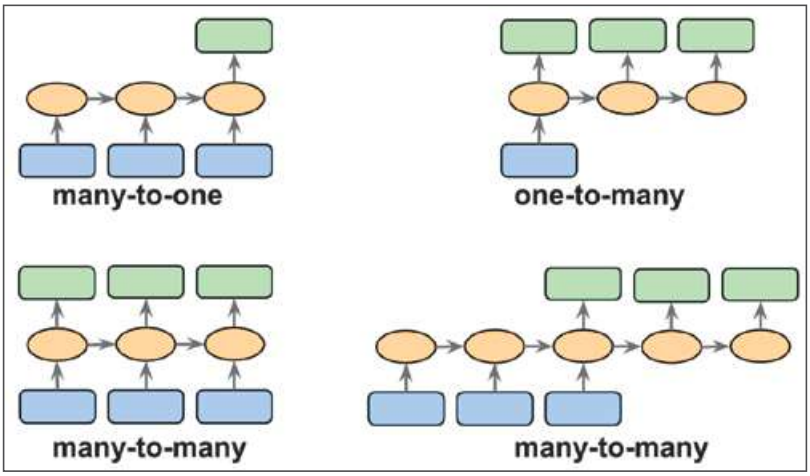
The standard neural network models that we have covered so far, such as MLPs and CNNs, are not capable of handling *the order* of input samples. Intuitively, one can say that such models do not have a *memory* of the past seen samples. For instance, the samples are passed through the feedforward and 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.

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, based on the explanations in the excellent article *The Unreasonable Effectiveness of Recurrent Neural Networks* by Andrej Karpathy (`http://karpathy.github.io/2015/05/21/rnn-effectiveness/`), 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 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.

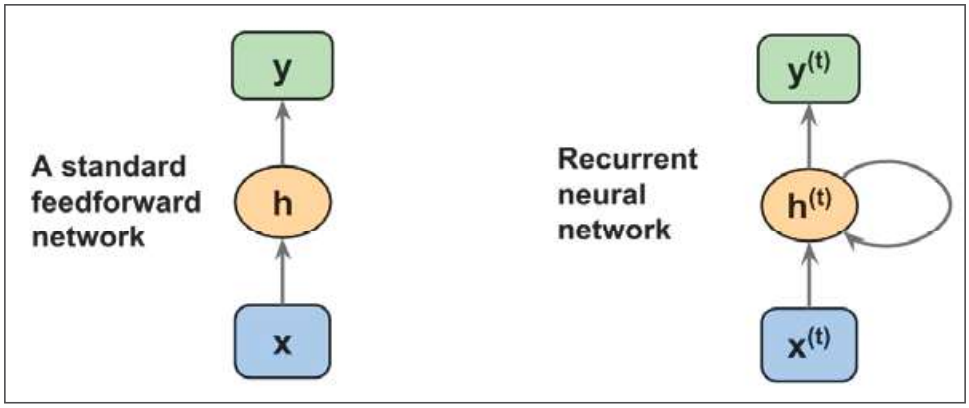
Now, since we know about the categories of sequence modeling, we can move forward to discuss the structure of an RNN.

RNNs for modeling sequences

In this section, now that we understand sequences, we can look at the foundations of RNNs. We'll start by introducing the typical structure of an RNN, and we'll see how the data flows through it with one or more hidden layers. We'll 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.

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 $\boldsymbol{y}^{(t)}$ as the final output, or it could be many-to-one if, for example, we only use the last element of $\boldsymbol{y}^{(t)}$ as the final output.

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

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 $\boldsymbol{x}^{(t)}$ and the previous values of hidden units at $t - 1$, indicated as $\boldsymbol{h}^{(t-1)}$.

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 $\boldsymbol{h}_1^{(t)}$ and gets its input from the data point $\boldsymbol{x}^{(t)}$ and the hidden values in the same layer, but the previous time step $\boldsymbol{h}_1^{(t-1)}$
- $layer = 2$: The second hidden layer, $\boldsymbol{h}_2^{(t)}$ receives its inputs from the hidden units from the layer below at the current time step ($\boldsymbol{h}_1^{(t)}$) and its own hidden values from the previous time step $\boldsymbol{h}_2^{(t-1)}$

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'll 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:

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

Training RNNs using BPTT

The learning algorithm for RNNs was introduced in 1990s *Backpropagation Through Time: What It Does and How to Do It* (*Paul Werbos, Proceedings of IEEE*, 78(10):1550-1560, *1990*).

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 times $t=1$ to $t=T$:

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

Since the loss at time $1:t$ is dependent on the hidden units at all previous time steps $1:t$, the gradient will be computed as follows:

$$\frac{\partial L^{(t)}}{\partial \boldsymbol{W}_{hh}}=\frac{\partial L^{(t)}}{\partial \boldsymbol{y}^{(t)}}\times\frac{\partial \boldsymbol{y}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\times\left(\sum_{k=1}^{t}\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}\times\frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}_{hh}}\right)$$

Here, $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$ is computed as a multiplication of adjacent time steps:

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}=\prod_{i=k+1}^{t}\frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}$$

The challenges of learning long-range interactions

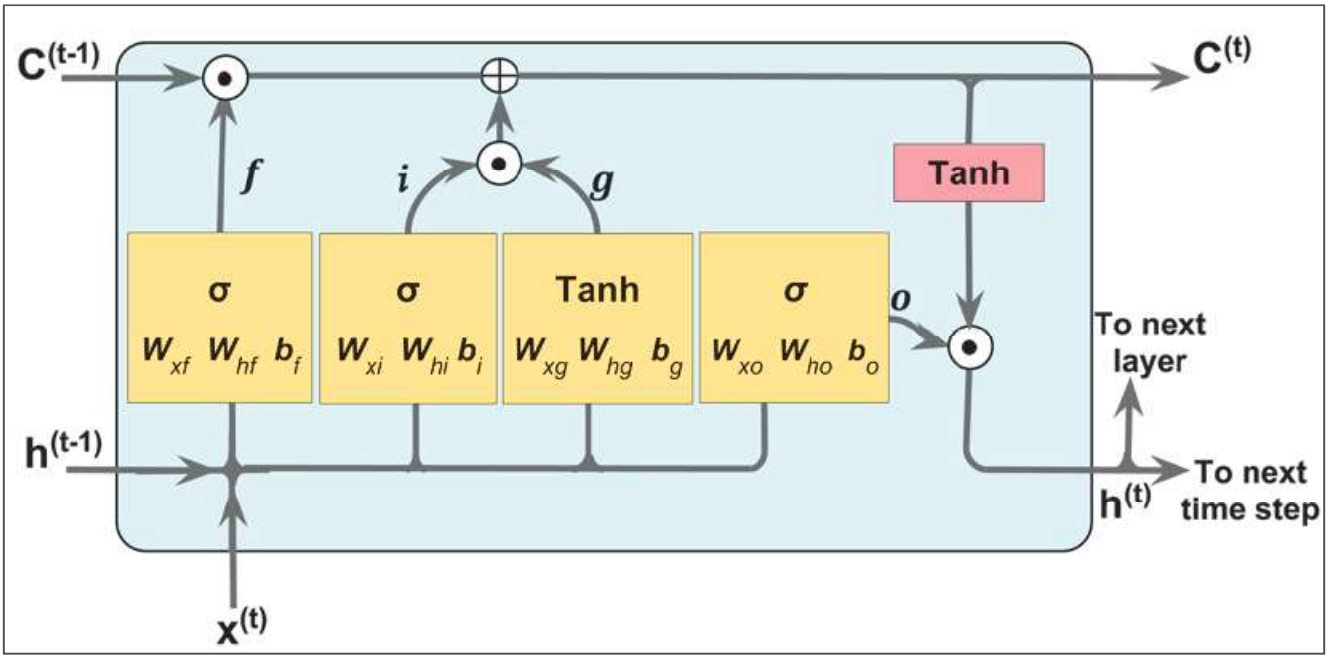
Backpropagation through time, or BPTT, which we briefly mentioned in the previous information box, introduces some new challenges.

Because of the multiplicative factor $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$ 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:

LSTM units

LSTMs were first introduced to overcome the vanishing gradient problem (*Long Short-Term Memory*, *S. Hochreiter* and *J. Schmidhuber*, *Neural Computation*, 9(8): 1735-1780, *1997*). 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, $\boldsymbol{C}^{(t-1)}$, is modified to get the cell state at the current time step, $\boldsymbol{C}^{(t)}$, 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'll 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, $\boldsymbol{x}^{(t)}$ refers to the input data at time *t*, and $\boldsymbol{h}^{(t-1)}$ indicates the hidden units at time $t-1$.

Given this, the hidden units at the current time step are computed as follows:

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{C}^{(t)}\right)$$

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 cells easily. We'll 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 (*An Empirical Exploration of Recurrent Network Architectures, Rafal Jozefowicz, Wojciech Zaremba*, and *Ilya Sutskever, Proceedings of ICML*, 2342-2350, *2015*).

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. If you are interested in learning more about these modern RNN architectures, refer to the paper, *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling* by *Junyoung Chung and others 2014* (`https://arxiv.org/pdf/1412.3555v1.pdf`).

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'll 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. If you like to have a big picture overview and see all the code at once before diving into the discussion, we recommend you to take a look at the code implementation first, which you can view at `https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch16/ch16.ipynb`.

Note, before we start coding in this chapter, that since we're using a very modern build of TensorFlow, we'll be using code from the `contrib` submodule of TensorFlow's Python API, in the latest version of TensorFlow (1.3.0) from 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 an eye on the TensorFlow API documentation (`https://www.tensorflow.org/api_docs/python/`) to be updated with the latest version details, in particular, if you have any problems using the `tf.contrib` code described in this chapter.

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

You may recall from *Chapter 8, Applying Machine Learning to Sentiment Analysis*, that sentiment analysis is concerned with analyzing the expressed opinion of a sentence or a 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 a 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.

Preparing the data

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

```
>>> 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 done 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. If you want to learn more about `Counter`, refer to its documentation at `https://docs.python.org/3/library/collections.html#collections.Counter`.

In the following code, we will define a `counts` object 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 model), 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 word, 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:

```
>>> ## Preprocessing the data:
>>> ## Separate words and
>>> ## count each word's occurrence
```

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 naïve 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 the curse of dimensionality
- The extraction of salient features since the embedding layer in a neural network is trainable

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]:

```
embedding = tf.Variable(
                tf.random_uniform(
                    shape=(n_words, embedding_size),
                    minval=-1, maxval=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`:

```
embed_x = tf.nn.embedding_lookup(embedding, 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. Feel free to read more about this function from its official documentation at `https://www.tensorflow.org/api_docs/python/tf/nn/embedding_lookup`.

Building an RNN model

Now we're ready to build an RNN model. We'll 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'll see the implementation of this class and its methods broken into separate code sections.

The SentimentRNN class constructor

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

```
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()
```

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's 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've used here, and it determines the number of hidden units in each RNN layer.

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'll be going through it in plenty of detail. First, we'll look at the code in full, so we can see everything together, and then we'll analyze each of its main parts:

```
    def build(self):
        ## Define the placeholders
        tf_x = tf.placeholder(tf.int32,
                    shape=(self.batch_size, self.seq_len),
                    name='tf_x')
        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),
                        minval=-1, maxval=1),
                    name='embedding')
        embed_x = tf.nn.embedding_lookup(
                    embedding, tf_x,
                    name='embeded_x')

        ## Define LSTM cell and stack them together
        cells = tf.contrib.rnn.MultiRNNCell(
                [tf.contrib.rnn.DropoutWrapper(
                   tf.contrib.rnn.BasicLSTMCell(self.lstm_size),
                   output_keep_prob=tf_keepprob)
                 for i in range(self.num_layers)])

        ## 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=self.initial_state)

        ## Note: lstm_outputs shape:
```

Step 1 – defining multilayer RNN cells

To examine how we coded 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 one cell.

You can read more about these functions at the following links:

- `tf.contrib.rnn.BasicLSTMCell:https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/BasicLSTMCell`
- `tf.contrib.rnn.DropoutWrapper: https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/DropoutWrapper`
- `tf.contrib.rnn.MultiRNNCell: https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/MultiRNNCell`

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'll recall from the architecture of LSTM cells, there are three types of inputs in an LSTM cell—input data $\boldsymbol{x}^{(t)}$, activations of hidden units from the previous time step $\boldsymbol{h}^{(t-1)}$, and the cell state from the previous time step $\boldsymbol{C}^{(t-1)}$.

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.

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.nn.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.nn.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.

Feel free to read more about the `tf.nn.dynamic_rnn` function at its official documentation page at `https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn`.

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.

The train method

The next method in our `SentimentRNN` class is `train`. This method call is quite similar to the train methods we created in *Chapter 14, Going Deeper – The Mechanics of TensorFlow* and *Chapter 15, Classifying Images with Deep Convolutional Neural Networks* except that we have an additional tensor, `state`, that we feed into our network.

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:

```
    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=self.batch_size), 1):
                feed = {'tf_x:0' : batch_x,
                        'tf_keepprob:0' : 1.0,
                        self.initial_state : test_state}
                if return_proba:
                    pred, test_state = sess.run(
                        ['probabilities:0', self.final_state],
                        feed_dict=feed)
                else:
                    pred, test_state = sess.run(
                        ['labels:0', self.final_state],
                        feed_dict=feed)

                preds.append(pred)

        return np.concatenate(preds)
```

Instantiating the SentimentRNN class

We've now coded and examined all four parts of our `SentimentRNN` class, which were the class constructor, the `build` method, the `train` method, and the `predict` method.

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

```
>>> 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)
```

Notice here that we use `num_layers=1` to use a single RNN layer. Although our implementation allows us to create 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.

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`:

```
>>> rnn.train(X_train, y_train, num_epochs=40)
Epoch: 1/40 Iteration: 20 | Train loss: 0.70637
Epoch: 1/40 Iteration: 40 | Train loss: 0.60539
Epoch: 1/40 Iteration: 60 | Train loss: 0.66977
Epoch: 1/40 Iteration: 80 | Train loss: 0.51997
...
```

The trained model is saved using TensorFlow's checkpointing system, which we discussed in *Chapter 14*, *Going Deeper – The Mechanics of TensorFlow*. Now, we can use the trained model for predicting the class labels on the test set, as follows:

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

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*, *Applying Machine Learning to Sentiment Analysis*.

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 don't repeatedly use the test set for evaluation to avoid introducing bias through test data leakage, which we discussed in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

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

```
>>> proba = rnn.predict(X_test, return_proba=True)
```

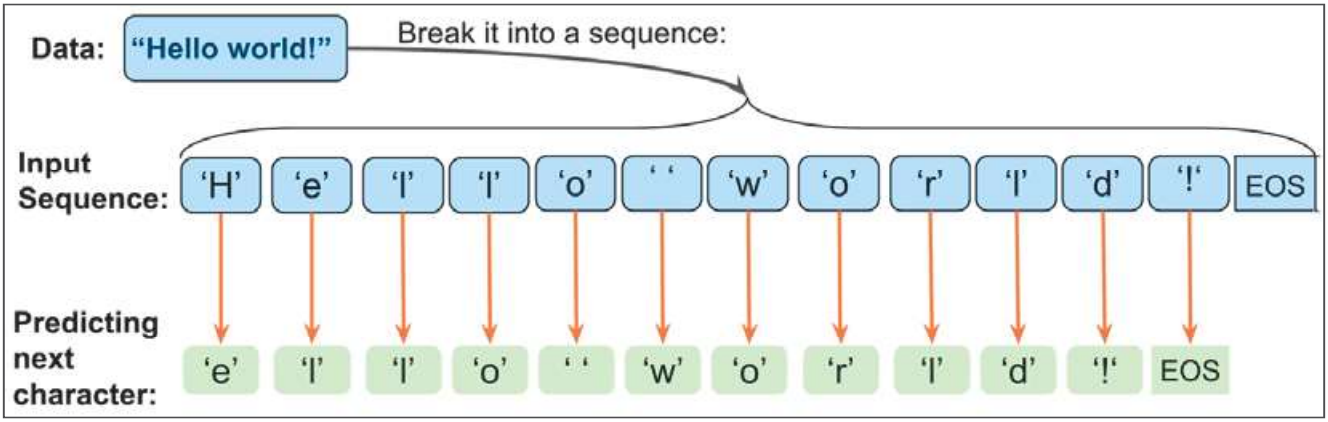
So this was our first RNN model for sentiment analysis. We'll now go further and create an RNN for character-by-character language modeling in TensorFlow, as another popular application of sequence modeling.

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. One of the interesting efforts in this area is the work done by Sutskever, Martens, and Hinton (*Generating Text with Recurrent Neural Networks, Ilya Sutskever, James Martens*, and *Geoffrey E. Hinton, Proceedings of the 28th International Conference on Machine Learning (ICML-11), 2011* `https://pdfs.semanticscholar.org/93c2/0e38c85b69fc2d2eb314b3c1217913f7db11.pdf`).

In the model that we'll 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 a time. The network will process each new character in conjunction with the memory of the previously seen characters to predict the next 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'll also get a chance to play with a gradient clipping technique to avoid this exploding gradient problem.

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`.

Note that this link will directly take you to the download page. If you are using macOS or a Linux operating system, you can download the file with the following command in the Terminal:

```
curl http://www.gutenberg.org/cache/epub/2265/pg2265.txt > pg2265.txt
```

If this resource becomes unavailable in future, a copy of this text is also included in this chapter's code directory in the book's code repository at `https://github.com/rasbt/python-machine-learning-book-2nd-edition`.

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:

```
>>> 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 point, we've now completed the data preprocessing steps, and we have the data in the proper format. In the next section, we'll implement the RNN model for character-level language modeling.

Building a character-level RNN model

To build a character-level neural network, we'll 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'll 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 – performing sentiment analysis of IMDb movie reviews using multilayer RNNs* section. So, we'll skip the description of building the RNN components here.

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're 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:

```
import tensorflow as tf
import os

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()
```

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've 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.

The build method

The next method of the `CharRNN` class is `build`, which is very similar to the `build` method in the *Project one – performing sentiment analysis of IMDb movie reviews using multilayer RNNs* section, except for some minor differences. The `build` method first defines two local variables, `batch_size` and `num_steps`, based on the mode, as follows:

$$in\ sampling\ \ mode: \begin{cases} batch\_size = 1 \\ num\_steps = 1 \end{cases}$$

$$in\ training\ \ mode: \begin{cases} batch\_size = self.batch\_size \\ num\_steps = self.num\_steps \end{cases}$$

Recall that in the sentiment analysis implementation, we used an embedding layer to create a salient representation for 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` from 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.

```
                shape=[-1, self.lstm_size],
                name='seq_output_reshaped')

        logits = tf.layers.dense(
                inputs=seq_output_reshaped,
                units=self.num_classes,
                activation=None,
                name='logits')

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

        y_reshaped = tf.reshape(
                y_onehot,
                shape=[-1, self.num_classes],
                name='y_reshaped')
        cost = tf.reduce_mean(
                tf.nn.softmax_cross_entropy_with_logits(
                    logits=logits,
                    labels=y_reshaped),
                name='cost')

        # 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')
```

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 – performing sentiment analysis of IMDb movie reviews using multilayer RNNs* section. Here is the `train` method code, which will look very familiar to the sentiment analysis version we built earlier in this chapter:

```
    def train(self, train_x, train_y,
              num_epochs, ckpt_dir='./model/'):
        ## Create the checkpoint directory
```

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 – performing sentiment analysis of IMDb movie reviews using multilayer RNNs* section. 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:

```
    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_dict=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_dict=feed)

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

        return ''.join(observed_seq)
```

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:

```
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
```

Note, of course, that this function should be defined *before* the definition of the `CharRNN` class; we've explained it in this order here so that we can explain the concepts in order. Browse through the code notebook that accompanies this chapter to get a better overview of the order in which the functions are defined.

Creating and training the CharRNN Model

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

```
>>> 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/')
```

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.

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'll call the `sample` method to load the saved model in the `./model-100/` folder, and generate a sequence of 500 characters:

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

The generated text will look like the following:

```
The stall soues tay and the hates,
The perse in there is that so the meanes this made there

   Ham. Ile teath thes are this makere of a driane,
Why shis mestend the Casst of is singe,
In this to this, to mers it is for marth,
Ase hinees sim thig tald ow a tore andere,
In histhene tistere shere this wile and my Lord:
And tit mighes the secleer allost heruen, and that hash to sall and hears,
If you his moses tonger and mout ofr mesting a forte tis at

   Pomin. Where in you dist and sintere shan shall
```

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 be 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.

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've 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.