Chapter 28

Project: Text Generation With Alice in Wonderland

Recurrent neural networks can also be used as generative models. This means that in addition to being used for predictive models (making predictions) they can learn the sequences of a problem and then generate entirely new plausible sequences for the problem domain. Generative models like this are useful not only to study how well a model has learned a problem, but to learn more about the problem domain itself. In this project you will discover how to create a generative model for text, character-by-character using LSTM recurrent neural networks in Python with Keras. After completing this project you will know:

- Where to download a free corpus of text that you can use to train text generative models.
- How to frame the problem of text sequences to a recurrent neural network generative model.
- How to develop an LSTM to generate plausible text sequences for a given problem.

Let's get started.

Note: You may want to speed up the computation for this tutorial by using GPU rather than CPU hardware, such as the process described in Chapter 5. This is a suggestion, not a requirement. The tutorial will work just fine on the CPU.

28.1 Problem Description: Text Generation

Many of the classical texts are no longer protected under copyright. This means that you can download all of the text for these books for free and use them in experiments, like creating generative models. Perhaps the best place to get access to free books that are no longer protected by copyright is Project Gutenberg[1]. In this tutorial we are going to use a favorite book from childhood as the dataset: Alice's Adventures in Wonderland by Lewis Carroll[2].

We are going to learn the dependencies between characters and the conditional probabilities of characters in sequences so that we can in turn generate wholly new and original sequences of characters. This tutorial is a lot of fun and I recommend repeating these experiments with

[1] https://www.gutenberg.org/

[2] https://www.gutenberg.org/ebooks/11

other books from Project Gutenberg. These experiments are not limited to text, you can also experiment with other ASCII data, such as computer source code, marked up documents in LaTeX, HTML or Markdown and more.

You can download the complete text in ASCII format (Plain Text UTF-8)[3] for this book for free and place it in your working directory with the filename `wonderland.txt`. Now we need to prepare the dataset ready for modeling. Project Gutenberg adds a standard header and footer to each book and this is not part of the original text. Open the file in a text editor and delete the header and footer. The header is obvious and ends with the text:

```
*** START OF THIS PROJECT GUTENBERG EBOOK ALICE'S ADVENTURES IN WONDERLAND ***
```

Listing 28.1: Signal of the End of the File Header.

The footer is all of the text after the line of text that says:

```
THE END
```

Listing 28.2: Signal of the Start of the File Footer.

You should be left with a text file that has about 3,330 lines of text.

28.2 Develop a Small LSTM Recurrent Neural Network

In this section we will develop a simple LSTM network to learn sequences of characters from Alice in Wonderland. In the next section we will use this model to generate new sequences of characters. Let's start off by importing the classes and functions we intend to use to train our model.

```python
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
```

Listing 28.3: Import Classes and Functions.

Next we need to load the ASCII text for the book into memory and convert all of the characters to lowercase to reduce the vocabulary that the network must learn.

```python
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
```

Listing 28.4: Load the Dataset and Covert to Lowercase.

Now that the book is loaded, we must prepare the data for modeling by the neural network. We cannot model the characters directly, instead we must convert the characters to integers. We can do this easily by first creating a set of all of the distinct characters in the book, then creating a map of each character to a unique integer.

[3] http://www.gutenberg.org/cache/epub/11/pg11.txt

```
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
```

Listing 28.5: Create Char-to-Integer Mapping.

For example, the list of unique sorted lowercase characters in the book is as follows:

```
['\n', '\r', ' ', '!', '"', "'", '(', ')', '*', ',', '-', '.', ':', ';', '?', '[', ']',
    '_', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
    'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '\xbb', '\xbf', '\xef']
```

Listing 28.6: List of Unique Characters in the Dataset.

You can see that there may be some characters that we could remove to further clean up the dataset that will reduce the vocabulary and may improve the modeling process. Now that the book has been loaded and the mapping prepared, we can summarize the dataset.

```
n_chars = len(raw_text)
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
```

Listing 28.7: Summarize the Loaded Dataset.

Running the code to this point produces the following output.

```
Total Characters: 147674
Total Vocab: 47
```

Listing 28.8: Output from Summarize the Dataset.

We can see that the book has just under 150,000 characters and that when converted to lowercase that there are only 47 distinct characters in the vocabulary for the network to learn. Much more than the 26 in the alphabet. We now need to define the training data for the network. There is a lot of flexibility in how you choose to break up the text and expose it to the network during training. In this tutorial we will split the book text up into subsequences with a fixed length of 100 characters, an arbitrary length. We could just as easily split the data up by sentences and pad the shorter sequences and truncate the longer ones.

Each training pattern of the network is comprised of 100 time steps of one character (`X`) followed by one character output (`y`). When creating these sequences, we slide this window along the whole book one character at a time, allowing each character a chance to be learned from the 100 characters that preceded it (except the first 100 characters of course). For example, if the sequence length is 5 (for simplicity) then the first two training patterns would be as follows:

```
CHAPT -> E
HAPTE -> R
```

Listing 28.9: Example of Sequence Construction.

As we split up the book into these sequences, we convert the characters to integers using our lookup table we prepared earlier.

```
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
```

```
dataY = []
for i in range(0, n_chars - seq_length, 1):
  seq_in = raw_text[i:i + seq_length]
  seq_out = raw_text[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
```

Listing 28.10: Create Input/Output Patterns from Raw Dataset.

Running the code to this point shows us that when we split up the dataset into training data for the network to learn that we have just under 150,000 training pattens. This makes sense as excluding the first 100 characters, we have one training pattern to predict each of the remaining characters.

```
Total Patterns: 147574
```

Listing 28.11: Output Summary of Created Patterns.

Now that we have prepared our training data we need to transform it so that it is suitable for use with Keras. First we must transform the list of input sequences into the form [samples, time steps, features] expected by an LSTM network. Next we need to rescale the integers to the range 0-to-1 to make the patterns easier to learn by the LSTM network that uses the sigmoid activation function by default.

Finally, we need to convert the output patterns (single characters converted to integers) into a one hot encoding. This is so that we can configure the network to predict the probability of each of the 47 different characters in the vocabulary (an easier representation) rather than trying to force it to predict precisely the next character. Each y value is converted into a sparse vector with a length of 47, full of zeros except with a 1 in the column for the letter (integer) that the pattern represents. For example, when `n` (integer value 31) is one hot encoded it looks as follows:

```
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.]
```

Listing 28.12: Sample of One Hot Encoded Output Integer.

We can implement these steps as below.

```
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
```

Listing 28.13: Prepare Data Ready For Modeling.

We can now define our LSTM model. Here we define a single hidden LSTM layer with 256 memory units. The network uses dropout with a probability of 20%. The output layer is a Dense layer using the softmax activation function to output a probability prediction for each of the 47 characters between 0 and 1. The problem is really a single character classification problem with 47 classes and as such is defined as optimizing the log loss (cross entropy), here using the ADAM optimization algorithm for speed.

```
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Listing 28.14: Create LSTM Network to Model the Dataset.

There is no test dataset. We are modeling the entire training dataset to learn the probability of each character in a sequence. We are not interested in the most accurate (classification accuracy) model of the training dataset. This would be a model that predicts each character in the training dataset perfectly. Instead we are interested in a generalization of the dataset that minimizes the chosen loss function. We are seeking a balance between generalization and overfitting but short of memorization.

The network is slow to train (about 300 seconds per epoch on an Nvidia K520 GPU). Because of the slowness and because of our optimization requirements, we will use model checkpointing to record all of the network weights to file each time an improvement in loss is observed at the end of the epoch. We will use the best set of weights (lowest loss) to instantiate our generative model in the next section.

```
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True,
    mode='min')
callbacks_list = [checkpoint]
```

Listing 28.15: Create Checkpoints For Best Seen Model.

We can now fit our model to the data. Here we use a modest number of 20 epochs and a large batch size of 128 patterns.

```
model.fit(X, y, epochs=20, batch_size=128, callbacks=callbacks_list)
```

Listing 28.16: Fit the Model.

The full code listing is provided below for completeness.

```
# Small LSTM Network to Generate Text for Alice in Wonderland
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
```

```
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
  seq_in = raw_text[i:i + seq_length]
  seq_out = raw_text[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True,
    mode='min')
callbacks_list = [checkpoint]
# fit the model
model.fit(X, y, epochs=20, batch_size=128, callbacks=callbacks_list)
```

Listing 28.17: Complete Code Listing for LSTM to Model Dataset.

You will see different results because of the stochastic nature of the model, and because it is hard to fix the random seed for LSTM models to get 100% reproducible results. This is not a concern for this generative model. After running the example, you should have a number of weight checkpoint files in the local directory. You can delete them all except the one with the smallest loss value. For example, when I ran this example, below was the checkpoint with the smallest loss that I achieved.

```
weights-improvement-19-1.9435.hdf5
```

Listing 28.18: Sample of Checkpoint Weights for Well Performing Model.

The network loss decreased almost every epoch and I expect the network could benefit from training for many more epochs. In the next section we will look at using this model to generate new text sequences.

28.3 Generating Text with an LSTM Network

Generating text using the trained LSTM network is relatively straightforward. Firstly, we load the data and define the network in exactly the same way, except the network weights are loaded from a checkpoint file and the network does not need to be trained.

Note: It seems that you might need to use the same machine/environment to generate text as was used to create the network weights (e.g. GPUs or CPUs), otherwise the network might just generate garbage.

```
# load the network weights
filename = "weights-improvement-19-1.9435.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Listing 28.19: Load Checkpoint Network Weights.

Also, when preparing the mapping of unique characters to integers, we must also create a reverse mapping that we can use to convert the integers back to characters so that we can understand the predictions.

```
int_to_char = dict((i, c) for i, c in enumerate(chars))
```

Listing 28.20: Mapping from Integers to Characters.

Finally, we need to actually make predictions. The simplest way to use the Keras LSTM model to make predictions is to first start off with a seed sequence as input, generate the next character then update the seed sequence to add the generated character on the end and trim off the first character. This process is repeated for as long as we want to predict new characters (e.g. a sequence of 1,000 characters in length). We can pick a random input pattern as our seed sequence, then print generated characters as we generate them.

```
# pick a random seed
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print("Seed:")
print("\"", ''.join([int_to_char[value] for value in pattern]), "\"")
# generate characters
for i in range(1000):
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(n_vocab)
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
  sys.stdout.write(result)
  pattern.append(index)
  pattern = pattern[1:len(pattern)]
print("\nDone.")
```

Listing 28.21: Seed Network and Generate Text.

The full code example for generating text using the loaded LSTM model is listed below for completeness.

```
# Load LSTM network and generate text
import sys
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.utils import np_utils
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
int_to_char = dict((i, c) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
  seq_in = raw_text[i:i + seq_length]
  seq_out = raw_text[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
# load the network weights
filename = "weights-improvement-19-1.9435.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
# pick a random seed
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print("Seed:")
print("\"", ''.join([int_to_char[value] for value in pattern]), "\"")
# generate characters
for i in range(1000):
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(n_vocab)
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
  sys.stdout.write(result)
  pattern.append(index)
  pattern = pattern[1:len(pattern)]
print("\nDone.")
```

Listing 28.22: Complete Code Listing for Generating Text for the Small LSTM Network.

Running this example first outputs the selected random seed, then each character as it is generated. For example, below are the results from one run of this text generator. The random seed was:

```
be no mistake about it: it was neither more nor less than a pig, and she
felt that it would be quit
```

Listing 28.23: Randomly Selected Sequence Used to Seed the Network.

The generated text with the random seed (cleaned up for presentation) was:

```
be no mistake about it: it was neither more nor less than a pig, and she
felt that it would be quit e aelin that she was a little want oe toiet
ano a grtpersent to the tas a little war th tee the tase oa teettee
the had been tinhgtt a little toiee at the cadl in a long tuiee aedun
thet sheer was a little tare gereen to be a gentle of the tabdit soenee
the gad ouw ie the tay a tirt of toiet at the was a little
anonersen, and thiu had been woite io a lott of tueh a tiie and taede
bot her aeain she cere thth the bene tith the tere bane to tee
toaete to tee the harter was a little tire the same oare cade an anl ano
the garee and the was so seat the was a little gareen and the sabdit,
and the white rabbit wese tilel an the caoe and the sabbit se teeteer,
and the white rabbit wese tilel an the cade in a lonk tfne the sabdi
ano aroing to tea the was sf teet whitg the was a little tane oo thete
the sabeit she was a little tartig to the tar tf tee the tame of the
cagd, and the white rabbit was a little toiee to be anle tite thete ofs
and the tabdit was the wiite rabbit, and
```

Listing 28.24: Generated Text With Random Seed Text.

We can note some observations about the generated text.

- It generally conforms to the line format observed in the original text of less than 80 characters before a new line.
- The characters are separated into word-like groups and most groups are actual English words (e.g. *the*, *little* and *was*), but many do not (e.g. *lott*, *tiie* and *taede*).
- Some of the words in sequence make sense(e.g. *and the white rabbit*), but many do not (e.g. *wese tilel*).

The fact that this character based model of the book produces output like this is very impressive. It gives you a sense of the learning capabilities of LSTM networks. The results are not perfect. In the next section we look at improving the quality of results by developing a much larger LSTM network.

28.4 Larger LSTM Recurrent Neural Network

We got results, but not excellent results in the previous section. Now, we can try to improve the quality of the generated text by creating a much larger network. We will keep the number of memory units the same at 256, but add a second layer.

```
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Listing 28.25: Define a Stacked LSTM Network.

We will also change the filename of the checkpointed weights so that we can tell the difference between weights for this network and the previous (by appending the word `bigger` in the filename).

```
filepath="weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
```

Listing 28.26: Filename for Checkpointing Network Weights for Larger Model.

Finally, we will increase the number of training epochs from 20 to 50 and decrease the batch size from 128 to 64 to give the network more of an opportunity to be updated and learn. The full code listing is presented below for completeness.

```
# Larger LSTM Network to Generate Text for Alice in Wonderland
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
  seq_in = raw_text[i:i + seq_length]
  seq_out = raw_text[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
```

```
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True,
    mode='min')
callbacks_list = [checkpoint]
# fit the model
model.fit(X, y, epochs=50, batch_size=64, callbacks=callbacks_list)
```

Listing 28.27: Complete Code Listing for the Larger LSTM Network.

Running this example takes some time, at least 700 seconds per epoch. After running this example, you may achieve a loss of about 1.2. For example the best result I achieved from running this model was stored in a checkpoint file with the name:

```
weights-improvement-47-1.2219-bigger.hdf5
```

Listing 28.28: Sample of Checkpoint Weights for Well Performing Larger Model.

Achieving a loss of 1.2219 at epoch 47. As in the previous section, we can use this best model from the run to generate text. The only change we need to make to the text generation script from the previous section is in the specification of the network topology and from which file to seed the network weights. The full code listing is provided below for completeness.

```
# Load Larger LSTM network and generate text
import sys
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.utils import np_utils
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
int_to_char = dict((i, c) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
```

```python
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
  seq_in = raw_text[i:i + seq_length]
  seq_out = raw_text[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
# load the network weights
filename = "weights-improvement-47-1.2219-bigger.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
# pick a random seed
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print("Seed:")
print("\"", ''.join([int_to_char[value] for value in pattern]), "\"")
# generate characters
for i in range(1000):
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(n_vocab)
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
  sys.stdout.write(result)
  pattern.append(index)
  pattern = pattern[1:len(pattern)]
print("\nDone.")
```

Listing 28.29: Complete Code Listing for Generating Text With the Larger LSTM Network.

One example of running this text generation script produces the output below. The randomly chosen seed text was:

```
d herself lying on the bank, with her
head in the lap of her sister, who was gently brushing away s
```

Listing 28.30: Randomly Selected Sequence Used to Seed the Network.

The generated text with the seed (cleaned up for presentation) was:

```
herself lying on the bank, with her
head in the lap of her sister, who was gently brushing away
so siee, and she sabbit said to herself and the sabbit said to herself and the sood
way of the was a little that she was a little lad good to the garden,
and the sood of the mock turtle said to herself, 'it was a little that
the mock turtle said to see it said to sea it said to sea it say it
the marge hard sat hn a little that she was so sereated to herself, and
she sabbit said to herself, 'it was a little little shated of the sooe
of the coomouse it was a little lad good to the little gooder head. and
said to herself, 'it was a little little shated of the mouse of the
good of the courte, and it was a little little shated in a little that
the was a little little shated of the thmee said to see it was a little
book of the was a little that she was so sereated to hare a little the
began sitee of the was of the was a little that she was so seally and
the sabbit was a little lad good to the little gooder head of the gad
seared to see it was a little lad good to the little good
```

Listing 28.31: Generated Text With Random Seed Text.

We can see that generally there are fewer spelling mistakes and the text looks more realistic, but is still quite nonsensical. For example the same phrases get repeated again and again like *said to herself* and *little*. Quotes are opened but not closed. These are better results but there is still a lot of room for improvement.

28.5 Extension Ideas to Improve the Model

Below are a sample of ideas that you may want to investigate to further improve the model:

- Predict fewer than 1,000 characters as output for a given seed.
- Remove all punctuation from the source text, and therefore from the models' vocabulary.
- Try a one hot encoded for the input sequences.
- Train the model on padded sentences rather than random sequences of characters.
- Increase the number of training epochs to 100 or many hundreds.
- Add dropout to the visible input layer and consider tuning the dropout percentage.
- Tune the batch size, try a batch size of 1 as a (very slow) baseline and larger sizes from there.
- Add more memory units to the layers and/or more layers.
- Experiment with scale factors (temperature) when interpreting the prediction probabilities.
- Change the LSTM layers to be *stateful* to maintain state across batches.

28.6 Summary

In this project you discovered how you can develop an LSTM recurrent neural network for text generation in Python with the Keras deep learning library. After completing this project you know:

- Where to download the ASCII text for classical books for free that you can use for training.
- How to train an LSTM network on text sequences and how to use the trained network to generate new sequences.
- How to develop stacked LSTM networks and lift the performance of the model.

28.6.1 Next

This tutorial concludes Part VI and your introduction to recurrent neural networks in Keras. Next in Part VII we will conclude this book and highlight additional resources that you can use to continue your studies.