Chapter 27

Understanding Stateful LSTM Recurrent Neural Networks

A powerful and popular recurrent neural network is the long short-term model network or LSTM. It is widely used because the architecture overcomes the vanishing and exploding gradient problem that plagues all recurrent neural networks, allowing very large and very deep networks to be created. Like other recurrent neural networks, LSTM networks maintain state, and the specifics of how this is implemented in Keras framework can be confusing. In this lesson you will discover exactly how state is maintained in LSTM networks by the Keras deep learning library. After reading this lesson you will know:

- How to develop a naive LSTM network for a sequence prediction problem.
- How to carefully manage state through batches and features with an LSTM network.
- Hot to manually manage state in an LSTM network for stateful prediction.

Let's get started.

27.1 Problem Description: Learn the Alphabet

In this tutorial we are going to develop and contrast a number of different LSTM recurrent neural network models. The context of these comparisons will be a simple sequence prediction problem of learning the alphabet. That is, given a letter of the alphabet, predict the next letter of the alphabet. This is a simple sequence prediction problem that once understood can be generalized to other sequence prediction problems like time series prediction and sequence classification. Let's prepare the problem with some Python code that we can reuse from example to example. Firstly, let's import all of the classes and functions we plan to use in this tutorial.

```
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
```

Listing 27.1: Import Classes and Functions.

Next, we can seed the random number generator to ensure that the results are the same each time the code is executed.

```
# fix random seed for reproducibility
numpy.random.seed(7)
```

Listing 27.2: Seed the Random Number Generators.

We can now define our dataset, the alphabet. We define the alphabet in uppercase characters for readability. Neural networks model numbers, so we need to map the letters of the alphabet to integer values. We can do this easily by creating a dictionary (map) of the letter index to the character. We can also create a reverse lookup for converting predictions back into characters to be used later.

```
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
```

Listing 27.3: Define the Alphabet Dataset.

Now we need to create our input and output pairs on which to train our neural network. We can do this by defining an input sequence length, then reading sequences from the input alphabet sequence. For example we use an input length of 1. Starting at the beginning of the raw input data, we can read off the first letter `A` and the next letter as the prediction `B`. We move along one character and repeat until we reach a prediction of `Z`.

```
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print(seq_in, '->', seq_out)
```

Listing 27.4: Create Patterns from Dataset.

We also print out the input pairs for sanity checking. Running the code to this point will produce the following output, summarizing input sequences of length 1 and a single output character.

```
A -> B
B -> C
C -> D
D -> E
E -> F
F -> G
G -> H
H -> I
I -> J
J -> K
K -> L
```

```
L -> M
M -> N
N -> O
O -> P
P -> Q
Q -> R
R -> S
S -> T
T -> U
U -> V
V -> W
W -> X
X -> Y
Y -> Z
```

Listing 27.5: Sample Alphabet Training Patterns.

We need to reshape the NumPy array into a format expected by the LSTM networks, that is `[samples, time steps, features]`.

```
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), seq_length, 1))
```

Listing 27.6: Reshape Training Patterns for LSTM Layer.

Once reshaped, we can then normalize the input integers to the range 0-to-1, the range of the sigmoid activation functions used by the LSTM network.

```
# normalize
X = X / float(len(alphabet))
```

Listing 27.7: Normalize Training Patterns.

Finally, we can think of this problem as a sequence classification task, where each of the 26 letters represents a different class. As such, we can convert the output (y) to a one hot encoding, using the Keras built-in function `to_categorical()`.

```
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
```

Listing 27.8: One Hot Encode Output Patterns.

We are now ready to fit different LSTM models.

27.2 LSTM for Learning One-Char to One-Char Mapping

Let's start off by designing a simple LSTM to learn how to predict the next character in the alphabet given the context of just one character. We will frame the problem as a random collection of one-letter input to one-letter output pairs. As we will see this is a difficult framing of the problem for the LSTM to learn. Let's define an LSTM network with 32 units and an output layer using the softmax activation function for making predictions. Because this is a multiclass classification problem, we can use the log loss function (called `categorical_crossentropy` in Keras), and optimize the network using the ADAM optimization function. The model is fit over 500 epochs with a batch size of 1.

```
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
```

Listing 27.9: Define and Fit LSTM Network Model.

After we fit the model we can evaluate and summarize the performance on the entire training dataset.

```
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 27.10: Evaluate the Fit LSTM Network Model.

We can then re-run the training data through the network and generate predictions, converting both the input and output pairs back into their original character format to get a visual idea of how well the network learned the problem.

```
# demonstrate some model predictions
for pattern in dataX:
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(len(alphabet))
  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]
  print(seq_in, "->", result)
```

Listing 27.11: Make Predictions Using the Fit LSTM Network.

The entire code listing is provided below for completeness.

```
# Naive LSTM to learn one-char to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
# fix random seed for reproducibility
numpy.random.seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print(seq_in, '->', seq_out)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(len(alphabet))
  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]
  print(seq_in, "->", result)
```

Listing 27.12: LSTM Network for one-char to one-char Mapping.

Running this example produces the following output.

```
Model Accuracy: 84.00%
['A'] -> B
['B'] -> C
['C'] -> D
['D'] -> E
['E'] -> F
['F'] -> G
['G'] -> H
['H'] -> I
['I'] -> J
['J'] -> K
['K'] -> L
['L'] -> M
['M'] -> N
['N'] -> O
['O'] -> P
['P'] -> Q
['Q'] -> R
['R'] -> S
['S'] -> T
['T'] -> U
['U'] -> W
['V'] -> Y
['W'] -> Z
['X'] -> Z
['Y'] -> Z
```

Listing 27.13: Output from the one-char to one-char Mapping.

We can see that this problem is indeed difficult for the network to learn. The reason is, the poor LSTM units do not have any context to work with. Each input-output pattern is shown to the network in a random order and the state of the network is reset after each pattern (each batch where each batch contains one pattern). This is abuse of the LSTM network architecture, treating it like a standard Multilayer Perceptron. Next, let's try a different framing of the problem in order to provide more sequence to the network from which to learn.

27.3 LSTM for a Feature Window to One-Char Mapping

A popular approach to adding more context to data for Multilayer Perceptrons is to use the window method. This is where previous steps in the sequence are provided as additional input features to the network. We can try the same trick to provide more context to the LSTM network. Here, we increase the sequence length from 1 to 3, for example:

```
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3
```

Listing 27.14: Increase Sequence Length.

Which creates training patterns like:

```
ABC -> D
BCD -> E
CDE -> F
```

Listing 27.15: Sample of Longer Input Sequence Length.

Each element in the sequence is then provided as a new input feature to the network. This requires a modification of how the input sequences reshaped in the data preparation step:

```
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), 1, seq_length))
```

Listing 27.16: Reshape Input so Sequence is Features.

It also requires a modification for how the sample patterns are reshaped when demonstrating predictions from the model.

```
x = numpy.reshape(pattern, (1, 1, len(pattern)))
```

Listing 27.17: Reshape Input for Predictions so Sequence is Features.

The entire code listing is provided below for completeness.

```
# Naive LSTM to learn three-char window to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
# fix random seed for reproducibility
numpy.random.seed(7)
```

```
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print(seq_in, '->', seq_out)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), 1, seq_length))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
  x = numpy.reshape(pattern, (1, 1, len(pattern)))
  x = x / float(len(alphabet))
  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]
  print(seq_in, "->", result)
```

Listing 27.18: LSTM Network for three-char features to one-char Mapping.

Running this example provides the following output.

```
Model Accuracy: 86.96%
['A', 'B', 'C'] -> D
['B', 'C', 'D'] -> E
['C', 'D', 'E'] -> F
['D', 'E', 'F'] -> G
['E', 'F', 'G'] -> H
['F', 'G', 'H'] -> I
['G', 'H', 'I'] -> J
['H', 'I', 'J'] -> K
['I', 'J', 'K'] -> L
['J', 'K', 'L'] -> M
['K', 'L', 'M'] -> N
['L', 'M', 'N'] -> O
```

```
['M', 'N', 'O'] -> P
['N', 'O', 'P'] -> Q
['O', 'P', 'Q'] -> R
['P', 'Q', 'R'] -> S
['Q', 'R', 'S'] -> T
['R', 'S', 'T'] -> U
['S', 'T', 'U'] -> V
['T', 'U', 'V'] -> Y
['U', 'V', 'W'] -> Z
['V', 'W', 'X'] -> Z
['W', 'X', 'Y'] -> Z
```

Listing 27.19: Output from the three-char Features to one-char Mapping.

We can see a small lift in performance that may or may not be real. This is a simple problem that we were still not able to learn with LSTMs even with the window method. Again, this is a misuse of the LSTM network by a poor framing of the problem. Indeed, the sequences of letters are time steps of one feature rather than one time step of separate features. We have given more context to the network, but not more sequence as it expected. In the next section, we will give more context to the network in the form of time steps.

27.4 LSTM for a Time Step Window to One-Char Mapping

In Keras, the intended use of LSTMs is to provide context in the form of time steps, rather than windowed features like with other network types. We can take our first example and simply change the sequence length from 1 to 3.

```
seq_length = 3
```

Listing 27.20: Increase Sequence Length.

Again, this creates input-output pairs that look like:

```
ABC -> D
BCD -> E
CDE -> F
DEF -> G
```

Listing 27.21: Sample of Longer Input Sequence Length.

The difference is that the reshaping of the input data takes the sequence as a time step sequence of one feature, rather than a single time step of multiple features.

```
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), seq_length, 1))
```

Listing 27.22: Reshape Input so Sequence is Time Steps.

This is the intended use of providing sequence context to your LSTM in Keras. The full code example is provided below for completeness.

```
# Naive LSTM to learn three-char time steps to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
# fix random seed for reproducibility
numpy.random.seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print(seq_in, '->', seq_out)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(len(alphabet))
  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]
  print(seq_in, "->", result)
```

Listing 27.23: LSTM Network for three-char Time Steps to one-char Mapping.

Running this example provides the following output.

```
Model Accuracy: 100.00%
['A', 'B', 'C'] -> D
['B', 'C', 'D'] -> E
['C', 'D', 'E'] -> F
['D', 'E', 'F'] -> G
['E', 'F', 'G'] -> H
['F', 'G', 'H'] -> I
['G', 'H', 'I'] -> J
```

```
['H', 'I', 'J'] -> K
['I', 'J', 'K'] -> L
['J', 'K', 'L'] -> M
['K', 'L', 'M'] -> N
['L', 'M', 'N'] -> O
['M', 'N', 'O'] -> P
['N', 'O', 'P'] -> Q
['O', 'P', 'Q'] -> R
['P', 'Q', 'R'] -> S
['Q', 'R', 'S'] -> T
['R', 'S', 'T'] -> U
['S', 'T', 'U'] -> V
['T', 'U', 'V'] -> W
['U', 'V', 'W'] -> X
['V', 'W', 'X'] -> Y
['W', 'X', 'Y'] -> Z
```

Listing 27.24: Output from the three-char Time Steps to one-char Mapping.

We can see that the model learns the problem perfectly as evidenced by the model evaluation and the example predictions. But it has learned a simpler problem. Specifically, it has learned to predict the next letter from a sequence of three letters in the alphabet. It can be shown any random sequence of three letters from the alphabet and predict the next letter.

It cannot actually enumerate the alphabet. I expect that a larger enough Multilayer Perceptron network might be able to learn the same mapping using the window method. The LSTM networks are stateful. They should be able to learn the whole alphabet sequence, but by default the Keras implementation resets the network state after each training batch.

27.5 LSTM State Maintained Between Samples Within A Batch

The Keras implementation of LSTMs resets the state of the network after each batch. This suggests that if we had a batch size large enough to hold all input patterns and if all the input patterns were ordered sequentially, that the LSTM could use the context of the sequence within the batch to better learn the sequence. We can demonstrate this easily by modifying the first example for learning a one-to-one mapping and increasing the batch size from 1 to the size of the training dataset. Additionally, Keras shuffles the training dataset before each training epoch. To ensure the training data patterns remain sequential, we can disable this shuffling.

```
model.fit(X, y, epochs=5000, batch_size=len(dataX), verbose=2, shuffle=False)
```

Listing 27.25: Increase Batch Size to Cover Entire Dataset.

The network will learn the mapping of characters using the within-batch sequence, but this context will not be available to the network when making predictions. We can evaluate both the ability of the network to make predictions randomly and in sequence. The full code example is provided below for completeness.

```
# Naive LSTM to learn one-char to one-char mapping with all data in each batch
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
from keras.preprocessing.sequence import pad_sequences
# fix random seed for reproducibility
numpy.random.seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print(seq_in, '->', seq_out)
# convert list of lists to array and pad sequences if needed
X = pad_sequences(dataX, maxlen=seq_length, dtype='float32')
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (X.shape[0], seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(16, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=5000, batch_size=len(dataX), verbose=2, shuffle=False)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(len(alphabet))
  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]
  print(seq_in, "->", result)
# demonstrate predicting random patterns
print("Test a Random Pattern:")
for i in range(0,20):
  pattern_index = numpy.random.randint(len(dataX))
  pattern = dataX[pattern_index]
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(len(alphabet))
  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]
print(seq_in, "->", result)
```

Listing 27.26: LSTM Network for one-char to one-char Mapping Within Batch.

Running the example provides the following output.

```
Model Accuracy: 100.00%
['A'] -> B
['B'] -> C
['C'] -> D
['D'] -> E
['E'] -> F
['F'] -> G
['G'] -> H
['H'] -> I
['I'] -> J
['J'] -> K
['K'] -> L
['L'] -> M
['M'] -> N
['N'] -> O
['O'] -> P
['P'] -> Q
['Q'] -> R
['R'] -> S
['S'] -> T
['T'] -> U
['U'] -> V
['V'] -> W
['W'] -> X
['X'] -> Y
['Y'] -> Z
Test a Random Pattern:
['T'] -> U
['V'] -> W
['M'] -> N
['Q'] -> R
['D'] -> E
['V'] -> W
['T'] -> U
['U'] -> V
['J'] -> K
['F'] -> G
['N'] -> O
['B'] -> C
['M'] -> N
['F'] -> G
['F'] -> G
['P'] -> Q
['A'] -> B
['K'] -> L
['W'] -> X
['E'] -> F
```

Listing 27.27: Output from the one-char to one-char Mapping Within Batch.

As we expected, the network is able to use the within-sequence context to learn the alphabet, achieving 100% accuracy on the training data. Importantly, the network can make accurate predictions for the next letter in the alphabet for randomly selected characters. Very impressive.

27.6 Stateful LSTM for a One-Char to One-Char Mapping

We have seen that we can break-up our raw data into fixed size sequences and that this representation can be learned by the LSTM, but only to learn random mappings of 3 characters to 1 character. We have also seen that we can pervert batch size to offer more sequence to the network, but only during training. Ideally, we want to expose the network to the entire sequence and let it learn the inter-dependencies, rather than us define those dependencies explicitly in the framing of the problem.

We can do this in Keras by making the LSTM layers stateful and manually resetting the state of the network at the end of the epoch, which is also the end of the training sequence. This is truly how the LSTM networks are intended to be used. We find that by allowing the network itself to learn the dependencies between the characters, that we need a smaller network (half the number of units) and fewer training epochs (almost half). We first need to define our LSTM layer as stateful. In so doing, we must explicitly specify the batch size as a dimension on the input shape. This also means that when we evaluate the network or make predictions, we must also specify and adhere to this same batch size. This is not a problem now as we are using a batch size of 1. This could introduce difficulties when making predictions when the batch size is not one as predictions will need to be made in batch and in sequence.

```
batch_size = 1
model.add(LSTM(16, batch_input_shape=(batch_size, X.shape[1], X.shape[2]), stateful=True))
```

Listing 27.28: Define a Stateful LSTM Layer.

An important difference in training the stateful LSTM is that we train it manually one epoch at a time and reset the state after each epoch. We can do this in a `for` loop. Again, we do not shuffle the input, preserving the sequence in which the input training data was created.

```
for i in range(300):
  model.fit(X, y, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
  model.reset_states()
```

Listing 27.29: Manually Manage LSTM State For Each Epoch.

As mentioned, we specify the batch size when evaluating the performance of the network on the entire training dataset.

```
# summarize performance of the model
scores = model.evaluate(X, y, batch_size=batch_size, verbose=0)
model.reset_states()
print("Model Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 27.30: Evaluate Model Using Pre-defined Batch Size.

Finally, we can demonstrate that the network has indeed learned the entire alphabet. We can seed it with the first letter `A`, request a prediction, feed the prediction back in as an input, and repeat the process all the way to `Z`.

```
# demonstrate some model predictions
seed = [char_to_int[alphabet[0]]]
for i in range(0, len(alphabet)-1):
  x = numpy.reshape(seed, (1, len(seed), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  print(int_to_char[seed[0]], "->", int_to_char[index])
  seed = [index]
model.reset_states()
```

Listing 27.31: Seed Network and Make Predictions from A to Z.

We can also see if the network can make predictions starting from an arbitrary letter.

```
# demonstrate a random starting point
letter = "K"
seed = [char_to_int[letter]]
print("New start: ", letter)
for i in range(0, 5):
  x = numpy.reshape(seed, (1, len(seed), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  print(int_to_char[seed[0]], "->", int_to_char[index])
  seed = [index]
model.reset_states()
```

Listing 27.32: Seed Network with a Random Letter and a Sequence of Predictions.

The entire code listing is provided below for completeness.

```
# Stateful LSTM to learn one-char to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
# fix random seed for reproducibility
numpy.random.seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print(seq_in, '->', seq_out)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), seq_length, 1))
```

```
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
batch_size = 1
model = Sequential()
model.add(LSTM(16, batch_input_shape=(batch_size, X.shape[1], X.shape[2]), stateful=True))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
for i in range(300):
  model.fit(X, y, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
  model.reset_states()
# summarize performance of the model
scores = model.evaluate(X, y, batch_size=batch_size, verbose=0)
model.reset_states()
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
seed = [char_to_int[alphabet[0]]]
for i in range(0, len(alphabet)-1):
  x = numpy.reshape(seed, (1, len(seed), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  print(int_to_char[seed[0]], "->", int_to_char[index])
  seed = [index]
model.reset_states()
# demonstrate a random starting point
letter = "K"
seed = [char_to_int[letter]]
print("New start: ", letter)
for i in range(0, 5):
  x = numpy.reshape(seed, (1, len(seed), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  print(int_to_char[seed[0]], "->", int_to_char[index])
  seed = [index]
model.reset_states()
```

Listing 27.33: Stateful LSTM Network for one-char to one-char Mapping.

Running the example provides the following output.

```
Model Accuracy: 100.00%
A -> B
B -> C
C -> D
D -> E
E -> F
F -> G
G -> H
H -> I
I -> J
J -> K
K -> L
L -> M
```

```
M -> N
N -> O
O -> P
P -> Q
Q -> R
R -> S
S -> T
T -> U
U -> V
V -> W
W -> X
X -> Y
Y -> Z
New start: K
K -> B
B -> C
C -> D
D -> E
E -> F
```

Listing 27.34: Output from the Stateful LSTM for one-char to one-char Mapping.

We can see that the network has memorized the entire alphabet perfectly. It used the context of the samples themselves and learned whatever dependency it needed to predict the next character in the sequence. We can also see that if we seed the network with the first letter, that it can correctly rattle off the rest of the alphabet. We can also see that it has only learned the full alphabet sequence and that from a cold start. When asked to predict the next letter from `K` that it predicts `B` and falls back into regurgitating the entire alphabet. To truly predict `K`, the state of the network would need to be warmed up iteratively fed the letters from `A` to `J`. This tells us that we could achieve the same effect with a *stateless* LSTM by preparing training data like:

```
---a -> b
--ab -> c
-abc -> d
abcd -> e
```

Listing 27.35: Sample of Equivalent Training Data for Non-Stateful LSTM Layers.

Where the input sequence is fixed at 25 (`a`-to-`y` to predict `z`) and patterns are prefixed with zero-padding. Finally, this raises the question of training an LSTM network using variable length input sequences to predict the next character.

27.7 LSTM with Variable Length Input to One-Char Output

In the previous section we discovered that the Keras *stateful* LSTM was really only a short cut to replaying the first `n`-sequences, but didn't really help us learn a generic model of the alphabet. In this section we explore a variation of the *stateless* LSTM that learns random subsequences of the alphabet and an effort to build a model that can be given arbitrary letters or subsequences of letters and predict the next letter in the alphabet.

Firstly, we are changing the framing of the problem. To simplify we will define a maximum input sequence length and set it to a small value like 5 to speed up training. This defines the maximum length of subsequences of the alphabet which will be drawn for training. In extensions, this could just be set to the full alphabet (26) or longer if we allow looping back to the start of the sequence. We also need to define the number of random sequences to create, in this case, 1,000. This too could be more or less. I expect fewer patterns are actually required.

```
# prepare the dataset of input to output pairs encoded as integers
num_inputs = 1000
max_len = 5
dataX = []
dataY = []
for i in range(num_inputs):
  start = numpy.random.randint(len(alphabet)-2)
  end = numpy.random.randint(start, min(start+max_len,len(alphabet)-1))
  sequence_in = alphabet[start:end+1]
  sequence_out = alphabet[end + 1]
  dataX.append([char_to_int[char] for char in sequence_in])
  dataY.append(char_to_int[sequence_out])
  print(sequence_in, '->', sequence_out)
```

Listing 27.36: Create Dataset of Variable Length Input Sequences.

Running this code in the broader context will create input patterns that look like the following:

```
PQRST -> U
W -> X
O -> P
OPQ -> R
IJKLM -> N
QRSTU -> V
ABCD -> E
X -> Y
GHIJ -> K
```

Listing 27.37: Sample of Variable Length Input Sequences.

The input sequences vary in length between 1 and `max_len` and therefore require zero padding. Here, we use left-hand-side (prefix) padding with the Keras built in `pad_sequences()` function.

```
X = pad_sequences(dataX, maxlen=max_len, dtype='float32')
```

Listing 27.38: Left-Pad Variable Length Input Sequences.

The trained model is evaluated on randomly selected input patterns. This could just as easily be new randomly generated sequences of characters. I also believe this could also be a linear sequence seeded with `A` with outputs fed back in as single character inputs. The full code listing is provided below for completeness.

```
# LSTM with Variable Length Input Sequences to One Character Output
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
from keras.preprocessing.sequence import pad_sequences
```

```
# fix random seed for reproducibility
numpy.random.seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
num_inputs = 1000
max_len = 5
dataX = []
dataY = []
for i in range(num_inputs):
  start = numpy.random.randint(len(alphabet)-2)
  end = numpy.random.randint(start, min(start+max_len,len(alphabet)-1))
  sequence_in = alphabet[start:end+1]
  sequence_out = alphabet[end + 1]
  dataX.append([char_to_int[char] for char in sequence_in])
  dataY.append(char_to_int[sequence_out])
  print(sequence_in, '->', sequence_out)
# convert list of lists to array and pad sequences if needed
X = pad_sequences(dataX, maxlen=max_len, dtype='float32')
# reshape X to be [samples, time steps, features]
X = numpy.reshape(X, (X.shape[0], max_len, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
batch_size = 1
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], 1)))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=500, batch_size=batch_size, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for i in range(20):
  pattern_index = numpy.random.randint(len(dataX))
  pattern = dataX[pattern_index]
  x = pad_sequences([pattern], maxlen=max_len, dtype='float32')
  x = numpy.reshape(x, (1, max_len, 1))
  x = x / float(len(alphabet))
  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]
  print(seq_in, "->", result)
```

Listing 27.39: LSTM Network for Variable Length Sequences to one-char Mapping.

Running this code produces the following output:

```
Model Accuracy: 98.90%
['Q', 'R'] -> S
```

```
['W', 'X'] -> Y
['W', 'X'] -> Y
['C', 'D'] -> E
['E'] -> F
['S', 'T', 'U'] -> V
['G', 'H', 'I', 'J', 'K'] -> L
['O', 'P', 'Q', 'R', 'S'] -> T
['C', 'D'] -> E
['O'] -> P
['N', 'O', 'P'] -> Q
['D', 'E', 'F', 'G', 'H'] -> I
['X'] -> Y
['K'] -> L
['M'] -> N
['R'] -> T
['K'] -> L
['E', 'F', 'G'] -> H
['Q'] -> R
['Q', 'R', 'S'] -> T
```

Listing 27.40: Output for the LSTM Network for Variable Length Sequences to one-char Mapping.

We can see that although the model did not learn the alphabet perfectly from the randomly generated subsequences, it did very well. The model was not tuned and may require more training or a larger network, or both (an exercise for the reader). This is a good natural extension to the *all sequential input examples in each batch* alphabet model learned above in that it can handle ad hoc queries, but this time of arbitrary sequence length (up to the max length).

27.8 Summary

In this lesson you discovered LSTM recurrent neural networks in Keras and how they manage state. Specifically, you learned:

- How to develop a naive LSTM network for one-character to one-character prediction.
- How to configure a naive LSTM to learn a sequence across time steps within a sample.
- How to configure an LSTM to learn a sequence across samples by manually managing state.

27.8.1 Next

In this lesson you developed your understanding for how LSTM networks maintain state for simple sequence prediction problems. Up next you will use your understanding of LSTM networks to develop larger text generation models.