Chapter 20

Project: Develop a Neural Language Model for Text Generation

A language model can predict the probability of the next word in the sequence, based on the words already observed in the sequence. Neural network models are a preferred method for developing statistical language models because they can use a distributed representation where different words with similar meanings have similar representation and because they can use a large context of recently observed words when making predictions. In this tutorial, you will discover how to develop a statistical language model using deep learning in Python. After completing this tutorial, you will know:

- How to prepare text for developing a word-based language model.
- How to design and fit a neural language model with a learned embedding and an LSTM hidden layer.
- How to use the learned language model to generate new text with similar statistical properties as the source text.

Let's get started.

20.1 Tutorial Overview

This tutorial is divided into the following parts:

1. The Republic by Plato
2. Data Preparation
3. Train Language Model
4. Use Language Model

20.2 The Republic by Plato

The Republic is the classical Greek philosopher Plato's most famous work. It is structured as a dialog (e.g. conversation) on the topic of order and justice within a city state The entire text is available for free in the public domain. It is available on the Project Gutenberg website in a number of formats. You can download the ASCII text version of the entire book (or books) here (you might need to open the URL twice):

- Download The Republic by Plato.
  http://www.gutenberg.org/cache/epub/1497/pg1497.txt

Download the book text and place it in your current working directly with the filename `republic.txt`. Open the file in a text editor and delete the front and back matter. This includes details about the book at the beginning, a long analysis, and license information at the end. The text should begin with:

> BOOK I.
>
> I went down yesterday to the Piraeus with Glaucon the son of Ariston, ...

And end with:

> ... And it shall be well with us both in this life and in the pilgrimage of a thousand years which we have been describing.

Save the cleaned version as `republic_clean.txt` in your current working directory. The file should be about 15,802 lines of text. Now we can develop a language model from this text.

20.3 Data Preparation

We will start by preparing the data for modeling. The first step is to look at the data.

20.3.1 Review the Text

Open the text in an editor and just look at the text data. For example, here is the first piece of dialog:

> BOOK I.
>
> I went down yesterday to the Piraeus with Glaucon the son of Ariston, that I might offer up my prayers to the goddess (Bendis, the Thracian Artemis.); and also because I wanted to see in what manner they would celebrate the festival, which was a new thing. I was delighted with the procession of the inhabitants; but that of the Thracians was equally, if not more, beautiful. When we had finished our prayers and viewed the spectacle, we turned in the direction of the city; and at that instant Polemarchus the son of Cephalus chanced to catch sight of us from a distance as we were starting on our way home, and told his servant to run and bid us wait for him. The servant took hold of me by the cloak behind, and said: Polemarchus desires you to wait.

> I turned round, and asked him where his master was.
>
> There he is, said the youth, coming after you, if you will only wait.
>
> Certainly we will, said Glaucon; and in a few minutes Polemarchus appeared, and with him Adeimantus, Glaucon's brother, Niceratus the son of Nicias, and several others who had been at the procession.
>
> Polemarchus said to me: I perceive, Socrates, that you and your companion are already on your way to the city.
>
> You are not far wrong, I said.
>
> ...

What do you see that we will need to handle in preparing the data? Here's what I see from a quick look:

- Book/Chapter headings (e.g. *BOOK I.*).
- Lots of punctuation (e.g. -, *;*-, *?*-, and more).
- Strange names (e.g. *Polemarchus*).
- Some long monologues that go on for hundreds of lines.
- Some quoted dialog (e.g. '...').

These observations, and more, suggest at ways that we may wish to prepare the text data. The specific way we prepare the data really depends on how we intend to model it, which in turn depends on how we intend to use it.

20.3.2 Language Model Design

In this tutorial, we will develop a model of the text that we can then use to generate new sequences of text. The language model will be statistical and will predict the probability of each word given an input sequence of text. The predicted word will be fed in as input to in turn generate the next word. A key design decision is how long the input sequences should be. They need to be long enough to allow the model to learn the context for the words to predict. This input length will also define the length of seed text used to generate new sequences when we use the model.

There is no correct answer. With enough time and resources, we could explore the ability of the model to learn with differently sized input sequences. Instead, we will pick a length of 50 words for the length of the input sequences, somewhat arbitrarily. We could process the data so that the model only ever deals with self-contained sentences and pad or truncate the text to meet this requirement for each input sequence. You could explore this as an extension to this tutorial.

Instead, to keep the example brief, we will let all of the text flow together and train the model to predict the next word across sentences, paragraphs, and even books or chapters in the text. Now that we have a model design, we can look at transforming the raw text into sequences of 100 input words to 1 output word, ready to fit a model.

20.3.3 Load Text

The first step is to load the text into memory. We can develop a small function to load the entire text file into memory and return it. The function is called `load_doc()` and is listed below. Given a filename, it returns a sequence of loaded text.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text
```

Listing 20.1: Function to load text into memory.

Using this function, we can load the cleaner version of the document in the file `republic_clean.txt` as follows:

```
# load document
in_filename = 'republic_clean.txt'
doc = load_doc(in_filename)
print(doc[:200])
```

Listing 20.2: Example of loading the text into memory.

Running this snippet loads the document and prints the first 200 characters as a sanity check.

```
BOOK I.

I went down yesterday to the Piraeus with Glaucon the son of Ariston,
that I might offer up my prayers to the goddess (Bendis, the Thracian
Artemis.); and also because I wanted to see in what
```

Listing 20.3: Example output of loading the text into memory.

So far, so good. Next, let's clean the text.

20.3.4 Clean Text

We need to transform the raw text into a sequence of tokens or words that we can use as a source to train the model. Based on reviewing the raw text (above), below are some specific operations we will perform to clean the text. You may want to explore more cleaning operations yourself as an extension.

- Replace '-' with a white space so we can split words better.
- Split words based on white space.
- Remove all punctuation from words to reduce the vocabulary size (e.g. 'What?' becomes 'What').
- Remove all words that are not alphabetic to remove standalone punctuation tokens.

- Normalize all words to lowercase to reduce the vocabulary size.

Vocabulary size is a big deal with language modeling. A smaller vocabulary results in a smaller model that trains faster. We can implement each of these cleaning operations in this order in a function. Below is the function `clean_doc()` that takes a loaded document as an argument and returns an array of clean tokens.

```
# turn a doc into clean tokens
def clean_doc(doc):
  # replace '--' with a space ' '
  doc = doc.replace('--', ' ')
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # make lower case
  tokens = [word.lower() for word in tokens]
  return tokens
```

Listing 20.4: Function to clean text.

We can run this cleaning operation on our loaded document and print out some of the tokens and statistics as a sanity check.

```
# clean document
tokens = clean_doc(doc)
print(tokens[:200])
print('Total Tokens: %d' % len(tokens))
print('Unique Tokens: %d' % len(set(tokens)))
```

Listing 20.5: Example of cleaning text.

First, we can see a nice list of tokens that look cleaner than the raw text. We could remove the 'Book I' chapter markers and more, but this is a good start.

```
['book', 'i', 'i', 'went', 'down', 'yesterday', 'to', 'the', 'piraeus', 'with', 'glaucon',
    'the', 'son', 'of', 'ariston', 'that', 'i', 'might', 'offer', 'up', 'my', 'prayers',
    'to', 'the', 'goddess', 'bendis', 'the', 'thracian', 'artemis', 'and', 'also',
    'because', 'i', 'wanted', 'to', 'see', 'in', 'what', 'manner', 'they', 'would',
    'celebrate', 'the', 'festival', 'which', 'was', 'a', 'new', 'thing', 'i', 'was',
    'delighted', 'with', 'the', 'procession', 'of', 'the', 'inhabitants', 'but', 'that',
    'of', 'the', 'thracians', 'was', 'equally', 'if', 'not', 'more', 'beautiful', 'when',
    'we', 'had', 'finished', 'our', 'prayers', 'and', 'viewed', 'the', 'spectacle', 'we',
    'turned', 'in', 'the', 'direction', 'of', 'the', 'city', 'and', 'at', 'that',
    'instant', 'polemarchus', 'the', 'son', 'of', 'cephalus', 'chanced', 'to', 'catch',
    'sight', 'of', 'us', 'from', 'a', 'distance', 'as', 'we', 'were', 'starting', 'on',
    'our', 'way', 'home', 'and', 'told', 'his', 'servant', 'to', 'run', 'and', 'bid', 'us',
    'wait', 'for', 'him', 'the', 'servant', 'took', 'hold', 'of', 'me', 'by', 'the',
    'cloak', 'behind', 'and', 'said', 'polemarchus', 'desires', 'you', 'to', 'wait', 'i',
    'turned', 'round', 'and', 'asked', 'him', 'where', 'his', 'master', 'was', 'there',
    'he', 'is', 'said', 'the', 'youth', 'coming', 'after', 'you', 'if', 'you', 'will',
    'only', 'wait', 'certainly', 'we', 'will', 'said', 'glaucon', 'and', 'in', 'a', 'few',
    'minutes', 'polemarchus', 'appeared', 'and', 'with', 'him', 'adeimantus', 'glaucons',
```

```
    'brother', 'niceratus', 'the', 'son', 'of', 'nicias', 'and', 'several', 'others',
    'who', 'had', 'been', 'at', 'the', 'procession', 'polemarchus', 'said']
```

Listing 20.6: Example output of tokenized and clean text.

We also get some statistics about the clean document. We can see that there are just under 120,000 words in the clean text and a vocabulary of just under 7,500 words. This is smallish and models fit on this data should be manageable on modest hardware.

```
Total Tokens: 118684
Unique Tokens: 7409
```

Listing 20.7: Example output summarizing properties of the clean text.

Next, we can look at shaping the tokens into sequences and saving them to file.

20.3.5 Save Clean Text

We can organize the long list of tokens into sequences of 50 input words and 1 output word. That is, sequences of 51 words. We can do this by iterating over the list of tokens from token 51 onwards and taking the prior 50 tokens as a sequence, then repeating this process to the end of the list of tokens. We will transform the tokens into space-separated strings for later storage in a file. The code to split the list of clean tokens into sequences with a length of 51 tokens is listed below.

```
# organize into sequences of tokens
length = 50 + 1
sequences = list()
for i in range(length, len(tokens)):
  # select sequence of tokens
  seq = tokens[i-length:i]
  # convert into a line
  line = ' '.join(seq)
  # store
  sequences.append(line)
print('Total Sequences: %d' % len(sequences))
```

Listing 20.8: Split document into sequences of text.

Running this piece creates a long list of lines. Printing statistics on the list, we can see that we will have exactly 118,633 training patterns to fit our model.

```
Total Sequences: 118633
```

Listing 20.9: Example output of splitting the document into sequences.

Next, we can save the sequences to a new file for later loading. We can define a new function for saving lines of text to a file. This new function is called `save_doc()` and is listed below. It takes as input a list of lines and a filename. The lines are written, one per line, in ASCII format.

```
# save tokens to file, one dialog per line
def save_doc(lines, filename):
  data = '\n'.join(lines)
  file = open(filename, 'w')
  file.write(data)
  file.close()
```

Listing 20.10: Function to save sequences of text to file.

We can call this function and save our training sequences to the file `republic_sequences.txt`.

```
# save sequences to file
out_filename = 'republic_sequences.txt'
save_doc(sequences, out_filename)
```

Listing 20.11: Example of saving sequences to file.

Take a look at the file with your text editor. You will see that each line is shifted along one word, with a new word at the end to be predicted; for example, here are the first 3 lines in truncated form:

```
book i i ... catch sight of
i i went ... sight of us
i went down ... of us from
...
```

Listing 20.12: Example contents of sequences saved to file.

20.3.6 Complete Example

Tying all of this together, the complete code listing is provided below.

```
import string
import re

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # replace '--' with a space ' '
  doc = doc.replace('--', ' ')
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # make lower case
  tokens = [word.lower() for word in tokens]
  return tokens

# save tokens to file, one dialog per line
```

```
def save_doc(lines, filename):
  data = '\n'.join(lines)
  file = open(filename, 'w')
  file.write(data)
  file.close()

# load document
in_filename = 'republic_clean.txt'
doc = load_doc(in_filename)
print(doc[:200])
# clean document
tokens = clean_doc(doc)
print(tokens[:200])
print('Total Tokens: %d' % len(tokens))
print('Unique Tokens: %d' % len(set(tokens)))
# organize into sequences of tokens
length = 50 + 1
sequences = list()
for i in range(length, len(tokens)):
  # select sequence of tokens
  seq = tokens[i-length:i]
  # convert into a line
  line = ' '.join(seq)
  # store
  sequences.append(line)
print('Total Sequences: %d' % len(sequences))
# save sequences to file
out_filename = 'republic_sequences.txt'
save_doc(sequences, out_filename)
```

Listing 20.13: Complete example preparing text data for modeling.

You should now have training data stored in the file `republic_sequences.txt` in your current working directory. Next, let's look at how to fit a language model to this data.

20.4 Train Language Model

We can now train a statistical language model from the prepared data. The model we will train is a neural language model. It has a few unique characteristics:

- It uses a distributed representation for words so that different words with similar meanings will have a similar representation.
- It learns the representation at the same time as learning the model.
- It learns to predict the probability for the next word using the context of the last 100 words.

Specifically, we will use an Embedding Layer to learn the representation of words, and a Long Short-Term Memory (LSTM) recurrent neural network to learn to predict words based on their context. Let's start by loading our training data.

20.4.1 Load Sequences

We can load our training data using the `load_doc()` function we developed in the previous section. Once loaded, we can split the data into separate training sequences by splitting based on new lines. The snippet below will load the `republic_sequences.txt` data file from the current working directory.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# load
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
```

Listing 20.14: Load the clean sequences from file.

Next, we can encode the training data.

20.4.2 Encode Sequences

The word embedding layer expects input sequences to be comprised of integers. We can map each word in our vocabulary to a unique integer and encode our input sequences. Later, when we make predictions, we can convert the prediction to numbers and look up their associated words in the same mapping. To do this encoding, we will use the `Tokenizer` class in the Keras API.

First, the `Tokenizer` must be trained on the entire training dataset, which means it finds all of the unique words in the data and assigns each a unique integer. We can then use the fit `Tokenizer` to encode all of the training sequences, converting each sequence from a list of words to a list of integers.

```
# integer encode sequences of words
tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
sequences = tokenizer.texts_to_sequences(lines)
```

Listing 20.15: Train a tokenizer on the loaded sequences.

We can access the mapping of words to integers as a dictionary attribute called `word_index` on the `Tokenizer` object. We need to know the size of the vocabulary for defining the embedding layer later. We can determine the vocabulary by calculating the size of the mapping dictionary.

Words are assigned values from 1 to the total number of words (e.g. 7,409). The `Embedding` layer needs to allocate a vector representation for each word in this vocabulary from index 1 to the largest index and because indexing of arrays is zero-offset, the index of the word at the end of the vocabulary will be 7,409; that means the array must be 7,409 + 1 in length. Therefore, when specifying the vocabulary size to the `Embedding` layer, we specify it as 1 larger than the actual vocabulary.

```
# vocabulary size
vocab_size = len(tokenizer.word_index) + 1
```

Listing 20.16: Calculate the size of the vocabulary.

20.4.3 Sequence Inputs and Output

Now that we have encoded the input sequences, we need to separate them into input (`X`) and output (`y`) elements. We can do this with array slicing. After separating, we need to one hot encode the output word. This means converting it from an integer to a vector of 0 values, one for each word in the vocabulary, with a 1 to indicate the specific word at the index of the words integer value.

This is so that the model learns to predict the probability distribution for the next word and the ground truth from which to learn from is 0 for all words except the actual word that comes next. Keras provides the `to_categorical()` that can be used to one hot encode the output words for each input-output sequence pair.

Finally, we need to specify to the `Embedding` layer how long input sequences are. We know that there are 50 words because we designed the model, but a good generic way to specify that is to use the second dimension (number of columns) of the input data's shape. That way, if you change the length of sequences when preparing data, you do not need to change this data loading code; it is generic.

```
# separate into input and output
sequences = array(sequences)
X, y = sequences[:,:-1], sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
seq_length = X.shape[1]
```

Listing 20.17: Split text data into input and output sequences.

20.4.4 Fit Model

We can now define and fit our language model on the training data. The learned embedding needs to know the size of the vocabulary and the length of input sequences as previously discussed. It also has a parameter to specify how many dimensions will be used to represent each word. That is, the size of the embedding vector space.

Common values are 50, 100, and 300. We will use 50 here, but consider testing smaller or larger values. We will use a two LSTM hidden layers with 100 memory cells each. More memory cells and a deeper network may achieve better results.

A dense fully connected layer with 100 neurons connects to the LSTM hidden layers to interpret the features extracted from the sequence. The output layer predicts the next word as a single vector the size of the vocabulary with a probability for each word in the vocabulary. A softmax activation function is used to ensure the outputs have the characteristics of normalized probabilities.

```
# define the model
def define_model(vocab_size, seq_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 50, input_length=seq_length))
```

```
model.add(LSTM(100, return_sequences=True))
model.add(LSTM(100))
model.add(Dense(100, activation='relu'))
model.add(Dense(vocab_size, activation='softmax'))
# compile network
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# summarize defined model
model.summary()
plot_model(model, to_file='model.png', show_shapes=True)
return model
```

Listing 20.18: Define the language model.

A summary of the defined network is printed as a sanity check to ensure we have constructed what we intended.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 50, 50)            370500
_________________________________________________________________
lstm_1 (LSTM)                (None, 50, 100)           60400
_________________________________________________________________
lstm_2 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense_1 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_2 (Dense)              (None, 7410)              748410
=================================================================
Total params: 1,269,810
Trainable params: 1,269,810
Non-trainable params: 0
_________________________________________________________________
```

Listing 20.19: Example output from summarizing the defined model.

A plot the defined model is then saved to file with the name `model.png`.

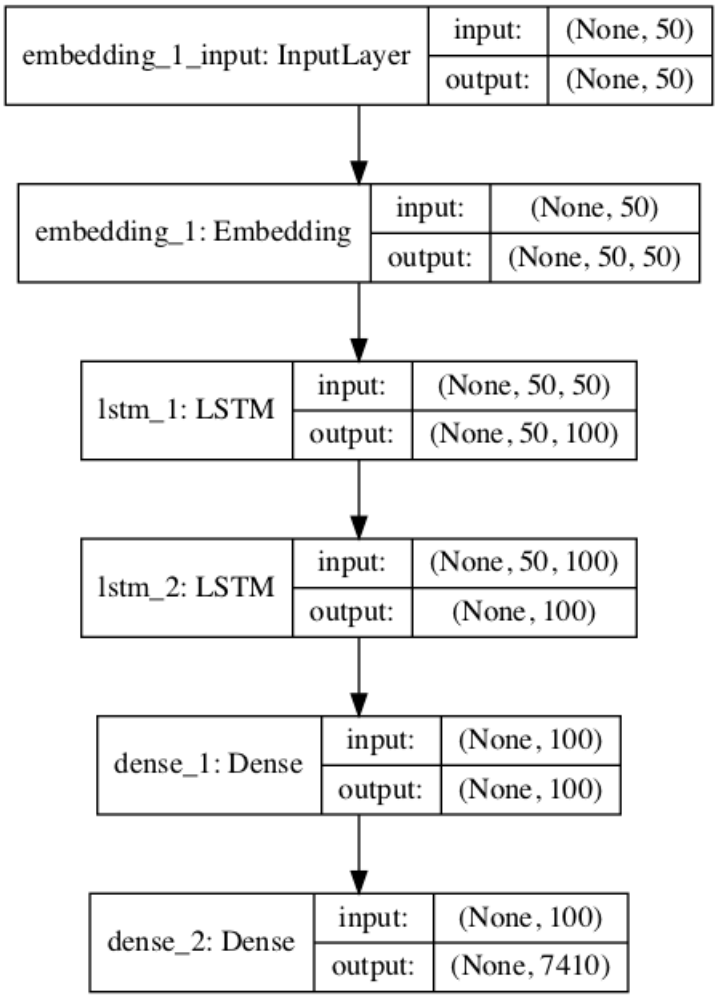

Figure 20.1: Plot of the defined word-based language model.

The model is compiled specifying the categorical cross entropy loss needed to fit the model. Technically, the model is learning a multiclass classification and this is the suitable loss function for this type of problem. The efficient Adam implementation to mini-batch gradient descent is used and accuracy is evaluated of the model. Finally, the model is fit on the data for 100 training epochs with a modest batch size of 128 to speed things up. Training may take a few hours on modern hardware without GPUs. You can speed it up with a larger batch size and/or fewer training epochs.

During training, you will see a summary of performance, including the loss and accuracy evaluated from the training data at the end of each batch update. You will get different results, but perhaps an accuracy of just over 50% of predicting the next word in the sequence, which is not bad. We are not aiming for 100% accuracy (e.g. a model that memorized the text), but rather a model that captures the essence of the text.

```
...
Epoch 96/100
118633/118633 [==============================] - 265s - loss: 2.0324 - acc: 0.5187
Epoch 97/100
118633/118633 [==============================] - 265s - loss: 2.0136 - acc: 0.5247
Epoch 98/100
```

```
118633/118633 [==============================] - 267s - loss: 1.9956 - acc: 0.5262
Epoch 99/100
118633/118633 [==============================] - 266s - loss: 1.9812 - acc: 0.5291
Epoch 100/100
118633/118633 [==============================] - 270s - loss: 1.9709 - acc: 0.5315
```

Listing 20.20: Example output from training the language model.

20.4.5 Save Model

At the end of the run, the trained model is saved to file. Here, we use the Keras model API to save the model to the file `model.h5` in the current working directory. Later, when we load the model to make predictions, we will also need the mapping of words to integers. This is in the `Tokenizer` object, and we can save that too using Pickle.

```
# save the model to file
model.save('model.h5')
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

Listing 20.21: Save the fit model and `Tokenizer` to file.

20.4.6 Complete Example

We can put all of this together; the complete example for fitting the language model is listed below.

```
from numpy import array
from pickle import dump
from keras.preprocessing.text import Tokenizer
from keras.utils.vis_utils import plot_model
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# define the model
def define_model(vocab_size, seq_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 50, input_length=seq_length))
  model.add(LSTM(100, return_sequences=True))
  model.add(LSTM(100))
  model.add(Dense(100, activation='relu'))
```

```
  model.add(Dense(vocab_size, activation='softmax'))
  # compile network
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model

# load
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
# integer encode sequences of words
tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
sequences = tokenizer.texts_to_sequences(lines)
# vocabulary size
vocab_size = len(tokenizer.word_index) + 1
# separate into input and output
sequences = array(sequences)
X, y = sequences[:,:-1], sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
seq_length = X.shape[1]
# define model
model = define_model(vocab_size, seq_length)
# fit model
model.fit(X, y, batch_size=128, epochs=100)
# save the model to file
model.save('model.h5')
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

Listing 20.22: Complete example training the language model.

20.5 Use Language Model

Now that we have a trained language model, we can use it. In this case, we can use it to generate new sequences of text that have the same statistical properties as the source text. This is not practical, at least not for this example, but it gives a concrete example of what the language model has learned. We will start by loading the training sequences again.

20.5.1 Load Data

We can use the same code from the previous section to load the training data sequences of text. Specifically, the `load_doc()` function.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
```

```
  file.close()
  return text

# load cleaned text sequences
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
```

Listing 20.23: Load the clean sequences from file.

We need the text so that we can choose a source sequence as input to the model for generating a new sequence of text. The model will require 50 words as input. Later, we will need to specify the expected length of input. We can determine this from the input sequences by calculating the length of one line of the loaded data and subtracting 1 for the expected output word that is also on the same line.

```
seq_length = len(lines[0].split()) - 1
```

Listing 20.24: Calculate the expected input length.

20.5.2 Load Model

We can now load the model from file. Keras provides the `load_model()` function for loading the model, ready for use.

```
# load the model
model = load_model('model.h5')
```

Listing 20.25: Load the saved model from file.

We can also load the tokenizer from file using the Pickle API.

```
# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
```

Listing 20.26: Load the saved `Tokenizer` from file.

We are ready to use the loaded model.

20.5.3 Generate Text

The first step in generating text is preparing a seed input. We will select a random line of text from the input text for this purpose. Once selected, we will print it so that we have some idea of what was used.

```
# select a seed text
seed_text = lines[randint(0,len(lines))]
print(seed_text + '\n')
```

Listing 20.27: Select random examples as seed text.

Next, we can generate new words, one at a time. First, the seed text must be encoded to integers using the same tokenizer that we used when training the model.

```
encoded = tokenizer.texts_to_sequences([seed_text])[0]
```

Listing 20.28: Encode the selected seed text.

The model can predict the next word directly by calling `model.predict_classes()` that will return the index of the word with the highest probability.

```
# predict probabilities for each word
yhat = model.predict_classes(encoded, verbose=0)
```

Listing 20.29: Predict the next word in the sequence.

We can then look up the index in the `Tokenizer`'s mapping to get the associated word.

```
out_word = ''
for word, index in tokenizer.word_index.items():
  if index == yhat:
    out_word = word
    break
```

Listing 20.30: Map the predicted integer to a word in the known vocabulary.

We can then append this word to the seed text and repeat the process. Importantly, the input sequence is going to get too long. We can truncate it to the desired length after the input sequence has been encoded to integers. Keras provides the `pad_sequences()` function that we can use to perform this truncation.

```
encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
```

Listing 20.31: Pad the encoded sequence.

We can wrap all of this into a function called `generate_seq()` that takes as input the model, the tokenizer, input sequence length, the seed text, and the number of words to generate. It then returns a sequence of words generated by the model.

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
  result = list()
  in_text = seed_text
  # generate a fixed number of words
  for _ in range(n_words):
    # encode the text as integer
    encoded = tokenizer.texts_to_sequences([in_text])[0]
    # truncate sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
    # predict probabilities for each word
    yhat = model.predict_classes(encoded, verbose=0)
    # map predicted word index to word
    out_word = ''
    for word, index in tokenizer.word_index.items():
      if index == yhat:
        out_word = word
        break
    # append to input
    in_text += ' ' + out_word
    result.append(out_word)
  return ' '.join(result)
```

Listing 20.32: Function to generate a sequence of words given the model and seed text.

We are now ready to generate a sequence of new words given some seed text.

```
# generate new text
generated = generate_seq(model, tokenizer, seq_length, seed_text, 50)
print(generated)
```

Listing 20.33: Example of generating a sequence of text.

Putting this all together, the complete code listing for generating text from the learned-language model is listed below.

```
from random import randint
from pickle import load
from keras.models import load_model
from keras.preprocessing.sequence import pad_sequences

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# generate a sequence from a language model
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
  result = list()
  in_text = seed_text
  # generate a fixed number of words
  for _ in range(n_words):
    # encode the text as integer
    encoded = tokenizer.texts_to_sequences([in_text])[0]
    # truncate sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
    # predict probabilities for each word
    yhat = model.predict_classes(encoded, verbose=0)
    # map predicted word index to word
    out_word = ''
    for word, index in tokenizer.word_index.items():
      if index == yhat:
        out_word = word
        break
    # append to input
    in_text += ' ' + out_word
    result.append(out_word)
  return ' '.join(result)

# load cleaned text sequences
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
seq_length = len(lines[0].split()) - 1
# load the model
model = load_model('model.h5')
# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
```

```
# select a seed text
seed_text = lines[randint(0,len(lines))]
print(seed_text + '\n')
# generate new text
generated = generate_seq(model, tokenizer, seq_length, seed_text, 50)
print(generated)
```

Listing 20.34: Complete example of generating sequences of text.

Running the example first prints the seed text.

```
when he said that a man when he grows old may learn many things for he can no more learn
    much than he can run much youth is the time for any extraordinary toil of course and
    therefore calculation and geometry and all the other elements of instruction which are a
```

Listing 20.35: Example output from selecting seed text.

Then 50 words of generated text are printed.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
preparation for dialectic should be presented to the name of idle spendthrifts of whom the
    other is the manifold and the unjust and is the best and the other which delighted to
    be the opening of the soul of the soul and the embroiderer will have to be said at
```

Listing 20.36: Example output of generated text.

You can see that the text seems reasonable. In fact, the addition of concatenation would help in interpreting the seed and the generated text. Nevertheless, the generated text gets the right kind of words in the right kind of order. Try running the example a few times to see other examples of generated text.

20.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Contrived Seed Text**. Hand craft or select seed text and evaluate how the seed text impacts the generated text, specifically the initial words or sentences generated.
- **Simplify Vocabulary**. Explore a simpler vocabulary, perhaps with stemmed words or stop words removed.
- **Data Cleaning**. Consider using more or less cleaning of the text, perhaps leave in some punctuation or perhaps replacing all fancy names with one or a handful. Evaluate how these changes to the size of the vocabulary impact the generated text.
- **Tune Model**. Tune the model, such as the size of the embedding or number of memory cells in the hidden layer, to see if you can develop a better model.
- **Deeper Model**. Extend the model to have multiple LSTM hidden layers, perhaps with dropout to see if you can develop a better model.

- **Develop Pre-Trained Embedding**. Extend the model to use pre-trained Word2Vec vectors to see if it results in a better model.
- **Use GloVe Embedding**. Use the GloVe word embedding vectors with and without fine tuning by the network and evaluate how it impacts training and the generated words.
- **Sequence Length**. Explore training the model with different length input sequences, both shorter and longer, and evaluate how it impacts the quality of the generated text.
- **Reduce Scope**. Consider training the model on one book (chapter) or a subset of the original text and evaluate the impact on training, training speed and the resulting generated text.
- **Sentence-Wise Model**. Split the raw data based on sentences and pad each sentence to a fixed length (e.g. the longest sentence length).

If you explore any of these extensions, I'd love to know.

20.7 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Project Gutenberg.
  https://www.gutenberg.org/
- The Republic by Plato on Project Gutenberg.
  https://www.gutenberg.org/ebooks/1497
- Republic (Plato) on Wikipedia.
  https://en.wikipedia.org/wiki/Republic_(Plato)
- Language model on Wikipedia.
  https://en.wikipedia.org/wiki/Language_model

20.8 Summary

In this tutorial, you discovered how to develop a word-based language model using a word embedding and a recurrent neural network. Specifically, you learned:

- How to prepare text for developing a word-based language model.
- How to design and fit a neural language model with a learned embedding and an LSTM hidden layer.
- How to use the learned language model to generate new text with similar statistical properties as the source text.

20.8.1 Next

This is the final chapter in the language modeling part. In the next part you will discover how to develop automatic caption generation for photographs.