Chapter 17

Neural Language Modeling

Language modeling is central to many important natural language processing tasks. Recently, neural-network-based language models have demonstrated better performance than classical methods both standalone and as part of more challenging natural language processing tasks. In this chapter, you will discover language modeling for natural language processing. After reading this chapter, you will know:

- Why language modeling is critical to addressing tasks in natural language processing.
- What a language model is and some examples of where they are used.
- How neural networks can be used for language modeling.

Let's get started.

17.1 Overview

This tutorial is divided into the following parts:

1. Problem of Modeling Language
2. Statistical Language Modeling
3. Neural Language Models

17.2 Problem of Modeling Language

Formal languages, like programming languages, can be fully specified. All the reserved words can be defined and the valid ways that they can be used can be precisely defined. We cannot do this with natural language. Natural languages are not designed; they emerge, and therefore there is no formal specification.

There may be formal rules and heuristics for parts of the language, but as soon as rules are defined, you will devise or encounter counter examples that contradict the rules. Natural languages involve vast numbers of terms that can be used in ways that introduce all kinds of ambiguities, yet can still be understood by other humans. Further, languages change, word

usages change: it is a moving target. Nevertheless, linguists try to specify the language with formal grammars and structures. It can be done, but it is very difficult and the results can be fragile. An alternative approach to specifying the model of the language is to learn it from examples.

17.3 Statistical Language Modeling

Statistical Language Modeling, or Language Modeling and LM for short, is the development of probabilistic models that are able to predict the next word in the sequence given the words that precede it.

> Language modeling is the task of assigning a probability to sentences in a language. [...] Besides assigning a probability to each sequence of words, the language models also assigns a probability for the likelihood of a given word (or a sequence of words) to follow a sequence of words
>
> — Page 105, *Neural Network Methods in Natural Language Processing*, 2017.

A language model learns the probability of word occurrence based on examples of text. Simpler models may look at a context of a short sequence of words, whereas larger models may work at the level of sentences or paragraphs. Most commonly, language models operate at the level of words.

> The notion of a language model is inherently probabilistic. A language model is a function that puts a probability measure over strings drawn from some vocabulary.
>
> — Page 238, *An Introduction to Information Retrieval*, 2008.

A language model can be developed and used standalone, such as to generate new sequences of text that appear to have come from the corpus. Language modeling is a root problem for a large range of natural language processing tasks. More practically, language models are used on the front-end or back-end of a more sophisticated model for a task that requires language understanding.

> ... language modeling is a crucial component in real-world applications such as machine-translation and automatic speech recognition, [...] For these reasons, language modeling plays a central role in natural-language processing, AI, and machine-learning research.
>
> — Page 105, *Neural Network Methods in Natural Language Processing*, 2017.

A good example is speech recognition, where audio data is used as an input to the model and the output requires a language model that interprets the input signal and recognizes each new word within the context of the words already recognized.

> Speech recognition is principally concerned with the problem of transcribing the speech signal as a sequence of words. [...] From this point of view, speech is assumed to be a generated by a language model which provides estimates of Pr(w) for all word strings w independently of the observed signal [...] The goal of speech recognition is to find the most likely word sequence given the observed acoustic signal.

— Pages 205-206, *The Oxford Handbook of Computational Linguistics*, 2005

Similarly, language models are used to generate text in many similar natural language processing tasks, for example:

- Optical Character Recognition
- Handwriting Recognition.
- Machine Translation.
- Spelling Correction.
- Image Captioning.
- Text Summarization
- And much more.

> Language modeling is the art of determining the probability of a sequence of words. This is useful in a large variety of areas including speech recognition, optical character recognition, handwriting recognition, machine translation, and spelling correction

— *A Bit of Progress in Language Modeling*, 2001.

Developing better language models often results in models that perform better on their intended natural language processing task. This is the motivation for developing better and more accurate language models.

> [language models] have played a key role in traditional NLP tasks such as speech recognition, machine translation, or text summarization. Often (although not always), training better language models improves the underlying metrics of the downstream task (such as word error rate for speech recognition, or BLEU score for translation), which makes the task of training better LMs valuable by itself.

— *Exploring the Limits of Language Modeling*, 2016.

17.4 Neural Language Models

Recently, the use of neural networks in the development of language models has become very popular, to the point that it may now be the preferred approach. The use of neural networks in language modeling is often called Neural Language Modeling, or NLM for short. Neural network approaches are achieving better results than classical methods both on standalone language models and when models are incorporated into larger models on challenging tasks like speech recognition and machine translation. A key reason for the leaps in improved performance may be the method's ability to generalize.

> Nonlinear neural network models solve some of the shortcomings of traditional language models: they allow conditioning on increasingly large context sizes with only a linear increase in the number of parameters, they alleviate the need for manually designing backoff orders, and they support generalization across different contexts.
>
> — Page 109, *Neural Network Methods in Natural Language Processing*, 2017.

Specifically, a word embedding is adopted that uses a real-valued vector to represent each word in a projected vector space. This learned representation of words based on their usage allows words with a similar meaning to have a similar representation.

> Neural Language Models (NLM) address the n-gram data sparsity issue through parameterization of words as vectors (word embeddings) and using them as inputs to a neural network. The parameters are learned as part of the training process. Word embeddings obtained through NLMs exhibit the property whereby semantically close words are likewise close in the induced vector space.
>
> — *Character-Aware Neural Language Model*, 2015.

This generalization is something that the representation used in classical statistical language models cannot easily achieve.

> "True generalization" is difficult to obtain in a discrete word indice space, since there is no obvious relation between the word indices.
>
> — *Connectionist language modeling for large vocabulary continuous speech recognition*, 2002.

Further, the distributed representation approach allows the embedding representation to scale better with the size of the vocabulary. Classical methods that have one discrete representation per word fight the curse of dimensionality with larger and larger vocabularies of words that result in longer and more sparse representations. The neural network approach to language modeling can be described using the three following model properties, taken from *A Neural Probabilistic Language Model*, 2003.

1. Associate each word in the vocabulary with a distributed word feature vector.
2. Express the joint probability function of word sequences in terms of the feature vectors of these words in the sequence.
3. Learn simultaneously the word feature vector and the parameters of the probability function.

This represents a relatively simple model where both the representation and probabilistic model are learned together directly from raw text data. Recently, the neural based approaches have started to outperform the classical statistical approaches.

> We provide ample empirical evidence to suggest that connectionist language models are superior to standard n-gram techniques, except their high computational (training) complexity.

— *Recurrent neural network based language model*, 2010.

Initially, feedforward neural network models were used to introduce the approach. More recently, recurrent neural networks and then networks with a long-term memory like the Long Short-Term Memory network, or LSTM, allow the models to learn the relevant context over much longer input sequences than the simpler feedforward networks.

> [an RNN language model] provides further generalization: instead of considering just several preceding words, neurons with input from recurrent connections are assumed to represent short term memory. The model learns itself from the data how to represent memory. While shallow feedforward neural networks (those with just one hidden layer) can only cluster similar words, recurrent neural network (which can be considered as a deep architecture) can perform clustering of similar histories. This allows for instance efficient representation of patterns with variable length.

— *Extensions of recurrent neural network language model*, 2011.

Recently, researchers have been seeking the limits of these language models. In the paper *Exploring the Limits of Language Modeling*, evaluating language models over large datasets, such as the corpus of one million words, the authors find that LSTM-based neural language models out-perform the classical methods.

> ... we have shown that RNN LMs can be trained on large amounts of data, and outperform competing models including carefully tuned N-grams.

— *Exploring the Limits of Language Modeling*, 2016.

Further, they propose some heuristics for developing high-performing neural language models in general:

- **Size matters**. The best models were the largest models, specifically number of memory units.
- **Regularization matters**. Use of regularization like dropout on input connections improves results.
- **CNNs vs Embeddings**. Character-level Convolutional Neural Network (CNN) models can be used on the front-end instead of word embeddings, achieving similar and sometimes better results.
- **Ensembles matter**. Combining the prediction from multiple models can offer large improvements in model performance.

17.5 Further Reading

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

17.5.1 Books

- *Neural Network Methods in Natural Language Processing*, 2017.
  http://amzn.to/2vStiIS
- *Natural Language Processing, Artificial Intelligence A Modern Approach*, 2009.
  http://amzn.to/2fDPfF3
- *Language models for information retrieval, An Introduction to Information Retrieval*, 2008.
  http://amzn.to/2vAavQd

17.5.2 Papers

- *A Neural Probabilistic Language Model*, NIPS, 2001.
  https://papers.nips.cc/paper/1839-a-neural-probabilistic-language-model.pdf
- *A Neural Probabilistic Language Model*, JMLR, 2003.
  http://www.jmlr.org/papers/v3/bengio03a.html
- *Connectionist language modeling for large vocabulary continuous speech recognition*, 2002.
  https://pdfs.semanticscholar.org/b4db/83366f925e9a1e1528ee9f6b41d7cd666f41.pdf
- *Recurrent neural network based language model*, 2010.
  http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf
- *Extensions of recurrent neural network language model*, 2011.
  http://ieeexplore.ieee.org/abstract/document/5947611/
- *Character-Aware Neural Language Model*, 2015.
  https://arxiv.org/abs/1508.06615
- *LSTM Neural Networks for Language Modeling*, 2012.
  https://pdfs.semanticscholar.org/f9a1/b3850dfd837793743565a8af95973d395a4e.pdf
- *Exploring the Limits of Language Modeling*, 2016.
  https://arxiv.org/abs/1602.02410

17.5.3 Articles

- Language Model, Wikipedia.
  https://en.wikipedia.org/wiki/Language_model
- Neural net language models, Scholarpedia.
  http://www.scholarpedia.org/article/Neural_net_language_models

17.6 Summary

In this chapter, you discovered language modeling for natural language processing tasks. Specifically, you learned:

- That natural language is not formally specified and requires the use of statistical models to learn from examples.
- That statistical language models are central to many challenging natural language processing tasks.
- That state-of-the-art results are achieved using neural language models, specifically those with word embeddings and recurrent neural network algorithms.

17.6.1 Next

In the next chapter, you will discover how you can develop a character-based neural language model.

Chapter 18

How to Develop a Character-Based Neural Language Model

A language model predicts the next word in the sequence based on the specific words that have come before it in the sequence. It is also possible to develop language models at the character level using neural networks. The benefit of character-based language models is their small vocabulary and flexibility in handling any words, punctuation, and other document structure. This comes at the cost of requiring larger models that are slower to train. Nevertheless, in the field of neural language models, character-based models offer a lot of promise for a general, flexible and powerful approach to language modeling. In this tutorial, you will discover how to develop a character-based neural language model. After completing this tutorial, you will know:

- How to prepare text for character-based language modeling.
- How to develop a character-based language model using LSTMs.
- How to use a trained character-based language model to generate text.

Let's get started.

18.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Sing a Song of Sixpence
2. Data Preparation
3. Train Language Model
4. Generate Text

18.2 Sing a Song of Sixpence

The nursery rhyme *Sing a Song of Sixpence* is well known in the west. The first verse is common, but there is also a 4 verse version that we will use to develop our character-based language model. It is short, so fitting the model will be fast, but not so short that we won't see anything interesting. The complete 4 verse version we will use as source text is listed below.

```
Sing a song of sixpence,
A pocket full of rye.
Four and twenty blackbirds,
Baked in a pie.

When the pie was opened
The birds began to sing;
Wasn't that a dainty dish,
To set before the king.

The king was in his counting house,
Counting out his money;
The queen was in the parlour,
Eating bread and honey.

The maid was in the garden,
Hanging out the clothes,
When down came a blackbird
And pecked off her nose.
```

Listing 18.1: *Sing a Song of Sixpence* nursery rhyme.

Copy the text and save it in a new file in your current working directory with the file name `rhyme.txt`.

18.3 Data Preparation

The first step is to prepare the text data. We will start by defining the type of language model.

18.3.1 Language Model Design

A language model must be trained on the text, and in the case of a character-based language model, the input and output sequences must be characters. The number of characters used as input will also define the number of characters that will need to be provided to the model in order to elicit the first predicted character. After the first character has been generated, it can be appended to the input sequence and used as input for the model to generate the next character.

Longer sequences offer more context for the model to learn what character to output next but take longer to train and impose more burden on seeding the model when generating text. We will use an arbitrary length of 10 characters for this model. There is not a lot of text, and 10 characters is a few words. We can now transform the raw text into a form that our model can learn; specifically, input and output sequences of characters.

18.3.2 Load Text

We must load the text into memory so that we can work with it. Below is a function named `load_doc()` that will load a text file given a filename and return the 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 18.2: Function to load a document into memory.

We can call this function with the filename of the nursery rhyme `rhyme.txt` to load the text into memory. The contents of the file are then printed to screen as a sanity check.

```
# load text
raw_text = load_doc('rhyme.txt')
print(raw_text)
```

Listing 18.3: Load the document into memory.

18.3.3 Clean Text

Next, we need to clean the loaded text. We will not do much to it on this example. Specifically, we will strip all of the new line characters so that we have one long sequence of characters separated only by white space.

```
# clean
tokens = raw_text.split()
raw_text = ' '.join(tokens)
```

Listing 18.4: Tokenize the loaded document.

You may want to explore other methods for data cleaning, such as normalizing the case to lowercase or removing punctuation in an effort to reduce the final vocabulary size and develop a smaller and leaner model.

18.3.4 Create Sequences

Now that we have a long list of characters, we can create our input-output sequences used to train the model. Each input sequence will be 10 characters with one output character, making each sequence 11 characters long. We can create the sequences by enumerating the characters in the text, starting at the 11th character at index 10.

```
# organize into sequences of characters
length = 10
sequences = list()
for i in range(length, len(raw_text)):
  # select sequence of tokens
  seq = raw_text[i-length:i+1]
```

```
  # store
  sequences.append(seq)
print('Total Sequences: %d' % len(sequences))
```

Listing 18.5: Convert text into fixed-length sequences.

Running this snippet, we can see that we end up with just under 400 sequences of characters for training our language model.

```
Total Sequences: 399
```

Listing 18.6: Example output of converting text into fixed-length sequences.

18.3.5 Save Sequences

Finally, we can save the prepared data to file so that we can load it later when we develop our model. Below is a function `save_doc()` that, given a list of strings and a filename, will save the strings to file, one per line.

```
# 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 18.7: Function to save sequences to file.

We can call this function and save our prepared sequences to the filename `char_sequences.txt` in our current working directory.

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

Listing 18.8: Call function to save sequences to file.

18.3.6 Complete Example

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

```
# 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

# 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 text
raw_text = load_doc('rhyme.txt')
print(raw_text)
# clean
tokens = raw_text.split()
raw_text = ' '.join(tokens)
# organize into sequences of characters
length = 10
sequences = list()
for i in range(length, len(raw_text)):
  # select sequence of tokens
  seq = raw_text[i-length:i+1]
  # store
  sequences.append(seq)
print('Total Sequences: %d' % len(sequences))
# save sequences to file
out_filename = 'char_sequences.txt'
save_doc(sequences, out_filename)
```

Listing 18.9: Complete example of preparing the text data.

Run the example to create the `char_sequences.txt` file. Take a look inside you should see something like the following:

```
Sing a song
ing a song
ng a song o
g a song of
 a song of
a song of s
 song of si
song of six
ong of sixp
ng of sixpe
...
```

Listing 18.10: Sample of the output file.

We are now ready to train our character-based neural language model.

18.4 Train Language Model

In this section, we will develop a neural language model for the prepared sequence data. The model will read encoded characters and predict the next character in the sequence. A Long Short-Term Memory recurrent neural network hidden layer will be used to learn the context from the input sequence in order to make the predictions.

18.4.1 Load Data

The first step is to load the prepared character sequence data from `char_sequences.txt`. We can use the same `load_doc()` function developed in the previous section. Once loaded, we split

the text by new line to give a list of sequences ready to be encoded.

```
# 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 = 'char_sequences.txt'
raw_text = load_doc(in_filename)
lines = raw_text.split('\n')
```

Listing 18.11: Load the prepared text data.

18.4.2 Encode Sequences

The sequences of characters must be encoded as integers. This means that each unique character will be assigned a specific integer value and each sequence of characters will be encoded as a sequence of integers. We can create the mapping given a sorted set of unique characters in the raw input data. The mapping is a dictionary of character values to integer values.

```
chars = sorted(list(set(raw_text)))
mapping = dict((c, i) for i, c in enumerate(chars))
```

Listing 18.12: Create a mapping between chars and integers.

Next, we can process each sequence of characters one at a time and use the dictionary mapping to look up the integer value for each character.

```
sequences = list()
for line in lines:
  # integer encode line
  encoded_seq = [mapping[char] for char in line]
  # store
  sequences.append(encoded_seq)
```

Listing 18.13: Integer encode sequences of characters.

The result is a list of integer lists. We need to know the size of the vocabulary later. We can retrieve this as the size of the dictionary mapping.

```
# vocabulary size
vocab_size = len(mapping)
print('Vocabulary Size: %d' % vocab_size)
```

Listing 18.14: Summarize the size of the vocabulary.

Running this piece, we can see that there are 38 unique characters in the input sequence data.

```
Vocabulary Size: 38
```

Listing 18.15: Example output from summarizing the size of the vocabulary.

18.4.3 Split Inputs and Output

Now that the sequences have been integer encoded, we can separate the columns into input and output sequences of characters. We can do this using a simple array slice.

```
sequences = array(sequences)
X, y = sequences[:,:-1], sequences[:,-1]
```

Listing 18.16: Split sequences into input and output elements.

Next, we need to one hot encode each character. That is, each character becomes a vector as long as the vocabulary (38 elements) with a 1 marked for the specific character. This provides a more precise input representation for the network. It also provides a clear objective for the network to predict, where a probability distribution over characters can be output by the model and compared to the ideal case of all 0 values with a 1 for the actual next character. We can use the `to_categorical()` function in the Keras API to one hot encode the input and output sequences.

```
sequences = [to_categorical(x, num_classes=vocab_size) for x in X]
X = array(sequences)
y = to_categorical(y, num_classes=vocab_size)
```

Listing 18.17: Convert sequences into a format ready for training.

We are now ready to fit the model.

18.4.4 Fit Model

The model is defined with an input layer that takes sequences that have 10 time steps and 38 features for the one hot encoded input sequences. Rather than specify these numbers, we use the second and third dimensions on the X input data. This is so that if we change the length of the sequences or size of the vocabulary, we do not need to change the model definition. The model has a single LSTM hidden layer with 75 memory cells, chosen with a little trial and error. The model has a fully connected output layer that outputs one vector with a probability distribution across all characters in the vocabulary. A softmax activation function is used on the output layer to ensure the output has the properties of a probability distribution.

```
# define the model
def define_model(X):
  model = Sequential()
  model.add(LSTM(75, input_shape=(X.shape[1], X.shape[2])))
  model.add(Dense(vocab_size, activation='softmax'))
  # compile model
  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 18.18: Define the language model.

The model is learning a multiclass classification problem, therefore we use the categorical log loss intended for this type of problem. The efficient Adam implementation of gradient descent is used to optimize the model and accuracy is reported at the end of each batch update. The

model is fit for 100 training epochs, again found with a little trial and error. Running this prints a summary of the defined network as a sanity check.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 75)                34200
_________________________________________________________________
dense_1 (Dense)              (None, 38)                2888
=================================================================
Total params: 37,088
Trainable params: 37,088
Non-trainable params: 0
_________________________________________________________________
```

Listing 18.19: Example output from summarizing the defined model.

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

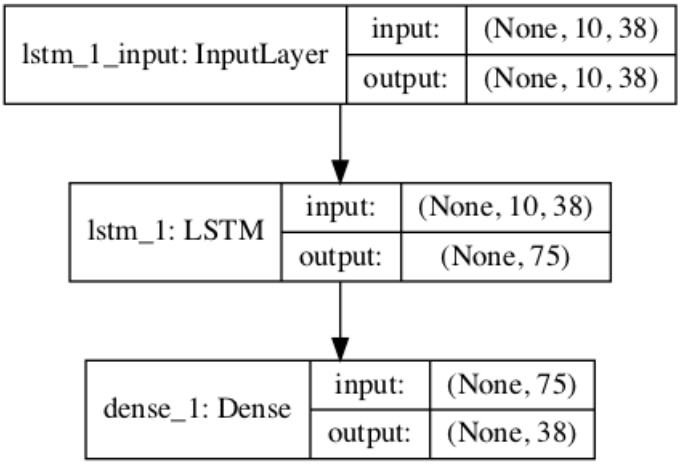

Figure 18.1: Plot of the defined character-based language model.

18.4.5 Save Model

After the model is fit, we save it to file for later use. The Keras model API provides the `save()` function that we can use to save the model to a single file, including weights and topology information.

```
# save the model to file
model.save('model.h5')
```

Listing 18.20: Save the fit model to file.

We also save the mapping from characters to integers that we will need to encode any input when using the model and decode any output from the model.

```
# save the mapping
dump(mapping, open('mapping.pkl', 'wb'))
```

Listing 18.21: Save the mapping of chars to integers to file.

18.4.6 Complete Example

Tying all of this together, the complete code listing for fitting the character-based neural language model is listed below.

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

# 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(X):
  model = Sequential()
  model.add(LSTM(75, input_shape=(X.shape[1], X.shape[2])))
  model.add(Dense(vocab_size, activation='softmax'))
  # compile model
  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 = 'char_sequences.txt'
raw_text = load_doc(in_filename)
lines = raw_text.split('\n')
# integer encode sequences of characters
chars = sorted(list(set(raw_text)))
mapping = dict((c, i) for i, c in enumerate(chars))
sequences = list()
for line in lines:
  # integer encode line
  encoded_seq = [mapping[char] for char in line]
  # store
  sequences.append(encoded_seq)
# vocabulary size
vocab_size = len(mapping)
print('Vocabulary Size: %d' % vocab_size)
# separate into input and output
sequences = array(sequences)
X, y = sequences[:,:-1], sequences[:,-1]
sequences = [to_categorical(x, num_classes=vocab_size) for x in X]
X = array(sequences)
```

```
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(X)
# fit model
model.fit(X, y, epochs=100, verbose=2)
# save the model to file
model.save('model.h5')
# save the mapping
dump(mapping, open('mapping.pkl', 'wb'))
```

Listing 18.22: Complete example of training the language model.

Running the example might take one minute. You will see that the model learns the problem well, perhaps too well for generating surprising sequences of characters.

```
...
Epoch 96/100
0s - loss: 0.2193 - acc: 0.9950
Epoch 97/100
0s - loss: 0.2124 - acc: 0.9950
Epoch 98/100
0s - loss: 0.2054 - acc: 0.9950
Epoch 99/100
0s - loss: 0.1982 - acc: 0.9950
Epoch 100/100
0s - loss: 0.1910 - acc: 0.9950
```

Listing 18.23: Example output from training the language model.

At the end of the run, you will have two files saved to the current working directory, specifically `model.h5` and `mapping.pkl`. Next, we can look at using the learned model.

18.5 Generate Text

We will use the learned language model to generate new sequences of text that have the same statistical properties.

18.5.1 Load Model

The first step is to load the model saved to the file `model.h5`. We can use the `load_model()` function from the Keras API.

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

Listing 18.24: Load the saved model.

We also need to load the pickled dictionary for mapping characters to integers from the file `mapping.pkl`. We will use the Pickle API to load the object.

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

Listing 18.25: Load the saved mapping from chars to integers.

We are now ready to use the loaded model.

18.5.2 Generate Characters

We must provide sequences of 10 characters as input to the model in order to start the generation process. We will pick these manually. A given input sequence will need to be prepared in the same way as preparing the training data for the model. First, the sequence of characters must be integer encoded using the loaded mapping.

```
# encode the characters as integers
encoded = [mapping[char] for char in in_text]
```

Listing 18.26: Encode input text to integers.

Next, the integers need to be one hot encoded using the `to_categorical()` Keras function. We also need to reshape the sequence to be 3-dimensional, as we only have one sequence and LSTMs require all input to be three dimensional (samples, time steps, features).

```
# one hot encode
encoded = to_categorical(encoded, num_classes=len(mapping))
encoded = encoded.reshape(1, encoded.shape[0], encoded.shape[1])
```

Listing 18.27: One hot encode the integer encoded text.

We can then use the model to predict the next character in the sequence. We use `predict_classes()` instead of `predict()` to directly select the integer for the character with the highest probability instead of getting the full probability distribution across the entire set of characters.

```
# predict character
yhat = model.predict_classes(encoded, verbose=0)
```

Listing 18.28: Predict the next character in the sequence.

We can then decode this integer by looking up the mapping to see the character to which it maps.

```
out_char = ''
for char, index in mapping.items():
  if index == yhat:
    out_char = char
    break
```

Listing 18.29: Map the predicted integer back to a character.

This character can then be added to the input sequence. We then need to make sure that the input sequence is 10 characters by truncating the first character from the input sequence text. We can use the `pad_sequences()` function from the Keras API that can perform this truncation operation. Putting all of this together, we can define a new function named `generate_seq()` for using the loaded model to generate new sequences of text.

```
# generate a sequence of characters with a language model
def generate_seq(model, mapping, seq_length, seed_text, n_chars):
  in_text = seed_text
  # generate a fixed number of characters
  for _ in range(n_chars):
    # encode the characters as integers
    encoded = [mapping[char] for char in in_text]
    # truncate sequences to a fixed length
```

```
    encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
    # one hot encode
    encoded = to_categorical(encoded, num_classes=len(mapping))
    encoded = encoded.reshape(1, encoded.shape[0], encoded.shape[1])
    # predict character
    yhat = model.predict_classes(encoded, verbose=0)
    # reverse map integer to character
    out_char = ''
    for char, index in mapping.items():
      if index == yhat:
        out_char = char
        break
    # append to input
    in_text += char
  return in_text
```

Listing 18.30: Function to predict a sequence of characters given seed text.

18.5.3 Complete Example

Tying all of this together, the complete example for generating text using the fit neural language model is listed below.

```
from pickle import load
from keras.models import load_model
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences

# generate a sequence of characters with a language model
def generate_seq(model, mapping, seq_length, seed_text, n_chars):
  in_text = seed_text
  # generate a fixed number of characters
  for _ in range(n_chars):
    # encode the characters as integers
    encoded = [mapping[char] for char in in_text]
    # truncate sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
    # one hot encode
    encoded = to_categorical(encoded, num_classes=len(mapping))
    encoded = encoded.reshape(1, encoded.shape[0], encoded.shape[1])
    # predict character
    yhat = model.predict_classes(encoded, verbose=0)
    # reverse map integer to character
    out_char = ''
    for char, index in mapping.items():
      if index == yhat:
        out_char = char
        break
    # append to input
    in_text += out_char
  return in_text

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

```
mapping = load(open('mapping.pkl', 'rb'))
# test start of rhyme
print(generate_seq(model, mapping, 10, 'Sing a son', 20))
# test mid-line
print(generate_seq(model, mapping, 10, 'king was i', 20))
# test not in original
print(generate_seq(model, mapping, 10, 'hello worl', 20))
```

Listing 18.31: Complete example of generating characters with the fit model.

Running the example generates three sequences of text. The first is a test to see how the model does at starting from the beginning of the rhyme. The second is a test to see how well it does at beginning in the middle of a line. The final example is a test to see how well it does with a sequence of characters never seen before.

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

```
Sing a song of sixpence, A poc
king was in his counting house
hello worls e pake wofey. The
```

Listing 18.32: Example output from generating sequences of characters.

We can see that the model did very well with the first two examples, as we would expect. We can also see that the model still generated something for the new text, but it is nonsense.

18.6 Further Reading

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

- Sing a Song of Sixpence on Wikipedia.
  https://en.wikipedia.org/wiki/Sing_a_Song_of_Sixpence
- Keras Utils API.
  https://keras.io/utils/
- Keras Sequence Processing API.
  https://keras.io/preprocessing/sequence/

18.7 Summary

In this tutorial, you discovered how to develop a character-based neural language model. Specifically, you learned:

- How to prepare text for character-based language modeling.
- How to develop a character-based language model using LSTMs.
- How to use a trained character-based language model to generate text.

18.7.1 Next

In the next chapter, you will discover how you can develop a word-based neural language model.

Chapter 19

How to Develop a Word-Based Neural Language Model

Language modeling involves predicting the next word in a sequence given the sequence of words already present. A language model is a key element in many natural language processing models such as machine translation and speech recognition. The choice of how the language model is framed must match how the language model is intended to be used. In this tutorial, you will discover how the framing of a language model affects the skill of the model when generating short sequences from a nursery rhyme. After completing this tutorial, you will know:

- The challenge of developing a good framing of a word-based language model for a given application.
- How to develop one-word, two-word, and line-based framings for word-based language models.
- How to generate sequences using a fit language model.

Let's get started.

19.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Framing Language Modeling
2. Jack and Jill Nursery Rhyme
3. Model 1: One-Word-In, One-Word-Out Sequences
4. Model 2: Line-by-Line Sequence
5. Model 3: Two-Words-In, One-Word-Out Sequence

19.2 Framing Language Modeling

A statistical language model is learned from raw text and predicts the probability of the next word in the sequence given the words already present in the sequence. Language models are a key component in larger models for challenging natural language processing problems, like machine translation and speech recognition. They can also be developed as standalone models and used for generating new sequences that have the same statistical properties as the source text.

Language models both learn and predict one word at a time. The training of the network involves providing sequences of words as input that are processed one at a time where a prediction can be made and learned for each input sequence. Similarly, when making predictions, the process can be seeded with one or a few words, then predicted words can be gathered and presented as input on subsequent predictions in order to build up a generated output sequence

Therefore, each model will involve splitting the source text into input and output sequences, such that the model can learn to predict words. There are many ways to frame the sequences from a source text for language modeling. In this tutorial, we will explore 3 different ways of developing word-based language models in the Keras deep learning library. There is no single best approach, just different framings that may suit different applications.

19.3 Jack and Jill Nursery Rhyme

Jack and Jill is a simple nursery rhyme. It is comprised of 4 lines, as follows:

```
Jack and Jill went up the hill
To fetch a pail of water
Jack fell down and broke his crown
And Jill came tumbling after
```

Listing 19.1: *Jack and Jill* nursery rhyme.

We will use this as our source text for exploring different framings of a word-based language model. We can define this text in Python as follows:

```
# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
    Jack fell down and broke his crown\n
    And Jill came tumbling after\n """
```

Listing 19.2: Sample text for this tutorial.

19.4 Model 1: One-Word-In, One-Word-Out Sequences

We can start with a very simple model. Given one word as input, the model will learn to predict the next word in the sequence. For example:

```
X,      y
Jack,   and
and,    Jill
Jill,   went
```

```
...
```

Listing 19.3: Example of input and output pairs.

The first step is to encode the text as integers. Each lowercase word in the source text is assigned a unique integer and we can convert the sequences of words to sequences of integers. Keras provides the `Tokenizer` class that can be used to perform this encoding. First, the `Tokenizer` is fit on the source text to develop the mapping from words to unique integers. Then sequences of text can be converted to sequences of integers by calling the `texts_to_sequences()` function.

```
# integer encode text
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
encoded = tokenizer.texts_to_sequences([data])[0]
```

Listing 19.4: Example of training a `Tokenizer` on the sample text.

We will need to know the size of the vocabulary later for both defining the word embedding layer in the model, and for encoding output words using a one hot encoding. The size of the vocabulary can be retrieved from the trained `Tokenizer` by accessing the `word_index` attribute.

```
# determine the vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
```

Listing 19.5: Summarize the size of the vocabulary.

Running this example, we can see that the size of the vocabulary is 21 words. We add one, because we will need to specify the integer for the largest encoded word as an array index, e.g. words encoded 1 to 21 with array indicies 0 to 21 or 22 positions. Next, we need to create sequences of words to fit the model with one word as input and one word as output.

```
# create word -> word sequences
sequences = list()
for i in range(1, len(encoded)):
  sequence = encoded[i-1:i+1]
  sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
```

Listing 19.6: Example of encoding the source text.

Running this piece shows that we have a total of 24 input-output pairs to train the network.

```
Total Sequences: 24
```

Listing 19.7: Example of output of summarizing the encoded text.

We can then split the sequences into input (`X`) and output elements (`y`). This is straightforward as we only have two columns in the data.

```
# split into X and y elements
sequences = array(sequences)
X, y = sequences[:,0],sequences[:,1]
```

Listing 19.8: Split the encoded text into input and output pairs.

We will fit our model to predict a probability distribution across all words in the vocabulary. That means that we need to turn the output element from a single integer into a one hot encoding with a 0 for every word in the vocabulary and a 1 for the actual word that the value. This gives the network a ground truth to aim for from which we can calculate error and update the model. Keras provides the `to_categorical()` function that we can use to convert the integer to a one hot encoding while specifying the number of classes as the vocabulary size.

```
# one hot encode outputs
y = to_categorical(y, num_classes=vocab_size)
```

Listing 19.9: One hot encode the output words.

We are now ready to define the neural network model. The model uses a learned word embedding in the input layer. This has one real-valued vector for each word in the vocabulary, where each word vector has a specified length. In this case we will use a 10-dimensional projection. The input sequence contains a single word, therefore the `input_length=1`. The model has a single hidden LSTM layer with 50 units. This is far more than is needed. The output layer is comprised of one neuron for each word in the vocabulary and uses a softmax activation function to ensure the output is normalized to look like a probability.

```
# define the model
def define_model(vocab_size):
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=1))
  model.add(LSTM(50))
  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 19.10: Define and compile the language model.

The structure of the network can be summarized as follows:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 1, 10)             220
_________________________________________________________________
lstm_1 (LSTM)                (None, 50)                12200
_________________________________________________________________
dense_1 (Dense)              (None, 22)                1122
=================================================================
Total params: 13,542
Trainable params: 13,542
Non-trainable params: 0
_________________________________________________________________
```

Listing 19.11: Example output summarizing the defined model.

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

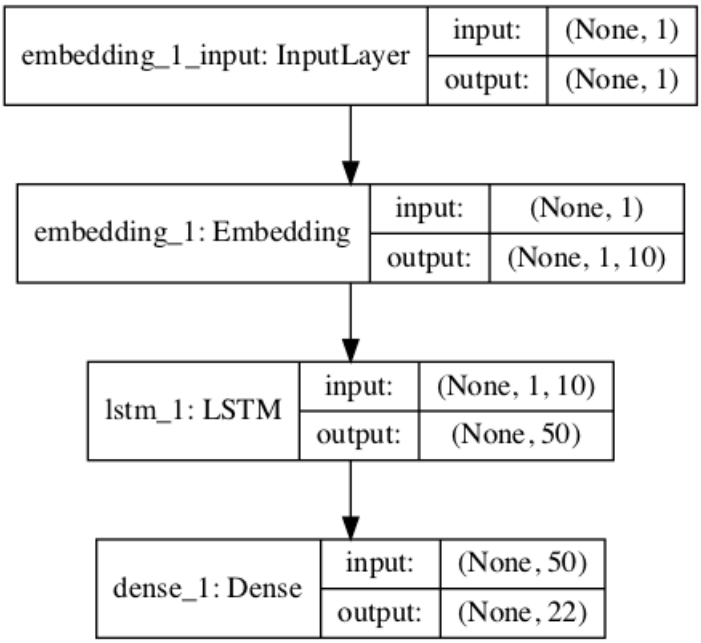

Figure 19.1: Plot of the defined word-based language model.

We will use this same general network structure for each example in this tutorial, with minor changes to the learned embedding layer. We can compile and fit the network on the encoded text data. Technically, we are modeling a multiclass classification problem (predict the word in the vocabulary), therefore using the categorical cross entropy loss function. We use the efficient Adam implementation of gradient descent and track accuracy at the end of each epoch. The model is fit for 500 training epochs, again, perhaps more than is needed. The network configuration was not tuned for this and later experiments; an over-prescribed configuration was chosen to ensure that we could focus on the framing of the language model.

After the model is fit, we test it by passing it a given word from the vocabulary and having the model predict the next word. Here we pass in '*Jack*' by encoding it and calling `model.predict_classes()` to get the integer output for the predicted word. This is then looked up in the vocabulary mapping to give the associated word.

```
# evaluate
in_text = 'Jack'
print(in_text)
encoded = tokenizer.texts_to_sequences([in_text])[0]
encoded = array(encoded)
yhat = model.predict_classes(encoded, verbose=0)
for word, index in tokenizer.word_index.items():
  if index == yhat:
    print(word)
```

Listing 19.12: Evaluate the fit language model.

This process could then be repeated a few times to build up a generated sequence of words. To make this easier, we wrap up the behavior in a function that we can call by passing in our model and the seed word.

```
# generate a sequence from the model
def generate_seq(model, tokenizer, seed_text, n_words):
```

```
  in_text, result = seed_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]
    encoded = array(encoded)
    # predict a word in the vocabulary
    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, result = out_word, result + ' ' + out_word
  return result
```

Listing 19.13: Function to generate output sequences given a fit model.

We can tie all of this together. The complete code listing is provided below.

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

# generate a sequence from the model
def generate_seq(model, tokenizer, seed_text, n_words):
  in_text, result = seed_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]
    encoded = array(encoded)
    # predict a word in the vocabulary
    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, result = out_word, result + ' ' + out_word
  return result

# define the model
def define_model(vocab_size):
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=1))
  model.add(LSTM(50))
  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

# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
    Jack fell down and broke his crown\n
    And Jill came tumbling after\n """
# integer encode text
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
encoded = tokenizer.texts_to_sequences([data])[0]
# determine the vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# create word -> word sequences
sequences = list()
for i in range(1, len(encoded)):
  sequence = encoded[i-1:i+1]
  sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
# split into X and y elements
sequences = array(sequences)
X, y = sequences[:,0],sequences[:,1]
# one hot encode outputs
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(vocab_size)
# fit network
model.fit(X, y, epochs=500, verbose=2)
# evaluate
print(generate_seq(model, tokenizer, 'Jack', 6))
```

Listing 19.14: Complete example of model1.

Running the example prints the loss and accuracy each training epoch.

```
...
Epoch 496/500
0s - loss: 0.2358 - acc: 0.8750
Epoch 497/500
0s - loss: 0.2355 - acc: 0.8750
Epoch 498/500
0s - loss: 0.2352 - acc: 0.8750
Epoch 499/500
0s - loss: 0.2349 - acc: 0.8750
Epoch 500/500
0s - loss: 0.2346 - acc: 0.8750
```

Listing 19.15: Example output of fitting the language model.

We can see that the model does not memorize the source sequences, likely because there is some ambiguity in the input sequences, for example:

```
jack => and
jack => fell
```

Listing 19.16: Example output of predicting the next word.

And so on. At the end of the run, *Jack* is passed in and a prediction or new sequence is generated. We get a reasonable sequence as output that has some elements of the source.

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

```
Jack and jill came tumbling after down
```

Listing 19.17: Example output of predicting a sequence of words.

This is a good first cut language model, but does not take full advantage of the LSTM's ability to handle sequences of input and disambiguate some of the ambiguous pairwise sequences by using a broader context.

19.5 Model 2: Line-by-Line Sequence

Another approach is to split up the source text line-by-line, then break each line down into a series of words that build up. For example:

```
X,                              y
_, _, _, _, _, Jack,            and
_, _, _, _, Jack, and,          Jill
_, _, _, Jack, and, Jill,       went
_, _, Jack, and, Jill, went,    up
_, Jack, and, Jill, went, up,   the
Jack, and, Jill, went, up, the, hill
```

Listing 19.18: Example framing of the problem as sequences of words.

This approach may allow the model to use the context of each line to help the model in those cases where a simple one-word-in-and-out model creates ambiguity. In this case, this comes at the cost of predicting words across lines, which might be fine for now if we are only interested in modeling and generating lines of text. Note that in this representation, we will require a padding of sequences to ensure they meet a fixed length input. This is a requirement when using Keras. First, we can create the sequences of integers, line-by-line by using the `Tokenizer` already fit on the source text.

```
# create line-based sequences
sequences = list()
for line in data.split('\n'):
  encoded = tokenizer.texts_to_sequences([line])[0]
  for i in range(1, len(encoded)):
    sequence = encoded[:i+1]
    sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
```

Listing 19.19: Example of preparing sequences of words.

Next, we can pad the prepared sequences. We can do this using the `pad_sequences()` function provided in Keras. This first involves finding the longest sequence, then using that as the length by which to pad-out all other sequences.

```
# pad input sequences
max_length = max([len(seq) for seq in sequences])
sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
print('Max Sequence Length: %d' % max_length)
```

Listing 19.20: Example of padding sequences of words.

Next, we can split the sequences into input and output elements, much like before.

```
# split into input and output elements
sequences = array(sequences)
X, y = sequences[:,:-1],sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
```

Listing 19.21: Example of preparing the input and output sequences.

The model can then be defined as before, except the input sequences are now longer than a single word. Specifically, they are `max_length-1` in length, -1 because when we calculated the maximum length of sequences, they included the input and output elements.

```
# define the model
def define_model(vocab_size, max_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=max_length-1))
  model.add(LSTM(50))
  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 19.22: Define and compile the language model.

We can use the model to generate new sequences as before. The `generate_seq()` function can be updated to build up an input sequence by adding predictions to the list of input words each iteration.

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, max_length, seed_text, n_words):
  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]
    # pre-pad sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=max_length, padding='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
  return in_text
```

Listing 19.23: Function to generate sequences of words given input text.

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

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

# generate a sequence from a language model
def generate_seq(model, tokenizer, max_length, seed_text, n_words):
  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]
    # pre-pad sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=max_length, padding='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
  return in_text

# define the model
def define_model(vocab_size, max_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=max_length-1))
  model.add(LSTM(50))
  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

# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
```

```
    Jack fell down and broke his crown\n
    And Jill came tumbling after\n """
# prepare the tokenizer on the source text
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
# determine the vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# create line-based sequences
sequences = list()
for line in data.split('\n'):
  encoded = tokenizer.texts_to_sequences([line])[0]
  for i in range(1, len(encoded)):
    sequence = encoded[:i+1]
    sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
# pad input sequences
max_length = max([len(seq) for seq in sequences])
sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
print('Max Sequence Length: %d' % max_length)
# split into input and output elements
sequences = array(sequences)
X, y = sequences[:,:-1],sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(vocab_size, max_length)
# fit network
model.fit(X, y, epochs=500, verbose=2)
# evaluate model
print(generate_seq(model, tokenizer, max_length-1, 'Jack', 4))
print(generate_seq(model, tokenizer, max_length-1, 'Jill', 4))
```

Listing 19.24: Complete example of model2.

Running the example achieves a better fit on the source data. The added context has allowed the model to disambiguate some of the examples. There are still two lines of text that start with "*Jack*" that may still be a problem for the network.

```
...
Epoch 496/500
0s - loss: 0.1039 - acc: 0.9524
Epoch 497/500
0s - loss: 0.1037 - acc: 0.9524
Epoch 498/500
0s - loss: 0.1035 - acc: 0.9524
Epoch 499/500
0s - loss: 0.1033 - acc: 0.9524
Epoch 500/500
0s - loss: 0.1032 - acc: 0.9524
```

Listing 19.25: Example output of fitting the language model.

At the end of the run, we generate two sequences with different seed words: *Jack* and *Jill*.

The first generated line looks good, directly matching the source text. The second is a bit strange. This makes sense, because the network only ever saw *Jill* within an input sequence, not at the beginning of the sequence, so it has forced an output to use the word *Jill*, i.e. the

last line of the rhyme.

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

```
Jack fell down and broke
Jill jill came tumbling after
```

Listing 19.26: Example output of generating sequences of words.

This was a good example of how the framing may result in better new lines, but not good partial lines of input.

19.6 Model 3: Two-Words-In, One-Word-Out Sequence

We can use an intermediate between the one-word-in and the whole-sentence-in approaches and pass in a sub-sequences of words as input. This will provide a trade-off between the two framings allowing new lines to be generated and for generation to be picked up mid line. We will use 3 words as input to predict one word as output. The preparation of the sequences is much like the first example, except with different offsets in the source sequence arrays, as follows:

```
# encode 2 words -> 1 word
sequences = list()
for i in range(2, len(encoded)):
  sequence = encoded[i-2:i+1]
  sequences.append(sequence)
```

Listing 19.27: Example of preparing constrained sequence data.

The complete example is listed below

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

# generate a sequence from a language model
def generate_seq(model, tokenizer, max_length, seed_text, n_words):
  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]
    # pre-pad sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=max_length, padding='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
  return in_text

# define the model
def define_model(vocab_size, max_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=max_length-1))
  model.add(LSTM(50))
  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

# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
    Jack fell down and broke his crown\n
    And Jill came tumbling after\n """
# integer encode sequences of words
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
encoded = tokenizer.texts_to_sequences([data])[0]
# retrieve vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# encode 2 words -> 1 word
sequences = list()
for i in range(2, len(encoded)):
  sequence = encoded[i-2:i+1]
  sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
# pad sequences
max_length = max([len(seq) for seq in sequences])
sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
print('Max Sequence Length: %d' % max_length)
# split into input and output elements
sequences = array(sequences)
X, y = sequences[:,:-1],sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(vocab_size, max_length)
# fit network
model.fit(X, y, epochs=500, verbose=2)
# evaluate model
print(generate_seq(model, tokenizer, max_length-1, 'Jack and', 5))
print(generate_seq(model, tokenizer, max_length-1, 'And Jill', 3))
print(generate_seq(model, tokenizer, max_length-1, 'fell down', 5))
print(generate_seq(model, tokenizer, max_length-1, 'pail of', 5))
```

Listing 19.28: Complete example of model3.

Running the example again gets a good fit on the source text at around 95% accuracy.

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

```
...
Epoch 496/500
0s - loss: 0.0685 - acc: 0.9565
Epoch 497/500
0s - loss: 0.0685 - acc: 0.9565
Epoch 498/500
0s - loss: 0.0684 - acc: 0.9565
Epoch 499/500
0s - loss: 0.0684 - acc: 0.9565
Epoch 500/500
0s - loss: 0.0684 - acc: 0.9565
```

Listing 19.29: Example output of fitting the language model.

We look at 4 generation examples, two start of line cases and two starting mid line.

```
Jack and jill went up the hill
And Jill went up the
fell down and broke his crown and
pail of water jack fell down and
```

Listing 19.30: Example output of generating sequences of words.

The first start of line case generated correctly, but the second did not. The second case was an example from the 4th line, which is ambiguous with content from the first line. Perhaps a further expansion to 3 input words would be better. The two mid-line generation examples were generated correctly, matching the source text.

We can see that the choice of how the language model is framed and the requirements on how the model will be used must be compatible. That careful design is required when using language models in general, perhaps followed-up by spot testing with sequence generation to confirm model requirements have been met.

19.7 Further Reading

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

- Jack and Jill on Wikipedia.
  https://en.wikipedia.org/wiki/Jack_and_Jill_(nursery_rhyme)
- Language Model on Wikpedia.
  https://en.wikipedia.org/wiki/Language_model
- Keras Embedding Layer API.
  https://keras.io/layers/embeddings/#embedding

- Keras Text Processing API.
  https://keras.io/preprocessing/text/
- Keras Sequence Processing API.
  https://keras.io/preprocessing/sequence/
- Keras Utils API.
  https://keras.io/utils/

19.8 Summary

In this tutorial, you discovered how to develop different word-based language models for a simple nursery rhyme. Specifically, you learned:

- The challenge of developing a good framing of a word-based language model for a given application.
- How to develop one-word, two-word, and line-based framings for word-based language models.
- How to generate sequences using a fit language model.

19.8.1 Next

In the next chapter, you will discover how you can develop a word-based neural language model on a large corpus of text.