How to Develop a Word Embedding Model for Predicting Movie Review Sentiment

by **Jason Brownlee** in **Deep Learning for Natural Language Processing**

Develop a Deep Learning Model to Automatically Classify Movie Reviews as Positive or Negative in Python with Keras, Step-by-Step.

Word embeddings are a technique for representing text where different words with similar meaning have a similar real-valued vector representation.

They are a key breakthrough that has led to great performance of neural network models on a suite of challenging natural language processing problems.

In this tutorial, you will discover how to develop word embedding models for neural networks to classify movie reviews.

After completing this tutorial, you will know:

- How to prepare movie review text data for classification with deep learning methods.
- How to learn a word embedding as part of fitting a deep learning model.
- How to learn a standalone word embedding and how to use a pre-trained embedding in a neural network model.

Let’s get started.

Note: This is an excerpt from: “Deep Learning for Natural Language Processing“. Take a look, if you want more step-by-step tutorials on getting the most out of deep learning methods when working with text data.

How to Develop a Word Embedding Model for Predicting Movie Review Sentiment

Photo by Katrina Br*?#*!@nd, some rights reserved.

Tutorial Overview

This tutorial is divided into 5 parts; they are:

1. Movie Review Dataset
2. Data Preparation
3. Train Embedding Layer
4. Train word2vec Embedding
5. Use Pre-trained Embedding

Python Environment

This tutorial assumes you have a Python SciPy environment installed, ideally with Python 3.

You must have Keras (2.2 or higher) installed with either the TensorFlow or Theano backend.

The tutorial also assumes you have scikit-learn, Pandas, NumPy, and Matplotlib installed.

If you need help with your environment, see this tutorial:

- How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda

A GPU is not required for this tutorial, nevertheless, you can access GPUs cheaply on Amazon Web Services. Learn how in this tutorial:

- How to Setup Amazon AWS EC2 GPUs to Train Keras Deep Learning Models (step-by-step)

Let's dive in.

1. Movie Review Dataset

The Movie Review Data is a collection of movie reviews retrieved from the imdb.com website in the early 2000s by Bo Pang and Lillian Lee. The reviews were collected and made available as part of their research on natural language processing.

The reviews were originally released in 2002, but an updated and cleaned up version were released in 2004, referred to as "v2.0".

The dataset is comprised of 1,000 positive and 1,000 negative movie reviews drawn from an archive of the rec.arts.movies.reviews newsgroup hosted at imdb.com. The authors refer to this dataset as the "polarity dataset."

Our data contains 1000 positive and 1000 negative reviews all written before 2002, with a cap of 20 reviews per author (312 authors total) per category. We refer to this corpus as the polarity dataset.

— A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts, 2004.

The data has been cleaned up somewhat, for example:

- The dataset is comprised of only English reviews.
- All text has been converted to lowercase.
- There is white space around punctuation like periods, commas, and brackets.
- Text has been split into one sentence per line.

The data has been used for a few related natural language processing tasks. For classification, the performance of machine learning models (such as Support Vector Machines) on the data is in the range of high 70% to low 80% (e.g. 78%-82%).

More sophisticated data preparation may see results as high as 86% with 10-fold cross validation. This gives us a ballpark of low-to-mid 80s if we were looking to use this dataset in experiments of modern methods.

… depending on choice of downstream polarity classifier, we can achieve highly statistically significant improvement (from 82.8% to 86.4%)

— A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts, 2004.
You can download the dataset from here:

- Movie Review Polarity Dataset (review_polarity.tar.gz, 3MB)
After unzipping the file, you will have a directory called "*txt_sentoken*" with two sub-directories containing the text "*neg*" and "*pos*" for negative and positive reviews. Reviews are stored one per file with a naming convention cv000 to cv999 for each neg and pos.
Next, let's look at loading and preparing the text data.

2. Data Preparation

In this section, we will look at 3 things:

1. Separation of data into training and test sets.
2. Loading and cleaning the data to remove punctuation and numbers.
3. Defining a vocabulary of preferred words.

Split into Train and Test Sets

We are pretending that we are developing a system that can predict the sentiment of a textual movie review as either positive or negative.

This means that after the model is developed, we will need to make predictions on new textual reviews. This will require all of the same data preparation to be performed on those new reviews as is performed on the training data for the model.

We will ensure that this constraint is built into the evaluation of our models by splitting the training and test datasets prior to any data preparation. This means that any knowledge in the data in the test set that could help us better prepare the data (e.g. the words used) are unavailable in the preparation of data used for training the model.

That being said, we will use the last 100 positive reviews and the last 100 negative reviews as a test set (100 reviews) and the remaining 1,800 reviews as the training dataset.

This is a 90% train, 10% split of the data.

The split can be imposed easily by using the filenames of the reviews where reviews named 000 to 899 are for training data and reviews named 900 onwards are for test.

Loading and Cleaning Reviews

The text data is already pretty clean; not much preparation is required.

If you are new to cleaning text data, see this post:

- How to Clean Text for Machine Learning with Python

Without getting bogged down too much in the details, we will prepare the data using the following way:

- Split tokens on white space.
- Remove all punctuation from words.
- Remove all words that are not purely comprised of alphabetical characters.
- Remove all words that are known stop words.
- Remove all words that have a length <= 1 character.

We can put all of these steps into a function called *clean_doc()* that takes as an argument the raw text loaded from a file and returns a list of cleaned tokens. We can also define a function *load_doc()* that loads a document from file ready for use with the *clean_doc()* function.
An example of cleaning the first positive review is listed below.

```
from nltk.corpus import stopwords
import string

# 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):
    # split into tokens by white space
    tokens = doc.split()
    # remove punctuation from each token
    table = str.maketrans('', '', string.punctuation)
    tokens = [w.translate(table) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load the document
filename = 'txt_sentoken/pos/cv000_29590.txt'
text = load_doc(filename)
tokens = clean_doc(text)
print(tokens)
```

Running the example prints a long list of clean tokens.

There are many more cleaning steps we may want to explore and I leave them as further exercises.

I’d love to see what you can come up with.
Post your approaches and findings in the comments at the end.

```
...
'creepy', 'place', 'even', 'acting', 'hell', 'solid', 'dreamy', 'depp', 'turning', 'typically', 'strong', 'performance', 'deftly', 'handling', 'british', 'accent', 'ians', 'holm', 'joe', 'goulds', 'secret', 'richardson', 'dalmatians', 'log', 'great', 'supporting', 'roles', 'big', 'surprise', 'graham', 'cringed', 'first', 'time', 'opened', 'mouth', 'imagining', 'attempt', 'irish', 'accent', 'actually', 'wasnt', 'half', 'bad', 'film', 'however', 'good', 'strong', 'violencegore', 'sexuality', 'language', 'drug', 'content']
```

Define a Vocabulary

It is important to define a vocabulary of known words when using a bag-of-words or embedding model.

The more words, the larger the representation of documents, therefore it is important to constrain the words to only those believed to be predictive. This is difficult to know beforehand and often it is important to test different hypotheses about how to construct a useful vocabulary.

We have already seen how we can remove punctuation and numbers from the vocabulary in the previous section. We can repeat this for all documents and build a set of all known words.

We can develop a vocabulary as a Counter, which is a dictionary mapping of words and their counts that allow us to easily update and query.

Each document can be added to the counter (a new function called *add_doc_to_vocab()*) and we can step over all of the reviews in the negative directory and then the positive directory (a new function called *process_docs()*).
The complete example is listed below.

```
from string import punctuation
from os import listdir
from collections import Counter
from nltk.corpus import stopwords

# 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):
	# split into tokens by white space
	tokens = doc.split()
	# remove punctuation from each token
	table = str.maketrans('', '', punctuation)
	tokens = [w.translate(table) for w in tokens]
	# remove remaining tokens that are not alphabetic
	tokens = [word for word in tokens if word.isalpha()]
	# filter out stop words
	stop_words = set(stopwords.words('english'))
	tokens = [w for w in tokens if not w in stop_words]
	# filter out short tokens
	tokens = [word for word in tokens if len(word) > 1]
	return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
	# load doc
	doc = load_doc(filename)
	# clean doc
	tokens = clean_doc(doc)
	# update counts
	vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab, is_trian):
	# walk through all files in the folder
	for filename in listdir(directory):
		# skip any reviews in the test set
		if is_trian and filename.startswith('cv9'):
			continue
		if not is_trian and not filename.startswith('cv9'):
			continue
			# create the full path of the file to open
			path = directory + '/' + filename
			# add doc to vocab
			add_doc_to_vocab(path, vocab)

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/neg', vocab, True)
process_docs('txt_sentoken/pos', vocab, True)
# print the size of the vocab
print(len(vocab))
# print the top words in the vocab
print(vocab.most_common(50))
```

Running the example shows that we have a vocabulary of 44,276 words.

We also can see a sample of the top 50 most used words in the movie reviews.

Note, that this vocabulary was constructed based on only those reviews in the training dataset.

```
44276
[('film', 7983), ('one', 4946), ('movie', 4826), ('like', 3201), ('even', 2262), ('good', 2080), ('time', 2041), ('story', 1907), ('films', 1873), ('would', 1844), ('much', 1824), ('also', 1757), ('characters', 1735), ('get', 1724), ('character', 1703), ('two', 1643), ('first', 1588), ('see', 1557), ('way', 1515), ('well', 1511), ('make', 1418), ('really', 1407), ('little', 1351), ('life', 1334), ('plot', 1288), ('people', 1269), ('could', 1248), ('bad', 1248), ('scene', 1241), ('movies', 1238), ('never', 1201), ('best', 1179), ('new', 1140), ('scenes', 1135), ('man', 1131), ('many', 1130), ('doesnt', 1118), ('know', 1092), ('dont', 1086), ('hes', 1024), ('great', 1014), ('another', 992), ('action', 985), ('love', 977), ('us', 967), ('go', 952), ('director', 948), ('end', 946), ('something', 945), ('still', 936)]
```

We can step through the vocabulary and remove all words that have a low occurrence, such as only being used once or twice in all reviews.

For example, the following snippet will retrieve only the tokens that of appears 2 or more times in all reviews.

```
# keep tokens with a min occurrence
min_occurane = 2
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
```

Running the above example with this addition shows that the vocabulary size drops by a little more than half its size from 44,276 to 25,767 words.

```
25767
```

Finally, the vocabulary can be saved to a new file called *vocab.txt* that we can later load and use to filter movie reviews prior to encoding them for modeling. We define a new function called *save_list()* that saves the vocabulary to file, with one word per file.
For example:

```
# save list to file
def save_list(lines, filename):
	# convert lines to a single blob of text
	data = '\n'.join(lines)
	# open file
	file = open(filename, 'w')
	# write text
	file.write(data)
	# close file
	file.close()

# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')
```

Running the min occurrence filter on the vocabulary and saving it to file, you should now have a new file called *vocab.txt* with only the words we are interested in.
The order of words in your file will differ, but should look something like the following:

```
aberdeen
dupe
burt
libido
hamlet
arlene
available
corners
web
columbia
...
```

We are now ready to look at learning features from the reviews.

3. Train Embedding Layer

In this section, we will learn a word embedding while training a neural network on the classification problem.

A word embedding is a way of representing text where each word in the vocabulary is represented by a real valued vector in a high-dimensional space. The vectors are learned in such a way that words that have similar meanings will have similar representation in the vector space (close in the vector space). This is a more expressive representation for text than more classical methods like bag-of-words, where relationships between words or tokens are ignored, or forced in bigram and trigram approaches.

The real valued vector representation for words can be learned while training the neural network. We can do this in the Keras deep learning library using the Embedding layer.
If you are new to word embeddings, see the post:

- What Are Word Embeddings for Text?

If you are new to word embedding layers in Keras, see the post:

- How to Use Word Embedding Layers for Deep Learning with Keras

The first step is to load the vocabulary. We will use it to filter out words from movie reviews that we are not interested in.

If you have worked through the previous section, you should have a local file called '*vocab.txt*' with one word per line. We can load that file and build a vocabulary as a set for checking the validity of tokens.

```
# 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 the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)
```

Next, we need to load all of the training data movie reviews. For that we can adapt the *process_docs()* from the previous section to load the documents, clean them, and return them as a list of strings, with one document per string. We want each document to be a string for easy encoding as a sequence of integers later.
Cleaning the document involves splitting each review based on white space, removing punctuation, and then filtering out all tokens not in the vocabulary.

The updated *clean_doc()* function is listed below.

```
# turn a doc into clean tokens
def clean_doc(doc, vocab):
	# split into tokens by white space
	tokens = doc.split()
	# remove punctuation from each token
	table = str.maketrans('', '', punctuation)
	tokens = [w.translate(table) for w in tokens]
	# filter out tokens not in vocab
	tokens = [w for w in tokens if w in vocab]
	tokens = ' '.join(tokens)
	return tokens
```

The updated *process_docs()* can then call the *clean_doc()* for each document on the '*pos*' and '*neg*' directories that are in our training dataset.

```
# load all docs in a directory
def process_docs(directory, vocab, is_trian):
	documents = list()
	# walk through all files in the folder
	for filename in listdir(directory):
		# skip any reviews in the test set
		if is_trian and filename.startswith('cv9'):
```

```
            continue
        if not is_trian and not filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # load the doc
        doc = load_doc(path)
        # clean doc
        tokens = clean_doc(doc, vocab)
        # add to list
        documents.append(tokens)
    return documents

# load all training reviews
positive_docs = process_docs('txt_sentoken/pos', vocab, True)
negative_docs = process_docs('txt_sentoken/neg', vocab, True)
train_docs = negative_docs + positive_docs
```

The next step is to encode each document as a sequence of integers.

The Keras Embedding layer requires integer inputs where each integer maps to a single token that has a specific real-valued vector representation within the embedding. These vectors are random at the beginning of training, but during training become meaningful to the network.

We can encode the training documents as sequences of integers using the Tokenizer class in the Keras API.
First, we must construct an instance of the class then train it on all documents in the training dataset. In this case, it develops a vocabulary of all tokens in the training dataset and develops a consistent mapping from words in the vocabulary to unique integers. We could just as easily develop this mapping ourselves using our vocabulary file.

```
# create the tokenizer
tokenizer = Tokenizer()
# fit the tokenizer on the documents
tokenizer.fit_on_texts(train_docs)
```

Now that the mapping of words to integers has been prepared, we can use it to encode the reviews in the training dataset. We can do that by calling the *texts_to_sequences()* function on the Tokenizer.

```
# sequence encode
encoded_docs = tokenizer.texts_to_sequences(train_docs)
```

We also need to ensure that all documents have the same length.

This is a requirement of Keras for efficient computation. We could truncate reviews to the smallest size or zero-pad (pad with the value ‘0’) reviews to the maximum length, or some hybrid. In this case, we will pad all reviews to the length of the longest review in the training dataset.

First, we can find the longest review using the *max()* function on the training dataset and take its length. We can then call the Keras function *pad_sequences()* to pad the sequences to the maximum length by adding 0 values on the end.

```
# pad sequences
max_length = max([len(s.split()) for s in train_docs])
Xtrain = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
```

Finally, we can define the class labels for the training dataset, needed to fit the supervised neural network model to predict the sentiment of reviews.

```
# define training labels
ytrain = array([0 for _ in range(900)] + [1 for _ in range(900)])
```

We can then encode and pad the test dataset, needed later to evaluate the model after we train it.

```
# load all test reviews
positive_docs = process_docs('txt_sentoken/pos', vocab, False)
negative_docs = process_docs('txt_sentoken/neg', vocab, False)
test_docs = negative_docs + positive_docs
# sequence encode
encoded_docs = tokenizer.texts_to_sequences(test_docs)
# pad sequences
Xtest = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
# define test labels
ytest = array([0 for _ in range(100)] + [1 for _ in range(100)])
```

We are now ready to define our neural network model.

The model will use an Embedding layer as the first hidden layer. The Embedding requires the specification of the vocabulary size, the size of the real-valued vector space, and the maximum length of input documents.

The vocabulary size is the total number of words in our vocabulary, plus one for unknown words. This could be the vocab set length or the size of the vocab within the tokenizer used to integer encode the documents, for example:

```
# define vocabulary size (largest integer value)
vocab_size = len(tokenizer.word_index) + 1
```

We will use a 100-dimensional vector space, but you could try other values, such as 50 or 150. Finally, the maximum document length was calculated above in the *max_length* variable used during padding.
The complete model definition is listed below including the Embedding layer.

We use a Convolutional Neural Network (CNN) as they have proven to be successful at document classification problems. A conservative CNN configuration is used with 32 filters (parallel fields for processing words) and a kernel size of 8 with a rectified linear ('relu') activation function. This is followed by a pooling layer that reduces the output of the convolutional layer by half.

Next, the 2D output from the CNN part of the model is flattened to one long 2D vector to represent the 'features' extracted by the CNN. The back-end of the model is a standard Multilayer Perceptron layers to interpret the CNN features. The output layer uses a sigmoid activation function to output a value between 0 and 1 for the negative and positive sentiment in the review.

For more advice on effective deep learning model configuration for text classification, see the post:

Best Practices for Document Classification with Deep Learning

```
# define model
model = Sequential()
model.add(Embedding(vocab_size, 100, input_length=max_length))
model.add(Conv1D(filters=32, kernel_size=8, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
print(model.summary())
```

Running just this piece provides a summary of the defined network.

We can see that the Embedding layer expects documents with a length of 442 words as input and encodes each word in the document as a 100 element vector.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 442, 100)          2576800
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 435, 32)           25632
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 217, 32)           0
_________________________________________________________________
flatten_1 (Flatten)          (None, 6944)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                69450
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 11
=================================================================
Total params: 2,671,893
Trainable params: 2,671,893
Non-trainable params: 0
_________________________________________________________________
```

Next, we fit the network on the training data.

We use a binary cross entropy loss function because the problem we are learning is a binary classification problem. The efficient Adam implementation of stochastic gradient descent is used and we keep track of accuracy in addition to loss during training. The model is trained for 10 epochs, or 10 passes through the training data.

The network configuration and training schedule were found with a little trial and error, but are by no means optimal for this problem. If you can get better results with a different configuration, let me know.

```
# compile network
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
```

After the model is fit, it is evaluated on the test dataset. This dataset contains words that we have not seen before and reviews not seen during training.

```
# evaluate
loss, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %f' % (acc*100))
```

We can tie all of this together.

The complete code listing is provided below.

```
from string import punctuation
from os import listdir
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# 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, vocab):
	# split into tokens by white space
	tokens = doc.split()
	# remove punctuation from each token
	table = str.maketrans('', '', punctuation)
	tokens = [w.translate(table) for w in tokens]
	# filter out tokens not in vocab
	tokens = [w for w in tokens if w in vocab]
	tokens = ' '.join(tokens)
	return tokens

# load all docs in a directory
```

```
def process_docs(directory, vocab, is_trian):
	documents = list()
	# walk through all files in the folder
	for filename in listdir(directory):
		# skip any reviews in the test set
		if is_trian and filename.startswith('cv9'):
			continue
		if not is_trian and not filename.startswith('cv9'):
			continue
		# create the full path of the file to open
		path = directory + '/' + filename
		# load the doc
		doc = load_doc(path)
		# clean doc
		tokens = clean_doc(doc, vocab)
		# add to list
		documents.append(tokens)
	return documents

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)

# load all training reviews
positive_docs = process_docs('txt_sentoken/pos', vocab, True)
negative_docs = process_docs('txt_sentoken/neg', vocab, True)
train_docs = negative_docs + positive_docs

# create the tokenizer
tokenizer = Tokenizer()
# fit the tokenizer on the documents
tokenizer.fit_on_texts(train_docs)

# sequence encode
encoded_docs = tokenizer.texts_to_sequences(train_docs)
# pad sequences
max_length = max([len(s.split()) for s in train_docs])
Xtrain = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
# define training labels
ytrain = array([0 for _ in range(900)] + [1 for _ in range(900)])

# load all test reviews
positive_docs = process_docs('txt_sentoken/pos', vocab, False)
negative_docs = process_docs('txt_sentoken/neg', vocab, False)
test_docs = negative_docs + positive_docs
# sequence encode
encoded_docs = tokenizer.texts_to_sequences(test_docs)
# pad sequences
Xtest = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
# define test labels
ytest = array([0 for _ in range(100)] + [1 for _ in range(100)])

# define vocabulary size (largest integer value)
vocab_size = len(tokenizer.word_index) + 1

# define model
model = Sequential()
model.add(Embedding(vocab_size, 100, input_length=max_length))
model.add(Conv1D(filters=32, kernel_size=8, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
print(model.summary())
# compile network
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
# evaluate
loss, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %f' % (acc*100))
```

Running the example prints the loss and accuracy at the end of each training epoch. We can see that the model very quickly achieves 100% accuracy on the training dataset.

At the end of the run, the model achieves an accuracy of 84.5% on the test dataset, which is a great score.

Given the stochastic nature of neural networks, your specific results will vary. Consider running the example a few times and taking the average score as the skill of the model.

```
...
Epoch 6/10
2s - loss: 0.0013 - acc: 1.0000
Epoch 7/10
2s - loss: 8.4573e-04 - acc: 1.0000
Epoch 8/10
2s - loss: 5.8323e-04 - acc: 1.0000
Epoch 9/10
2s - loss: 4.3155e-04 - acc: 1.0000
Epoch 10/10
2s - loss: 3.3083e-04 - acc: 1.0000
Test Accuracy: 84.500000
```

We have just seen an example of how we can learn a word embedding as part of fitting a neural network model.

Next, let’s look at how we can efficiently learn a standalone embedding that we could later use in our neural network.

4. Train word2vec Embedding

In this section, we will discover how to learn a standalone word embedding using an efficient algorithm called word2vec.

A downside of learning a word embedding as part of the network is that it can be very slow, especially for very large text datasets.

The word2vec algorithm is an approach to learning a word embedding from a text corpus in a standalone way. The benefit of the method is that it can produce high-quality word embeddings very efficiently, in terms of space and time complexity.

The first step is to prepare the documents ready for learning the embedding.

This involves the same data cleaning steps from the previous section, namely splitting documents by their white space, removing punctuation, and filtering out tokens not in the vocabulary.

The word2vec algorithm processes documents sentence by sentence. This means we will preserve the sentence-based structure during cleaning.

We start by loading the vocabulary, as before.

```
# 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 the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)
```

Next, we define a function named *doc_to_clean_lines()* to clean a loaded document line by line and return a list of the cleaned lines.

```
# turn a doc into clean tokens
def doc_to_clean_lines(doc, vocab):
	clean_lines = list()
	lines = doc.splitlines()
	for line in lines:
		# split into tokens by white space
		tokens = line.split()
		# remove punctuation from each token
		table = str.maketrans('', '', punctuation)
		tokens = [w.translate(table) for w in tokens]
		# filter out tokens not in vocab
		tokens = [w for w in tokens if w in vocab]
		clean_lines.append(tokens)
	return clean_lines
```

Next, we adapt the process_docs() function to load and clean all of the documents in a folder and return a list of all document lines.

The results from this function will be the training data for the word2vec model.

```
# load all docs in a directory
def process_docs(directory, vocab, is_trian):
	lines = list()
	# walk through all files in the folder
	for filename in listdir(directory):
		# skip any reviews in the test set
		if is_trian and filename.startswith('cv9'):
			continue
		if not is_trian and not filename.startswith('cv9'):
			continue
		# create the full path of the file to open
		path = directory + '/' + filename
		# load and clean the doc
		doc = load_doc(path)
		doc_lines = doc_to_clean_lines(doc, vocab)
		# add lines to list
		lines += doc_lines
	return lines
```

We can then load all of the training data and convert it into a long list of 'sentences' (lists of tokens) ready for fitting the word2vec model.

```
# load training data
positive_lines = process_docs('txt_sentoken/pos', vocab, True)
negative_lines = process_docs('txt_sentoken/neg', vocab, True)
sentences = negative_docs + positive_docs
print('Total training sentences: %d' % len(sentences))
```

We will use the word2vec implementation provided in the Gensim Python library.

Specifically the Word2Vec class.

For more on training a standalone word embedding with Gensim, see the post:

- How to Develop Word Embeddings in Python with Gensim

The model is fit when constructing the class. We pass in the list of clean sentences from the training data, then specify the size of the embedding vector space (we use 100 again), the number of neighboring words to look at when learning how to embed each word in the training sentences (we use 5 neighbors), the number of threads to use when fitting the model (we use 8, but change this if you have more or less CPU cores), and the minimum occurrence count for words to consider in the vocabulary (we set this to 1 as we have already prepared the vocabulary).

After the model is fit, we print the size of the learned vocabulary, which should match the size of our vocabulary in vocab.txt of 25,767 tokens.

```
# train word2vec model
model = Word2Vec(sentences, size=100, window=5, workers=8, min_count=1)
# summarize vocabulary size in model
words = list(model.wv.vocab)
print('Vocabulary size: %d' % len(words))
```

Finally, we save the learned embedding vectors to file using the save_word2vec_format() on the model's '*wv*' (word vector) attribute. The embedding is saved in ASCII format with one word and vector per line.
The complete example is listed below.

```
from string import punctuation
from os import listdir
from gensim.models import Word2Vec

# 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 doc_to_clean_lines(doc, vocab):
	clean_lines = list()
	lines = doc.splitlines()
	for line in lines:
		# split into tokens by white space
		tokens = line.split()
		# remove punctuation from each token
		table = str.maketrans('', '', punctuation)
		tokens = [w.translate(table) for w in tokens]
		# filter out tokens not in vocab
		tokens = [w for w in tokens if w in vocab]
		clean_lines.append(tokens)
	return clean_lines

# load all docs in a directory
def process_docs(directory, vocab, is_trian):
	lines = list()
	# walk through all files in the folder
	for filename in listdir(directory):
		# skip any reviews in the test set
		if is_trian and filename.startswith('cv9'):
			continue
		if not is_trian and not filename.startswith('cv9'):
			continue
		# create the full path of the file to open
		path = directory + '/' + filename
		# load and clean the doc
		doc = load_doc(path)
		doc_lines = doc_to_clean_lines(doc, vocab)
		# add lines to list
		lines += doc_lines
	return lines

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)
```

```

# load training data
positive_lines = process_docs('txt_sentoken/pos', vocab, True)
negative_lines = process_docs('txt_sentoken/neg', vocab, True)
sentences = negative_docs + positive_docs
print('Total training sentences: %d' % len(sentences))

# train word2vec model
model = Word2Vec(sentences, size=100, window=5, workers=8, min_count=1)
# summarize vocabulary size in model
words = list(model.wv.vocab)
print('Vocabulary size: %d' % len(words))

# save model in ASCII (word2vec) format
filename = 'embedding_word2vec.txt'
model.wv.save_word2vec_format(filename, binary=False)
```

Running the example loads 58,109 sentences from the training data and creates an embedding for a vocabulary of 25,767 words.

You should now have a file 'embedding_word2vec.txt' with the learned vectors in your current working directory.

```
Total training sentences: 58109
Vocabulary size: 25767
```

Next, let's look at using these learned vectors in our model.

5. Use Pre-trained Embedding

In this section, we will use a pre-trained word embedding prepared on a very large text corpus.

We can use the pre-trained word embedding developed in the previous section and the CNN model developed in the section before that.

The first step is to load the word embedding as a directory of words to vectors. The word embedding was saved in so-called '*word2vec*' format that contains a header line. We will skip this header line when loading the embedding.
The function below named *load_embedding()* loads the embedding and returns a directory of words mapped to the vectors in NumPy format.

```
# load embedding as a dict
def load_embedding(filename):
	# load embedding into memory, skip first line
	file = open(filename,'r')
	lines = file.readlines()[1:]
	file.close()
	# create a map of words to vectors
	embedding = dict()
	for line in lines:
		parts = line.split()
```

```
			# key is string word, value is numpy array for vector
			embedding[parts[0]] = asarray(parts[1:], dtype='float32')
		return embedding
```

Now that we have all of the vectors in memory, we can order them in such a way as to match the integer encoding prepared by the Keras Tokenizer.

Recall that we integer encode the review documents prior to passing them to the Embedding layer. The integer maps to the index of a specific vector in the embedding layer. Therefore, it is important that we lay the vectors out in the Embedding layer such that the encoded words map to the correct vector.

Below defines a function *get_weight_matrix()* that takes the loaded embedding and the tokenizer.word_index vocabulary as arguments and returns a matrix with the word vectors in the correct locations.

```
# create a weight matrix for the Embedding layer from a loaded embedding
def get_weight_matrix(embedding, vocab):
		# total vocabulary size plus 0 for unknown words
		vocab_size = len(vocab) + 1
		# define weight matrix dimensions with all 0
		weight_matrix = zeros((vocab_size, 100))
		# step vocab, store vectors using the Tokenizer's integer mapping
		for word, i in vocab.items():
			weight_matrix[i] = embedding.get(word)
		return weight_matrix
```

Now we can use these functions to create our new Embedding layer for our model.

```
...
# load embedding from file
raw_embedding = load_embedding('embedding_word2vec.txt')
# get vectors in the right order
embedding_vectors = get_weight_matrix(raw_embedding, tokenizer.word_index)
# create the embedding layer
embedding_layer = Embedding(vocab_size, 100, weights=[embedding_vectors], input_length=max_length,
trainable=False)
```

Note that the prepared weight matrix *embedding_vectors* is passed to the new Embedding layer as an argument and that we set the '*trainable*' argument to '*False*' to ensure that the network does not try to adapt the pre-learned vectors as part of training the network.
We can now add this layer to our model. We also have a slightly different model configuration with a lot more filters (128) in the CNN model and a kernel that matches the 5 words used as neighbors when developing the word2vec embedding. Finally, the back-end of the model was simplified.

```
# define model
model = Sequential()
model.add(embedding_layer)
model.add(Conv1D(filters=128, kernel_size=5, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
print(model.summary())
```

These changes were found with a little trial and error.

The complete code listing is provided below.

```
from string import punctuation
from os import listdir
from numpy import array
from numpy import asarray
from numpy import zeros
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# 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, vocab):
	# split into tokens by white space
	tokens = doc.split()
	# remove punctuation from each token
	table = str.maketrans('', '', punctuation)
	tokens = [w.translate(table) for w in tokens]
	# filter out tokens not in vocab
	tokens = [w for w in tokens if w in vocab]
	tokens = ' '.join(tokens)
	return tokens

# load all docs in a directory
def process_docs(directory, vocab, is_trian):
	documents = list()
	# walk through all files in the folder
	for filename in listdir(directory):
		# skip any reviews in the test set
		if is_trian and filename.startswith('cv9'):
			continue
		if not is_trian and not filename.startswith('cv9'):
			continue
		# create the full path of the file to open
		path = directory + '/' + filename
		# load the doc
		doc = load_doc(path)
		# clean doc
		tokens = clean_doc(doc, vocab)
		# add to list
		documents.append(tokens)
	return documents
```

```

# load embedding as a dict
def load_embedding(filename):
	# load embedding into memory, skip first line
	file = open(filename,'r')
	lines = file.readlines()[1:]
	file.close()
	# create a map of words to vectors
	embedding = dict()
	for line in lines:
		parts = line.split()
		# key is string word, value is numpy array for vector
		embedding[parts[0]] = asarray(parts[1:], dtype='float32')
	return embedding

# create a weight matrix for the Embedding layer from a loaded embedding
def get_weight_matrix(embedding, vocab):
	# total vocabulary size plus 0 for unknown words
	vocab_size = len(vocab) + 1
	# define weight matrix dimensions with all 0
	weight_matrix = zeros((vocab_size, 100))
	# step vocab, store vectors using the Tokenizer's integer mapping
	for word, i in vocab.items():
		weight_matrix[i] = embedding.get(word)
	return weight_matrix

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)

# load all training reviews
positive_docs = process_docs('txt_sentoken/pos', vocab, True)
negative_docs = process_docs('txt_sentoken/neg', vocab, True)
train_docs = negative_docs + positive_docs

# create the tokenizer
tokenizer = Tokenizer()
# fit the tokenizer on the documents
tokenizer.fit_on_texts(train_docs)

# sequence encode
encoded_docs = tokenizer.texts_to_sequences(train_docs)
# pad sequences
max_length = max([len(s.split()) for s in train_docs])
Xtrain = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
# define training labels
ytrain = array([0 for _ in range(900)] + [1 for _ in range(900)])

# load all test reviews
positive_docs = process_docs('txt_sentoken/pos', vocab, False)
negative_docs = process_docs('txt_sentoken/neg', vocab, False)
test_docs = negative_docs + positive_docs
# sequence encode
encoded_docs = tokenizer.texts_to_sequences(test_docs)
# pad sequences
Xtest = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
# define test labels
ytest = array([0 for _ in range(100)] + [1 for _ in range(100)])
```

```

# define vocabulary size (largest integer value)
vocab_size = len(tokenizer.word_index) + 1

# load embedding from file
raw_embedding = load_embedding('embedding_word2vec.txt')
# get vectors in the right order
embedding_vectors = get_weight_matrix(raw_embedding, tokenizer.word_index)
# create the embedding layer
embedding_layer = Embedding(vocab_size, 100, weights=[embedding_vectors], input_length=max_length,
trainable=False)

# define model
model = Sequential()
model.add(embedding_layer)
model.add(Conv1D(filters=128, kernel_size=5, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
print(model.summary())
# compile network
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
# evaluate
loss, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %f' % (acc*100))
```

Running the example shows that performance was not improved.

In fact, performance was a lot worse. The results show that the training dataset was learned successfully, but evaluation on the test dataset was very poor, at just above 50% accuracy.

The cause of the poor test performance may be because of the chosen word2vec configuration or the chosen neural network configuration.

```
...
Epoch 6/10
2s - loss: 0.3306 - acc: 0.8778
Epoch 7/10
2s - loss: 0.2888 - acc: 0.8917
Epoch 8/10
2s - loss: 0.1878 - acc: 0.9439
Epoch 9/10
2s - loss: 0.1255 - acc: 0.9750
Epoch 10/10
2s - loss: 0.0812 - acc: 0.9928
Test Accuracy: 53.000000
```

The weights in the embedding layer can be used as a starting point for the network, and adapted during the training of the network. We can do this by setting '*trainable=True*' (the default) in the creation of the embedding layer.

Repeating the experiment with this change shows slightly better results, but still poor.

I would encourage you to explore alternate configurations of the embedding and network to see if you can do better. Let me know how you do.

```
...
Epoch 6/10
4s - loss: 0.0950 - acc: 0.9917
Epoch 7/10
4s - loss: 0.0355 - acc: 0.9983
Epoch 8/10
4s - loss: 0.0158 - acc: 1.0000
Epoch 9/10
4s - loss: 0.0080 - acc: 1.0000
Epoch 10/10
4s - loss: 0.0050 - acc: 1.0000
Test Accuracy: 57.500000
```

It is possible to use pre-trained word vectors prepared on very large corpora of text data.

For example, both Google and Stanford provide pre-trained word vectors that you can download, trained with the efficient word2vec and GloVe methods respectively.

Let's try to use pre-trained vectors in our model.

You can download pre-trained GloVe vectors from the Stanford webpage. Specifically, vectors trained on Wikipedia data:

- glove.6B.zip (822 Megabyte download)

Unzipping the file, you will find pre-trained embeddings for various different dimensions. We will load the 100 dimension version in the file '*glove.6B.100d.txt*'
The Glove file does not contain a header file, so we do not need to skip the first line when loading the embedding into memory. The updated *load_embedding()* function is listed below.

```
# load embedding as a dict
def load_embedding(filename):
	# load embedding into memory, skip first line
	file = open(filename,'r')
	lines = file.readlines()
	file.close()
	# create a map of words to vectors
	embedding = dict()
	for line in lines:
		parts = line.split()
		# key is string word, value is numpy array for vector
		embedding[parts[0]] = asarray(parts[1:], dtype='float32')
	return embedding
```

It is possible that the loaded embedding does not contain all of the words in our chosen vocabulary. As such, when creating the Embedding weight matrix, we need to skip words that do not have a corresponding vector in the loaded GloVe data. Below is the updated, more defensive version of the *get_weight_matrix()* function.

```
# create a weight matrix for the Embedding layer from a loaded embedding
def get_weight_matrix(embedding, vocab):
	# total vocabulary size plus 0 for unknown words
	vocab_size = len(vocab) + 1
	# define weight matrix dimensions with all 0
	weight_matrix = zeros((vocab_size, 100))
	# step vocab, store vectors using the Tokenizer's integer mapping
	for word, i in vocab.items():
		vector = embedding.get(word)
		if vector is not None:
			weight_matrix[i] = vector
	return weight_matrix
```

We can now load the GloVe embedding and create the Embedding layer as before.

```
# load embedding from file
raw_embedding = load_embedding('glove.6B.100d.txt')
# get vectors in the right order
embedding_vectors = get_weight_matrix(raw_embedding, tokenizer.word_index)
# create the embedding layer
embedding_layer = Embedding(vocab_size, 100, weights=[embedding_vectors], input_length=max_length, trainable=False)
```

We will use the same model as before.

The complete example is listed below.

```
from string import punctuation
from os import listdir
from numpy import array
from numpy import asarray
from numpy import zeros
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# 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, vocab):
	# split into tokens by white space
	tokens = doc.split()
	# remove punctuation from each token
	table = str.maketrans('', '', punctuation)
	tokens = [w.translate(table) for w in tokens]
	# filter out tokens not in vocab
	tokens = [w for w in tokens if w in vocab]
	tokens = ' '.join(tokens)
	return tokens

```

```
# load all docs in a directory
def process_docs(directory, vocab, is_trian):
	documents = list()
	# walk through all files in the folder
	for filename in listdir(directory):
		# skip any reviews in the test set
		if is_trian and filename.startswith('cv9'):
			continue
		if not is_trian and not filename.startswith('cv9'):
			continue
		# create the full path of the file to open
		path = directory + '/' + filename
		# load the doc
		doc = load_doc(path)
		# clean doc
		tokens = clean_doc(doc, vocab)
		# add to list
		documents.append(tokens)
	return documents

# load embedding as a dict
def load_embedding(filename):
	# load embedding into memory, skip first line
	file = open(filename,'r')
	lines = file.readlines()
	file.close()
	# create a map of words to vectors
	embedding = dict()
	for line in lines:
		parts = line.split()
		# key is string word, value is numpy array for vector
		embedding[parts[0]] = asarray(parts[1:], dtype='float32')
	return embedding

# create a weight matrix for the Embedding layer from a loaded embedding
def get_weight_matrix(embedding, vocab):
	# total vocabulary size plus 0 for unknown words
	vocab_size = len(vocab) + 1
	# define weight matrix dimensions with all 0
	weight_matrix = zeros((vocab_size, 100))
	# step vocab, store vectors using the Tokenizer's integer mapping
	for word, i in vocab.items():
		vector = embedding.get(word)
		if vector is not None:
			weight_matrix[i] = vector
	return weight_matrix

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)

# load all training reviews
positive_docs = process_docs('txt_sentoken/pos', vocab, True)
negative_docs = process_docs('txt_sentoken/neg', vocab, True)
train_docs = negative_docs + positive_docs

# create the tokenizer
tokenizer = Tokenizer()
```

```
# fit the tokenizer on the documents
tokenizer.fit_on_texts(train_docs)

# sequence encode
encoded_docs = tokenizer.texts_to_sequences(train_docs)
# pad sequences
max_length = max([len(s.split()) for s in train_docs])
Xtrain = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
# define training labels
ytrain = array([0 for _ in range(900)] + [1 for _ in range(900)])

# load all test reviews
positive_docs = process_docs('txt_sentoken/pos', vocab, False)
negative_docs = process_docs('txt_sentoken/neg', vocab, False)
test_docs = negative_docs + positive_docs
# sequence encode
encoded_docs = tokenizer.texts_to_sequences(test_docs)
# pad sequences
Xtest = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
# define test labels
ytest = array([0 for _ in range(100)] + [1 for _ in range(100)])

# define vocabulary size (largest integer value)
vocab_size = len(tokenizer.word_index) + 1

# load embedding from file
raw_embedding = load_embedding('glove.6B.100d.txt')
# get vectors in the right order
embedding_vectors = get_weight_matrix(raw_embedding, tokenizer.word_index)
# create the embedding layer
embedding_layer = Embedding(vocab_size, 100, weights=[embedding_vectors], input_length=max_length,
trainable=False)

# define model
model = Sequential()
model.add(embedding_layer)
model.add(Conv1D(filters=128, kernel_size=5, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
print(model.summary())
# compile network
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
# evaluate
loss, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %f' % (acc*100))
```

Running the example shows better performance.

Again, the training dataset is easily learned and the model achieves 76% accuracy on the test dataset. This is good, but not as good as using a learned Embedding layer.

This may be cause of the higher quality vectors trained on more data and/or using a slightly different training process.

Your specific results may vary given the stochastic nature of neural networks. Try running the example a few times.

```
...
Epoch 6/10
2s - loss: 0.0278 - acc: 1.0000
Epoch 7/10
2s - loss: 0.0174 - acc: 1.0000
Epoch 8/10
2s - loss: 0.0117 - acc: 1.0000
Epoch 9/10
2s - loss: 0.0086 - acc: 1.0000
Epoch 10/10
2s - loss: 0.0068 - acc: 1.0000
Test Accuracy: 76.000000
```

In this case, it seems that learning the embedding as part of the learning task may be a better direction than using a specifically trained embedding or a more general pre-trained embedding.

Further Reading

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

Dataset

- Movie Review Data
- A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts, 2004.
- Movie Review Polarity Dataset (.tgz)
- Dataset Readme v2.0 and v1.1.

APIs

- collections API – Container datatypes
- Tokenizer Keras API
- Embedding Keras API
- Gensim Word2Vec API
- Gensim WordVector API

Embedding Methods

- word2vec on Google Code
- GloVe on Stanford

Related Posts

- Using pre-trained word embeddings in a Keras model, 2016.
- Implementing a CNN for Text Classification in TensorFlow, 2015.

Summary

In this tutorial, you discovered how to develop word embeddings for the classification of movie reviews.

Specifically, you learned:

- How to prepare movie review text data for classification with deep learning methods.
- How to learn a word embedding as part of fitting a deep learning model.
- How to learn a standalone word embedding and how to use a pre-trained embedding in a neural network model.

Do you have any questions?

Ask your questions in the comments below and I will do my best to answer.

Note: This post is an excerpt chapter from: "Deep Learning for Natural Language Processing"**.** Take a look, if you want more step-by-step tutorials on getting the most out of deep learning methods when working with text data.