Recurrent Neural Networks and Sentiment Analysis

Now, we will look at recurrent neural networks; a variation of the basic feed-forward neural network in PyTorch we learned to build in chapter 1. Generally, RNNs can be used for any task where data can be represented as a sequence. This includes things like stock price prediction using a time series of historic data, represented as a sequence. We commonly use RNNs in NLP as text can be thought of as a sequence of individual words and can be modelled as such. While a conventional neural network takes a single vector as an input into the model, an RNN can take a whole sequence of vectors. If we represent each word in a document as a vector embedding, we are able to represent our whole document as a sequence of vectors (or one order three tensor). We can then use RNNs (and a more sophisticated form of RNN known as a **Long Short Term Memory** (**LSTM**) to learn from our data.

In this chapter we will cover the basics of RNNs and the more advanced LSTM. We will then visit sentiment analysis and work through a practical example of how to build a LSTM to classify documents using PyTorch. Finally, we will host this simple model on Heroku, a simple cloud application platform, which will allow us to make predictions using our model.

This chapter covers the following topics:

* Building RNNs
* Working with LSTMs
* Building a Sentiment Analyzer using LSTMs
* Deploying the Application on Heroku

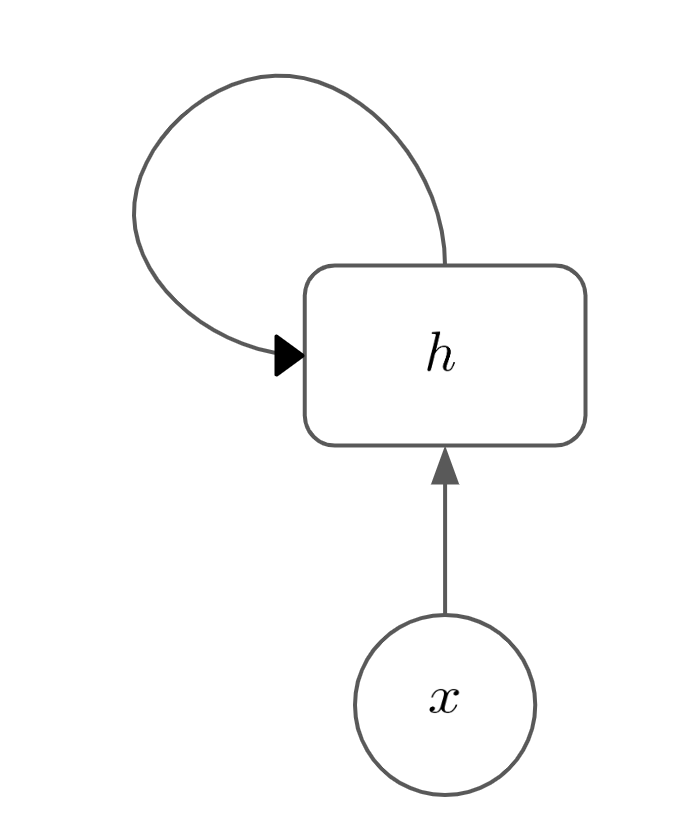
Technical Requirements

All code in this chapter can be found at <https://github.com/PacktPublishing/Hands-On-Natural-Language-Processing-with-PyTorch-1.x> . Heroku can be installed from [www.heroku.com](http://www.heroku.com) . The data was taken from <https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences>

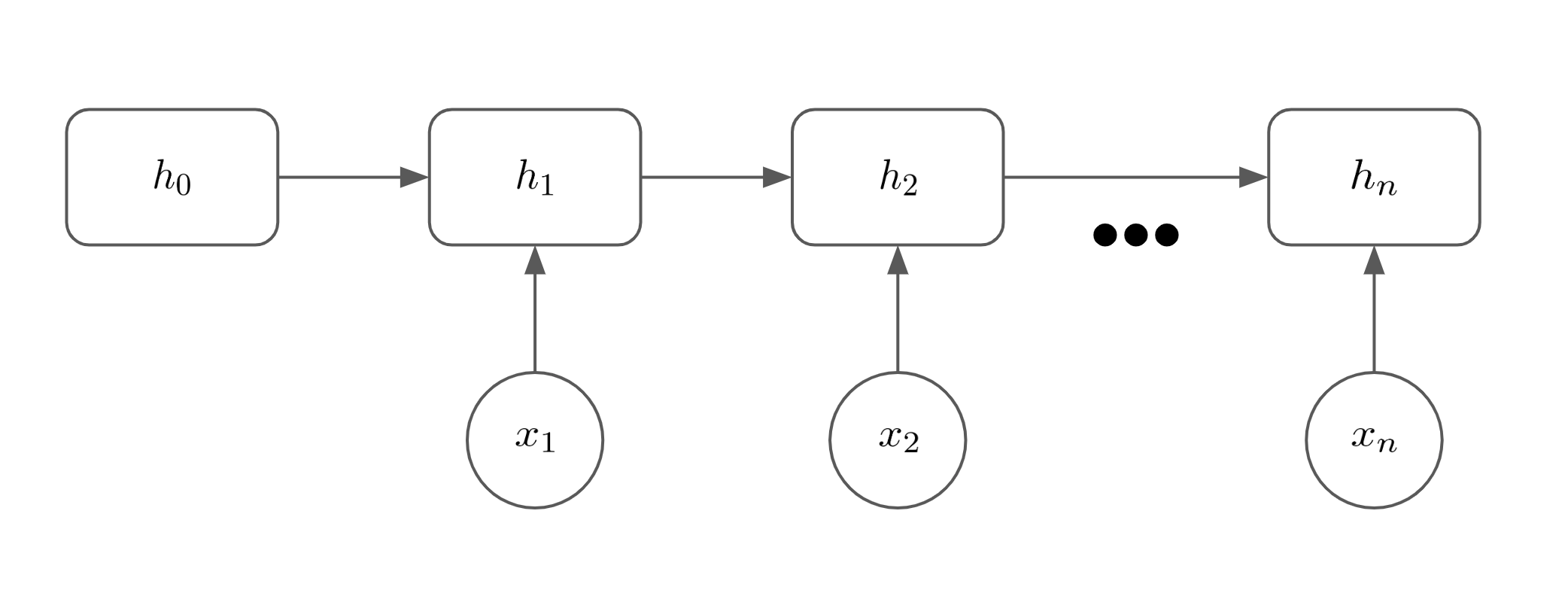
Building RNNs

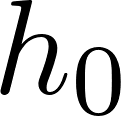
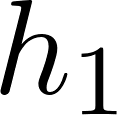
Recurrent neural networks consist of recurrent layers. While similar in many ways to the fully connected layers within a standard feed forward neural network, these recurrent layers consist of a hidden state which is updated at each step of the sequential input. This means that for any given sequence, the model is initialized with some hidden state, often represented as a one-dimensional vector. The first step of our sequence is then fed into our model and the hidden state is updated depending on some learned parameters. The second word is then fed into the network and the hidden state is updated again depending on some other learned parameters. These steps are repeated until the whole sequence has been processed and we are left with the final hidden state. This computation “loop”, with the hidden state being carried over from the previous computation and being updated is why we refer to these networks as being recurrent. This final hidden state is then connected to a further fully-connected layer and a final classification is predicted.

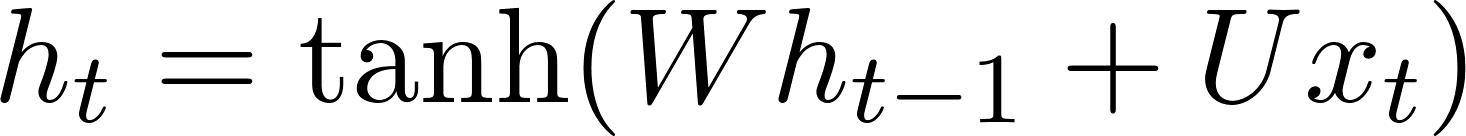
Our recurrent layer looks something like this, where [](https://www.codecogs.com/eqnedit.php?latex=h%250) is the hidden state and [](https://www.codecogs.com/eqnedit.php?latex=x%250) is our input at various time steps in our sequence. For each iteration, we update our hidden state at each time step [](https://www.codecogs.com/eqnedit.php?latex=x%250). The following image shows the layer:



Or, we can expand this out to the whole sequence of timesteps, which looks like this:



This layer is for an input [](https://www.codecogs.com/eqnedit.php?latex=n%250) timesteps long. Our hidden state is initialized in state [](https://www.codecogs.com/eqnedit.php?latex=h_%7B0%7D%250) and then uses our first input [](https://www.codecogs.com/eqnedit.php?latex=x_%7B1%7D%250) to compute the next hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7B1%7D%250). There are two sets of weight matrices that are also learned, matrix [](https://www.codecogs.com/eqnedit.php?latex=U%250) which learns how the hidden state changes between time steps, and matrix [](https://www.codecogs.com/eqnedit.php?latex=W%250) which learns how each input step affects the hidden state. We also apply a Tanh activation function to the resulting product, keeping the values of the hidden state between -1 and 1. The equation for calculating any hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7Bt%7D%250) becomes:

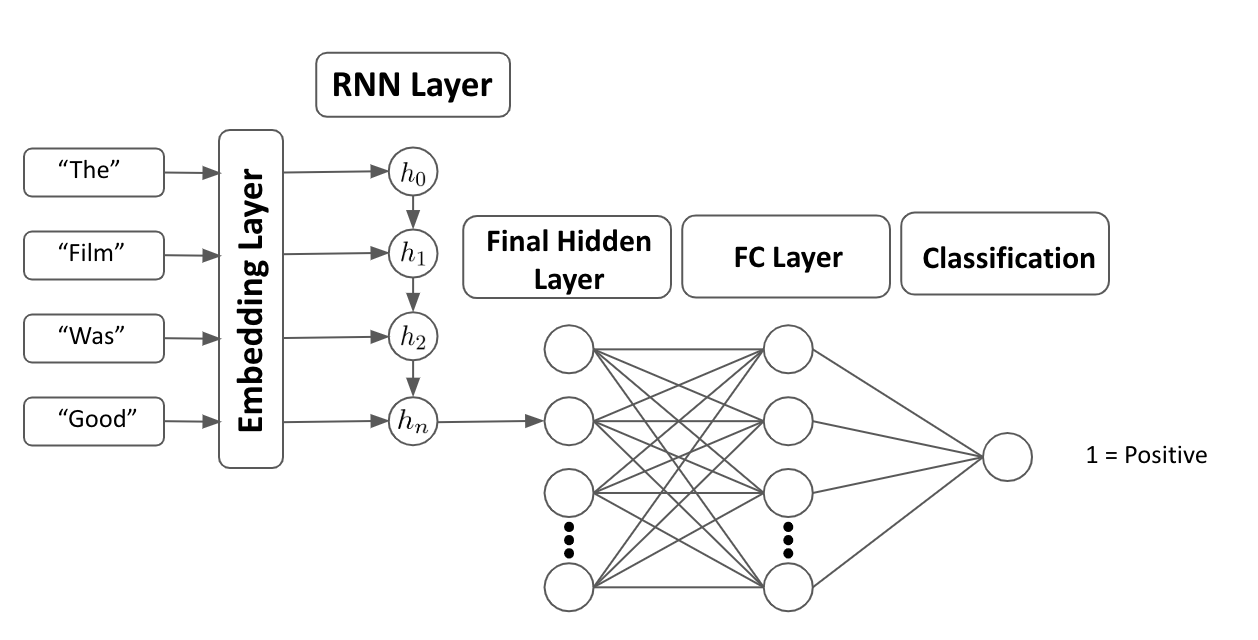
[](https://www.codecogs.com/eqnedit.php?latex=h_%7Bt%7D%20%3D%20%5Ctanh(Wh_%7Bt-1%7D%2BUx_%7Bt%7D)%250)

This is then repeated for each time step within our input sequence and the final output for this layer is our last hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7Bn%7D%250). When our network learns, we perform a forward pass through the network as before to compute our final classification. We then calculate a loss against this prediction and backpropagate through as the network before, calculating gradients as we go. The backpropagation process happens through all steps within the recurrent layer with the parameters between each input step and hidden state being learned.

We will see later that we can actually take the hidden state at each timestep rather than using the final hidden state which is useful for sequence-to-sequence translation tasks in NLP, however for the time being we will just take the hidden layer as output to the rest of the network.

Using RNNs for Sentiment Analysis

In the context of sentiment analysis, our model is trained on a sentiment analysis dataset of reviews consisting of a number of reviews in text and label of 0/1 depending on whether the review is negative or positive. This means that our model becomes a classification task (where the two classes are negative/positive). Our sentence is passed through a layer of learned word embeddings to form a representation of the sentence comprised of several vectors (one for each word). These vectors are then fed sequentially into our RNN layer and the final hidden state is passed through another fully-connected layer. Our model’s output is a single value between 0 and 1 depending on whether our model predicts negative or positive sentiment from the sentence. This means our complete classification model looks like this:

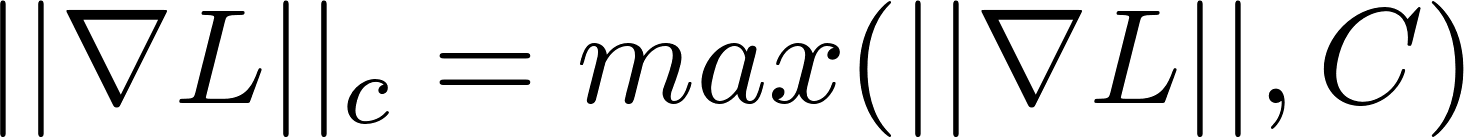


Now, we will highlight one of the issues with RNNs, gradient clipping:

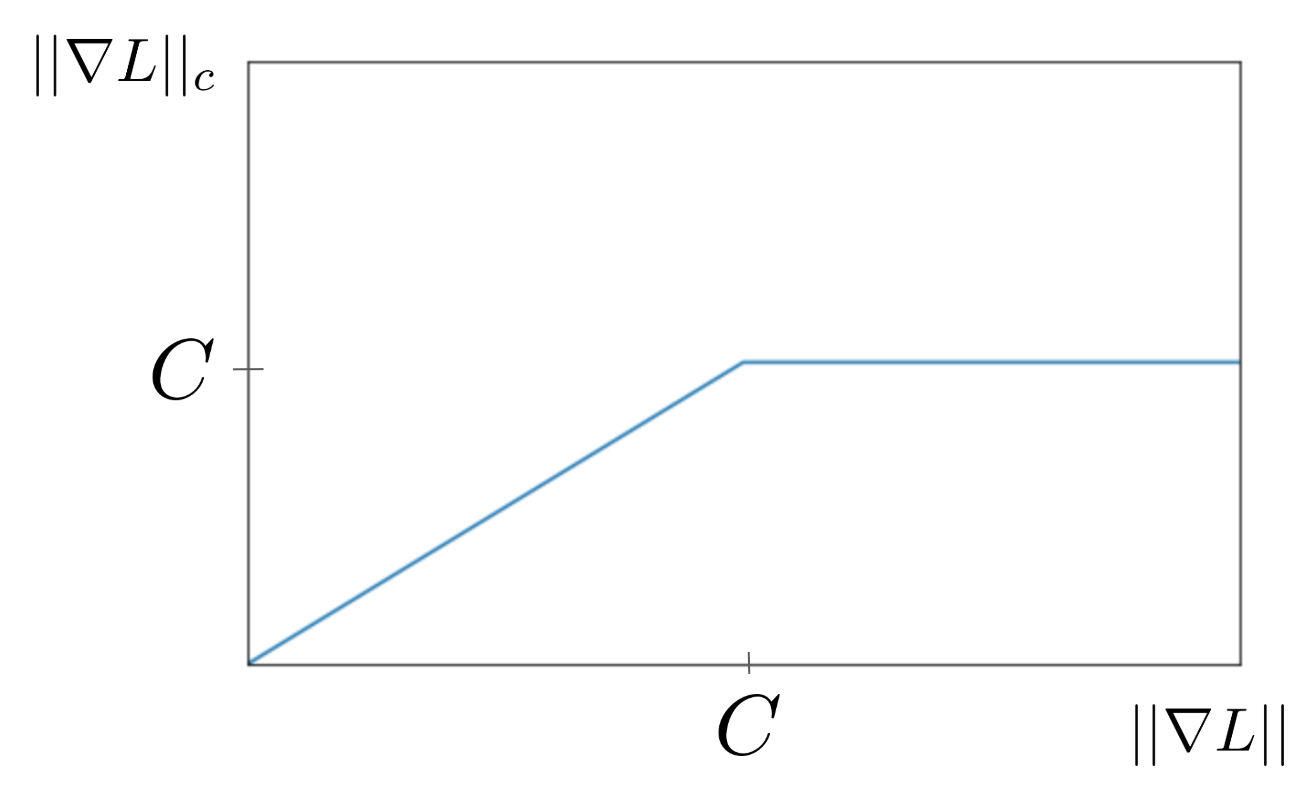
Gradient Clipping

One issue that we are often faced with within RNNs is that of **exploding or shrinking gradients**. If we think of the recursive layer as a very deep network. When calculating the gradients, we do so at every iteration of the hidden state. If the gradient of the loss relative to the weights at any given position becomes very big, this will have a multiplicative effect as it feeds forward through all the iterations of the recurrent layer. This can cause gradients to explode, getting very large very quickly. If we have large gradients, it can cause instability in our network. On the other hand, if our gradients within our hidden state are very small, this will again have a multiplicative effect and the gradients become close to zero. This means that the gradients can become too small to accurately update our parameters via gradient descent, meaning our model fails to learn.

One technique we can use to prevent our gradient from exploding is to use **gradient clipping**. This technique limits our gradients to prevent them from becoming too large. We simply choose a hyperparameter [](https://www.codecogs.com/eqnedit.php?latex=C%250) and can calculate our clipped gradient as:

[](https://www.codecogs.com/eqnedit.php?latex=%7C%7C%5Cnabla%20L%7C%7C_%7Bc%7D%20%3D%20max(%7C%7C%5Cnabla%20L%7C%7C%2CC)%250)

The following diagram shows how two variables related to one another:



Another technique we can use to prevent exploding or disappearing gradients is to shorten our input sequence length. The effective depth of our recurrent layer depends on the length of our input sequence as the sequence length determines how many iterative updates we need to perform on our hidden state. The fewer number of steps in this process, the smaller the multiplicative effects of our gradient accumulation between hidden states will be. By intelligently picking the maximum sequence length as a hyperparameter in our model, we can help prevent exploding and vanishing gradients.

Working with LSTMs

While RNNs allow us to use sequences of words as inputs to our models, they are far from perfect. RNNs suffer from two main flaws, which can be partially remedied by using a more sophisticated version of the RNN, known as a LSTM.

The basic structure of RNN, means that it is very difficult to retain information long term. Consider a sentence that’s 20 words long. From our first word in the sentence affecting the initial hidden state, to the last word in the sentence , our hidden state has been updated 20 times. From the beginning of our sentence, to our final hidden state, it is very difficult for an RNN to retain information about words at the beginning of the sentence. This means that RNNs aren’t very good at capturing long-term dependencies within sequences. This also ties into the vanishing gradient problem mentioned earlier where it is very inefficient to back propagate through long, sparse sequences of vectors.

Consider a long paragraph where we are trying to predict the next word. The sentence begins “I study maths…” and ends “my final exam is in ….”. Intuitively, we would expect the next work to be “maths” or some maths related field, however in an RNN model on a long sequence, our hidden state may struggle to retain the information for the beginning of the sentence until the end of the sentence through multiple update steps.

We also note that RNNs are poor at capturing the context of words within a sentence as a whole. We saw earlier when looking at our n-grams models that the meaning of a word in a sentence is dependent on its context within a sentence, determined by the words that occur before it and the words that occur after it. Within an RNN, our hidden state updates in one direction only. In a single forward pass, our hidden state is initialized and the first word in the sequence is passed into it. This process is then repeated with all the subsequent words in the sentence sequentially until we are left with our final hidden state. This means that for any given word in a sentence, we have only considered the cumulative effect of words that have occurred before it in the sentence up to that point. We do not consider any words that follow it, meaning we do not capture the full context of each word in the sentence.

In another example, we now wish to predict the missing word in a sentence, but it now occurs towards the beginning as opposed to at the end. “I grew up in … so I can speak fluent Dutch”. Here, we can intuitively guess that the person grew up in the Netherlands from the fact that they speak Dutch. However, because an RNN parses this information sequentially, it would only be using “I grew up in…” to make a prediction, missing the other key context within the sentence.

Both of these issues can be partially addressed using LSTMs.

Introducing LSTMs

LSTMs are more advanced versions of the RNN which contain two new properties. An **update gate** and a **forget gate**. These two additions make it easier for the network to learn long term dependencies. Consider the following film review:

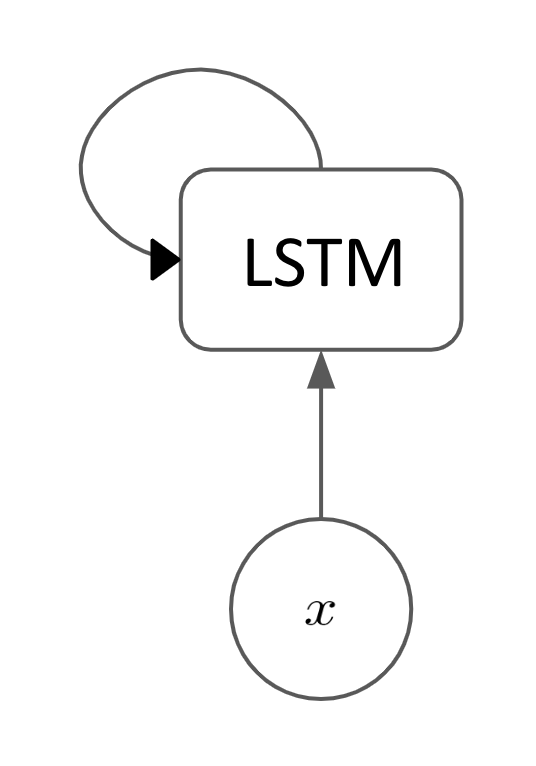
The film was amazing. I went to see it with my wife and my daughters on Tuesday afternoon. Although it didn’t expect it to be very entertaining, it turned out to be loads of fun. We would definitely go back and see it again given the chance.

In a sentiment analysis, it is clear that not all of the words in the sentence are relevant in determining if it is a positive or negative review. We repeat this sentence, but now highlight only the words that are relevant to gauging the sentiment of the review:

“The film was **amazing**. I went to see it with my wife and my daughters on Tuesday afternoon. Although it **didn’t expect it to be very entertaining**, it turned out to be **loads of fun**. We would **definitely go back and see it again** given the chance.”

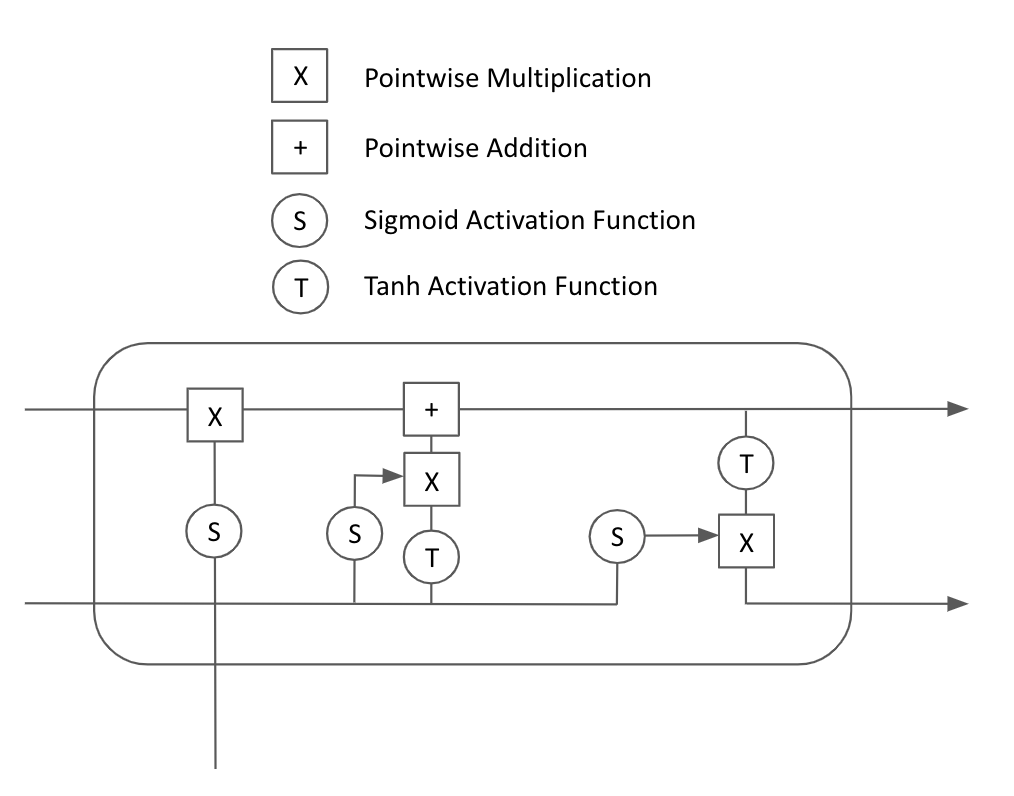
LSTMs attempt to do exactly this; remembering the relevant words within a sentence while forgetting all irrelevant information. By doing this, it stops the irrelevant information diluting the relevant information, meaning that long term dependencies can be better learned across long sequences.

LSTMs are very similar in structure to RNNs. While there is a hidden state that is carried over between steps within the LSTM, the inner workings of the LSTM cell itself is different to that of the RNN:

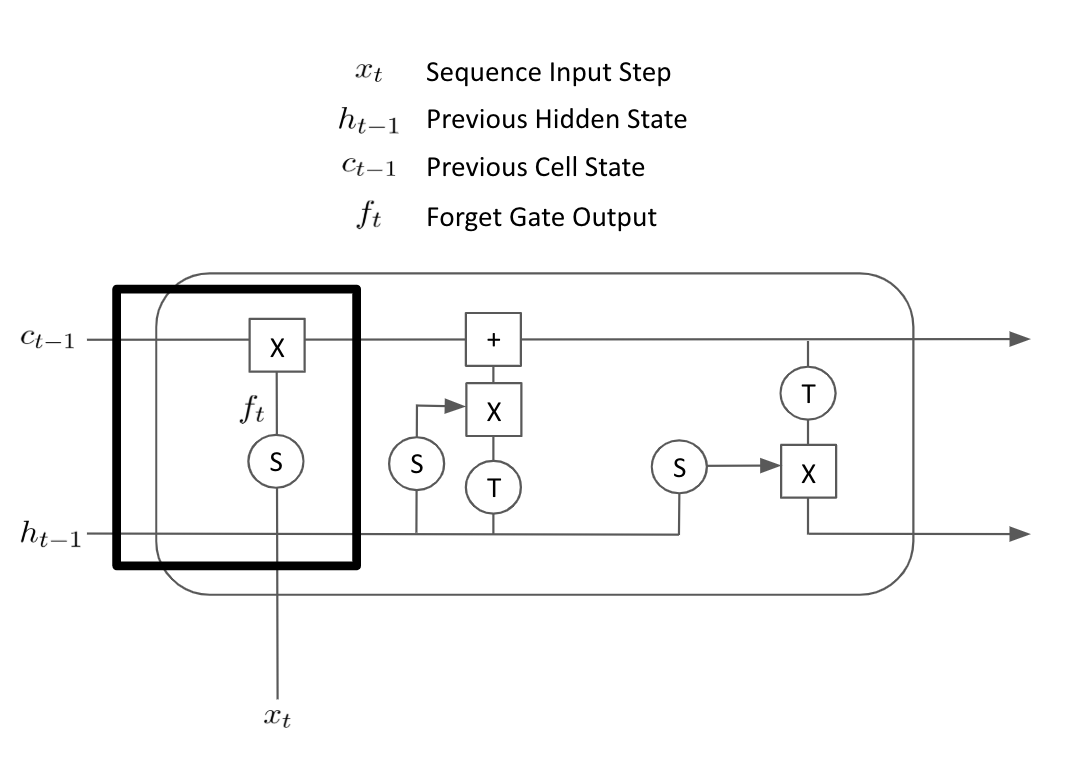


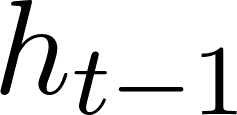
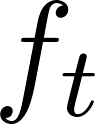
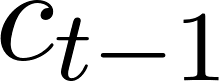
LSTM Cells

While an RNN cell just takes the previous hidden state and the new input step and calculates the next hidden state using some learned parameters, the inner workings of a LSTM cell are significantly more complicated:

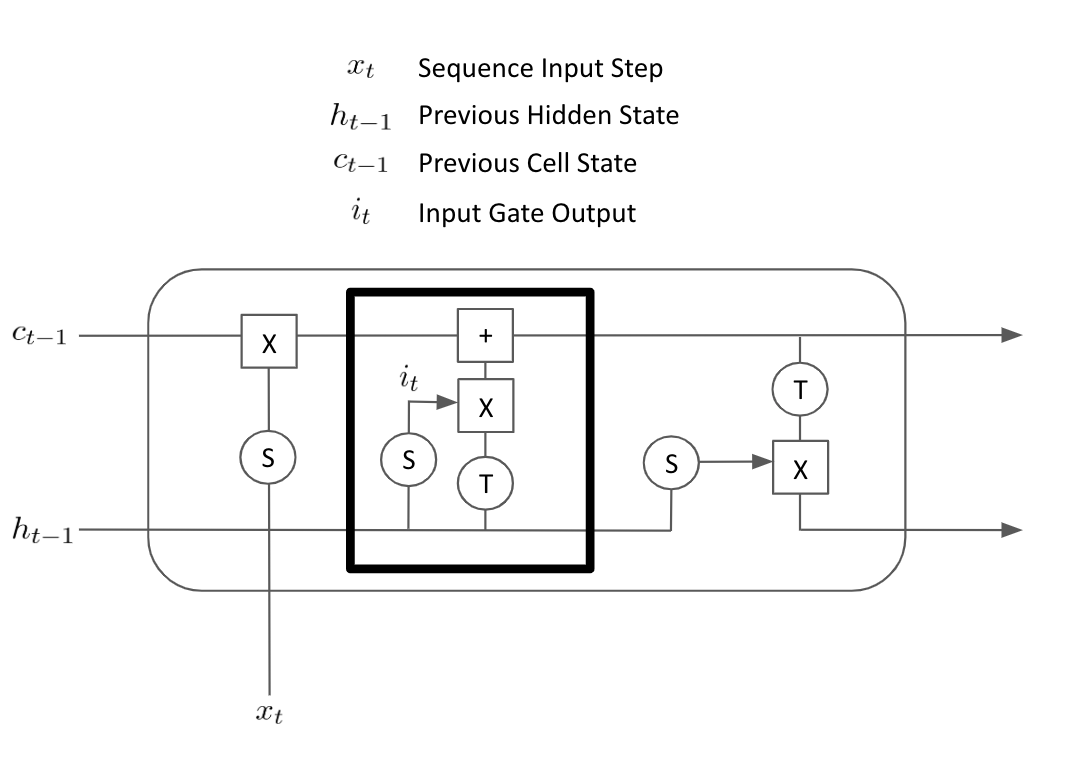


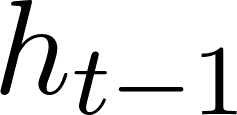
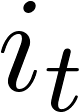
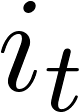
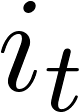
While this looks significantly more daunting than the RNN, we will explain each component of the LSTM cell in turn. We first look at the **forget gate** (indicated by the bold rectangle):



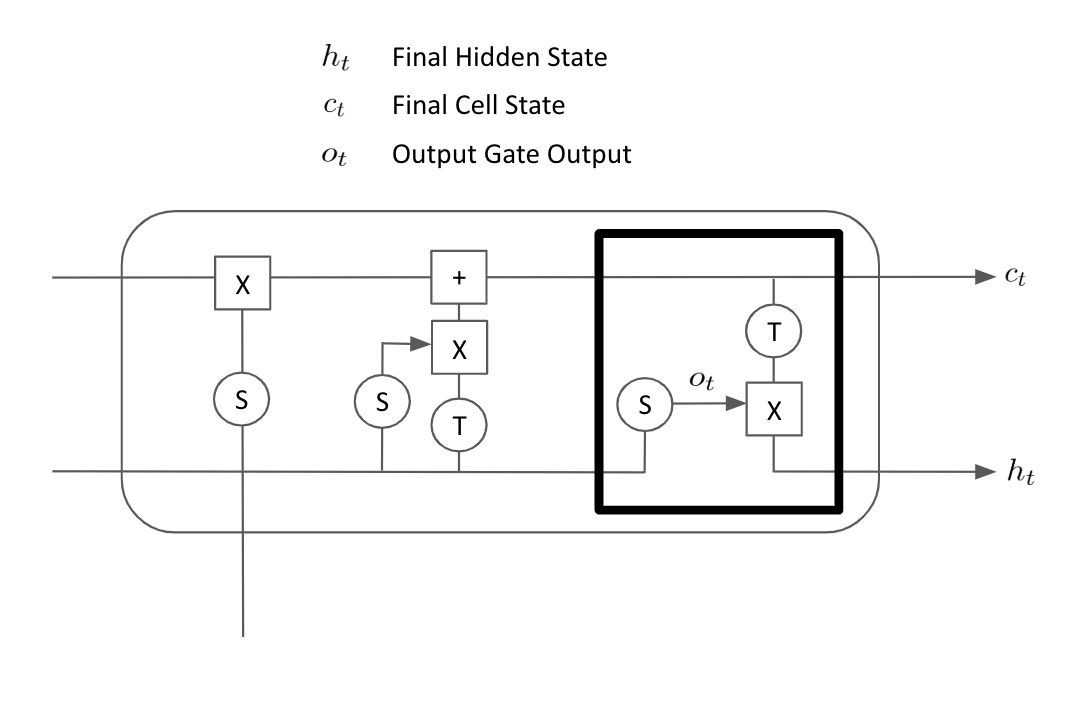
The forget gate essentially learns which elements of the sequence to forget. The previous hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7Bt-1%7D%250) and the latest input step [](https://www.codecogs.com/eqnedit.php?latex=x_%7Bt%7D%250) are concatenated together and passed through a matrix of learned weights on the forget gate and a sigmoid function that squashes the values between 0 and 1. This resulting matrix [](https://www.codecogs.com/eqnedit.php?latex=f_%7Bt%7D%250) is multiplied pointwise by the cell state from the previous step [](https://www.codecogs.com/eqnedit.php?latex=c_%7Bt-1%7D%250). This effectively applies a mask to the previous cell state so that only the relevant information from the previous cell state is brought forward.

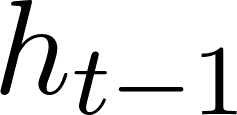
Next, we look at the **input gate**:



The input gate again takes the concatenated previous hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7Bt-1%7D%250) and current sequence input [](https://www.codecogs.com/eqnedit.php?latex=x_%7Bt%7D%250) and passes this through a sigmoid function with learned parameters which outputs another matrix [](https://www.codecogs.com/eqnedit.php?latex=i_%7Bt%7D%250), consisting of values between 0 and 1. The concatenated hidden state and sequence input also pass through a tanh function which squashes the output between -1 and 1. This is multiplied with matrix [](https://www.codecogs.com/eqnedit.php?latex=i_%7Bt%7D%250). This means that the learned parameters required to generate [](https://www.codecogs.com/eqnedit.php?latex=i_%7Bt%7D%250) effectively learns which elements should be kept from the current time step in our cell state. This is then added to the current cell state to get our final cell state which will be carried over to the next time step.

Finally, we at the final element of the LSTM cell; the **output gate:**



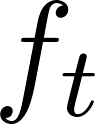
The output gate calculates the final outputs of the LSTM cell; both the cell state and the hidden state that is carried over to the next step. The cell state [](https://www.codecogs.com/eqnedit.php?latex=c_%7Bt%7D%250) is unchanged from the previous two steps and is a product of the forget gate and the input gate. The final hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7Bt%7D%250) is calculated by taking the concatenated previous hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7Bt-1%7D%250) and the current time step input [](https://www.codecogs.com/eqnedit.php?latex=x_%7Bt%7D%250) and passing through a sigmoid function with some learned parameters to get the output gate output [](https://www.codecogs.com/eqnedit.php?latex=o_%7Bt%7D%250). The final cell state [](https://www.codecogs.com/eqnedit.php?latex=c_%7Bt%7D%250) is passed through a tanh function and multiplied by the output gate output [](https://www.codecogs.com/eqnedit.php?latex=o_%7Bt%7D%250) to calculate the final hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7Bt%7D%250). This means that the learned parameters on the output gate effectively control which elements of the previous hidden state and current output are combined with the final cell state to carry over to the next time step as the new hidden state.

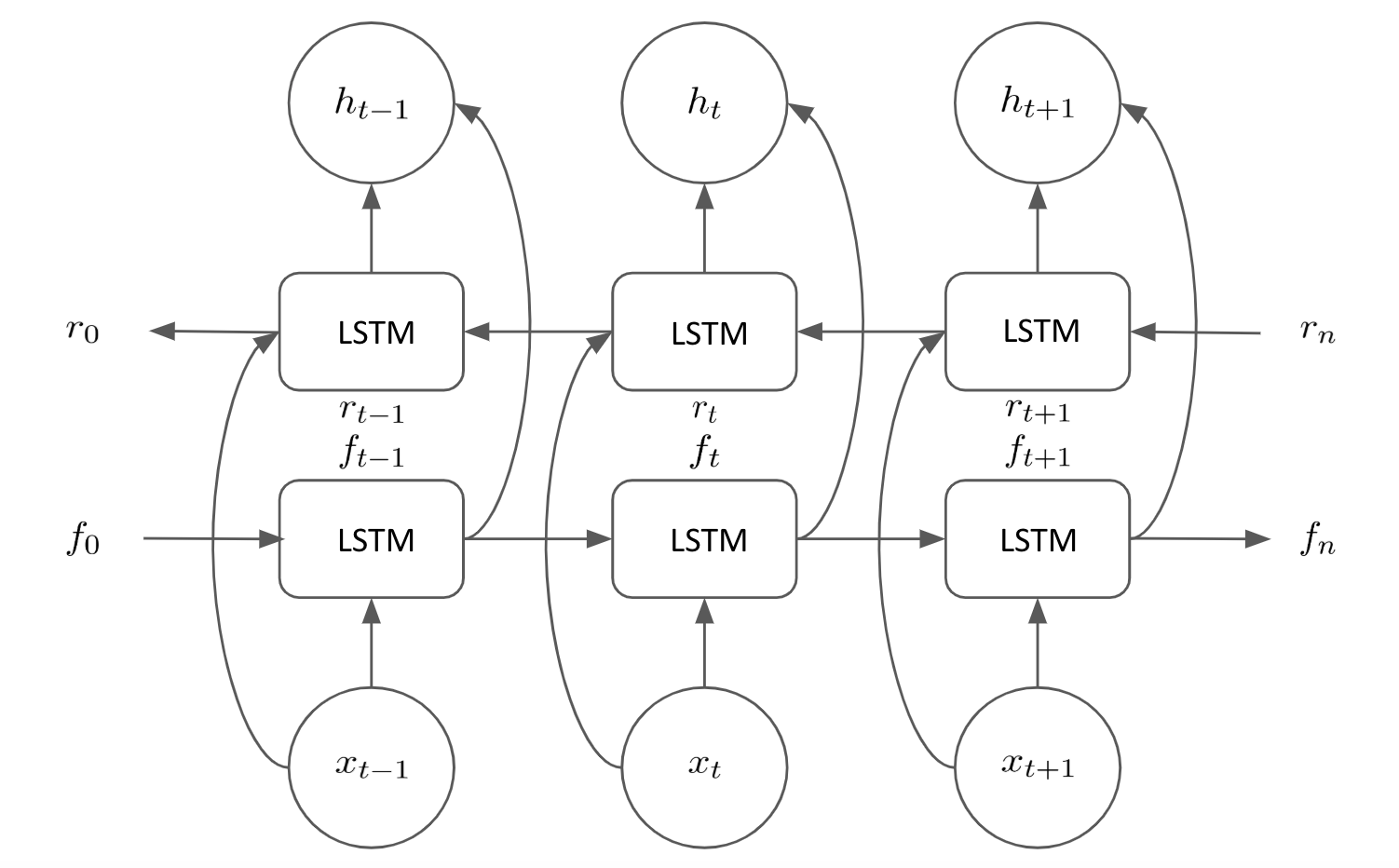
In our forward pass, we simply iterate through the model, initializing our hidden state and cell state and updating these at each time step using the LSTM cells until we are left with a final hidden state which is output to the next layer of our neural network. By backpropagation through all the layers of our LSTM, we are able to calculate the gradients relative to the loss of the network and thus, which direction to update our parameters through gradient descent. We learn several matrices or parameters; one for the input gate, one for the output gate and one for the forget gate.

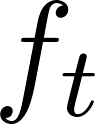
Because we are learning more parameters than for a simple RNN and our computation graph is more complex, the process of backpropagating through the network and updating the weights will likely take longer than for a simple RNN. However, despite the longer training time, we have shown that the LSTM offers significant advantages over a conventional RNN as the output gate, input gate and forget gate all combine to give the model the ability to determine which elements of the input should be used to update the hidden state and which elements of the hidden state should be forgotten going forward which means the model is better able to form long term dependencies and retain information from many sequence steps ago.

Bidirectional LSTMs

We mentioned that a failure of simple RNNs is that they fail to capture the full context of a word within a sentence as they are backwards-looking only. At each time step of the RNN, only the previously seen words are considered and the words occurring next within the sentence are not taken into account. While basic LSTMs are similarly backwards-facing, we can use a modified version of the LSTM known as a **bidirectional LSTM**  which considers both the words before and after it at each time step within the sequence.

The bidirectional LSTM processes sequence in regular order and in reverse order simultaneously, maintaining two hidden states. We’ll call these [](https://www.codecogs.com/eqnedit.php?latex=f_%7Bt%7D%250) for the forward hidden state and [](https://www.codecogs.com/eqnedit.php?latex=r_%7Bt%7D%250) for the reverse hidden state:



Here, we can see that we maintain these two hidden states throughout the whole process and use these to calculate a final hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7Bt%7D%250). Therefore, if we wish to calculate the final hidden state at timestep [](https://www.codecogs.com/eqnedit.php?latex=t%250), we use the forward hidden state [](https://www.codecogs.com/eqnedit.php?latex=f_%7Bt%7D%250), which has seen all words up to and including input [](https://www.codecogs.com/eqnedit.php?latex=x_%7Bt%7D%250), as well as the reverse hidden state [](https://www.codecogs.com/eqnedit.php?latex=r_%7Bt%7D%250) which has seen all the words after and including [](https://www.codecogs.com/eqnedit.php?latex=x_%7Bt%7D%250). Therefore, our final hidden state [](https://www.codecogs.com/eqnedit.php?latex=h_%7Bt%7D%250) is comprised of hidden states that have seen all the words in the sentence, not just the words occurring before timestep [](https://www.codecogs.com/eqnedit.php?latex=t%250). This means that the context of any given word within the whole sentence can be better captured. Bidirectional LSTMs have been shown to offer improved performance on several NLP tasks over conventional uni-directional LSTMs.

Building a Sentiment Analyzer using LSTMs

We now look to build our own simple LSTM to categorize sentences based on their sentiment. We will train our model on a dataset of 3,000 reviews that have been categorized as positive or negative. These reviews come from 3 different sources; film reviews, product reviews and location reviews in order to ensure our sentiment analyzer is robust. The dataset is balanced so consists of 1,500 positive reviews and 1,500 negative. We start by importing our dataset and examining it:

with open("sentiment labelled sentences/sentiment.txt") as f:

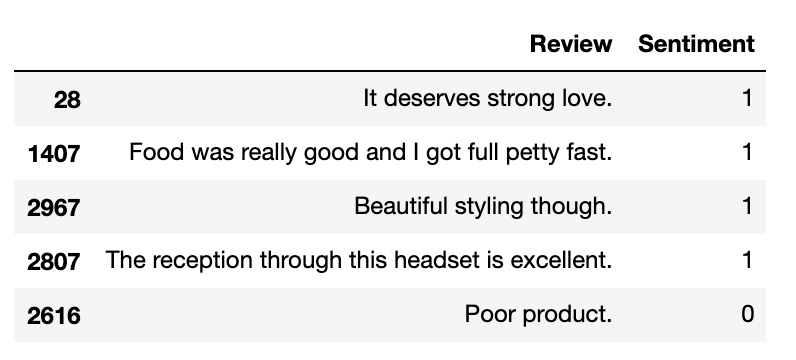
reviews = f.read()

data = pd.DataFrame([review.split('\t') for review in reviews.split('\n')])

data.columns = ['Review','Sentiment']

data = data.sample(frac=1)

Which returns the following output:

****

We read in our dataset from the file. Our dataset is tab separated so we split by tab and the new line character. We rename our columns and then use the sample function to randomly shuffle our data. Looking at our dataset, the first thing we need to be able to do is preprocess our sentences to feed them into our LSTM model.

Preprocessing the data

First, we create a function to tokenize our data; splitting each review into a list of individual pre-processed words. We loop through our dataset and for every review we remove punctuation, convert to lowercase and remove any trailing white space. We then use the NLTK tokenizer to create individual tokens from this pre-processed text.

def split\_words\_reviews(data):

text = list(data['Review'].values)

clean\_text = []

for t in text:

clean\_text.append(t.translate(str.maketrans('', '', punctuation)).lower().rstrip())

tokenized = [word\_tokenize(x) for x in clean\_text]

all\_text = []

for tokens in tokenized:

for t in tokens:

all\_text.append(t)

return tokenized, set(all\_text)

reviews, vocab = split\_words\_reviews(data)

reviews[0]

This results in the following output:

****

We return the reviews themselves, as well as a set of all words within all the reviews (ie. the vocabulary/corpus) which we will use to create our vocab dictionaries.

In order to fully prepare our sentences for entry into a neural network, we must convert our words into numbers. In order to do this, we create a couple of dictionaries, which will allow us to convert from word to index and from index to word. To do this, we simply loop through our corpus and assign an index to each unique word:

def create\_dictionaries(words):

word\_to\_int\_dict = {w:i+1 for i, w in enumerate(words)}

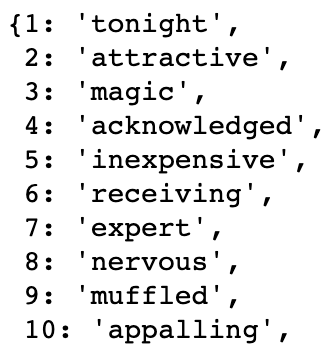
int\_to\_word\_dict = {i:w for w, i in word\_to\_int\_dict.items()}

return word\_to\_int\_dict, int\_to\_word\_dict

word\_to\_int\_dict, int\_to\_word\_dict = create\_dictionaries(vocab)

int\_to\_word\_dict

This gives the following output:

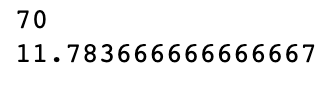


Our neural network will take inputs of a fixed length, however if we explore our reviews, we will see that our reviews are all of different lengths. In order to ensure that all our inputs are of the same length, we will **pad** our input sentences. This essentially means that we add empty tokens to our shorter sentences so that all sentences are of the same length. We first must decide on the length of the padding we wish to implement. We calculate the maximum length of a sentence in our input reviews, as well as the average length:

print(np.max([len(x) for x in reviews]))

print(np.mean([len(x) for x in reviews]))

Which gives:



We see that our longest sentence is 70 words long and our average sentence length is 11.78. To capture all the information from all our sentences, we would want to pad all of our sentences to be of length 70. However, using longer sentences means longer sequences which would cause our LSTM layer to become deeper. This means that model training takes longer as we have to backpropagate our gradients through more layers, but it also means that a large percentage of our inputs would just be sparse and full of empty tokens which makes learning from our data much less efficient. This is illustrated by the fact that our maximum sentence length is much larger than our average sentence length. In order to capture the majority of our sentence information without unnecessarily padding our inputs and making them too sparse we opt to use an input size of 50. You may wish to experiment with using different input sizes between 20 and 70 to see how this affects your model performance.

We create a function that allows us to pad our sentences so that they are all the same size. For reviews shorter than the sequence length, we pad them with empty tokens. For reviews longer than the sequence length, we simply drop any tokens over the maximum sequence length:

def pad\_text(tokenized\_reviews, seq\_length):

reviews = []

for review in tokenized\_reviews:

if len(review) >= seq\_length:

reviews.append(review[:seq\_length])

else:

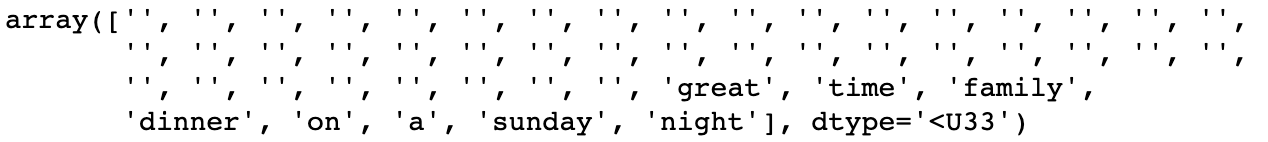
reviews.append(['']\*(seq\_length-len(review)) + review)

return np.array(reviews)

padded\_sentences = pad\_text(reviews, seq\_length = 50)

padded\_sentences[0]

Our padded sentence looks like this:



We must make one further adjustment to allow for the use of empty tokens within our model. Currently our vocabulary dictionaries do not know how to convert empty tokens to integers to use within our network. Because of this, we manually add these to our dictionaries with index 0 which means that empty tokens will be given a value of 0 when being fed into our model:

int\_to\_word\_dict[0] = ''

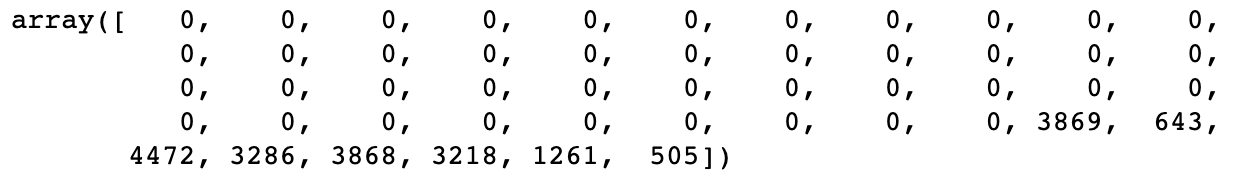
word\_to\_int\_dict[''] = 0

We are now very nearly ready to begin training our model. We perform one final step of preprocessing and encode all our padded sentences as numeric sequences for feeding into our neural network. This means that our padded sentence above now looks like this:

encoded\_sentences = np.array([[word\_to\_int\_dict[word] for word in review] for review in padded\_sentences])

encoded\_sentences[0]

Our encoded sentence is represented as:

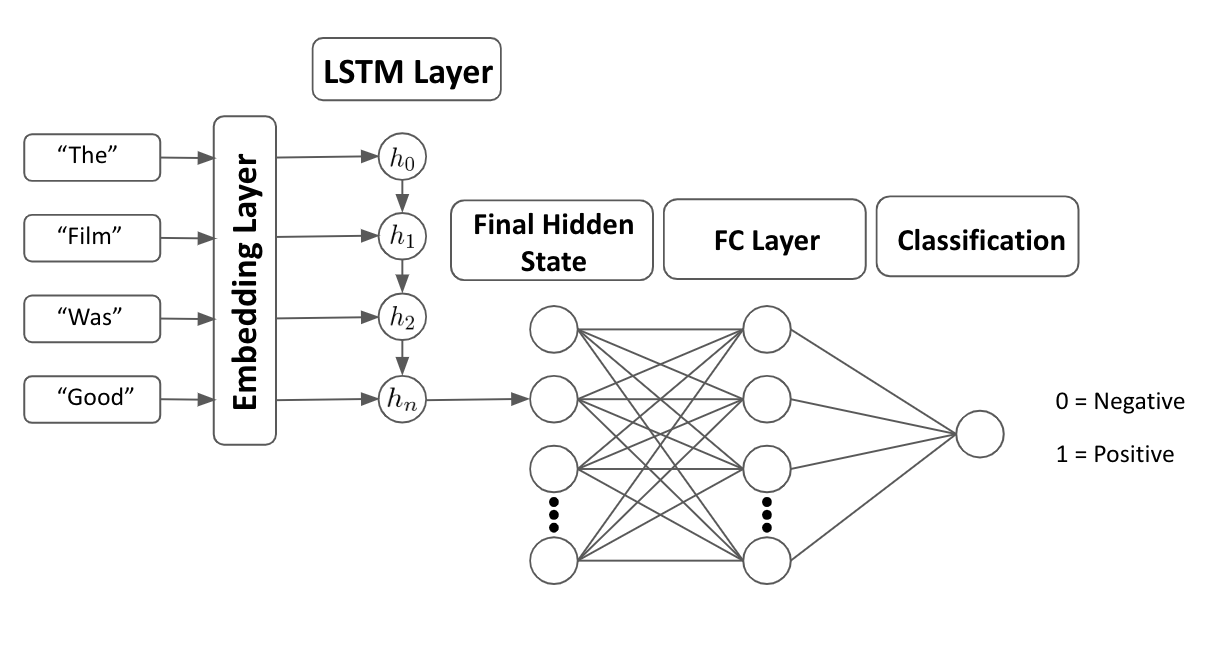
****

Now that we have all our input sequences encoded as numerical vectors, we are ready to begin designing our model architecture.

Model Architecture

Our model will consist of several main parts. Besides the input and output layers that are common to many neural networks we will first require an **embedding layer**. This is so our model learns vector representations of words it is being trained on. We could opt to use pre-computed embeddings (such as GLoVE), but for demonstrative purposes we will train our own embedding layer. Our input sequences are fed through the input layer and come out as sequences of vectors.

These vector sequences are then fed into our **LSTM layer**. As explained in detail earlier in the chapter, the LSTM layer learns sequentially from our sequence of embeddings and outputs a single vector output representing the final hidden state of the LSTM layer. This final hidden state is finally passed through a further **hidden layer** before the final output node predicts a value between 0 and 1 indicating whether the input sequence was a positive or negative review. This means that our model architecture looks something like this:



We will now demonstrate how to code this model from scratch using PyTorch. We create our class entitled “SentimentLSTM” which inherits from the nn.Module class. We define our init parameters as the size of our vocab, the number of LSTM layers our model will have and the size of our model’s hidden state:

class SentimentLSTM(nn.Module):

def \_\_init\_\_(self, n\_vocab, n\_embed, n\_hidden, n\_output, n\_layers, drop\_p = 0.8):

super().\_\_init\_\_()

self.n\_vocab = n\_vocab

self.n\_layers = n\_layers

self.n\_hidden = n\_hidden

We then define each of the layers of our network. Firstly, our embedding layer which will have the length of the number of words in our vocabulary and the size of the embedding vectors as a hyperparameter n\_embed to be specified. Our LSTM layer is defined using the output vector size from the embedding layer, the length of the model’s hidden state and the number of layers our LSTM layer will have. We also add an argument to specify that our LSTM can be trained on batches of data and an argument to allow us to implement network regularization via dropout. We define a further dropout layer with probability drop\_p (a hyperparameter to be specified on model creation), as well as our definitions of our final fully connected layer and output/prediction node (with a sigmoid activation function):

self.embedding = nn.Embedding(n\_vocab, n\_embed)

self.lstm = nn.LSTM(n\_embed, n\_hidden, n\_layers, batch\_first = True, dropout = drop\_p)

self.dropout = nn.Dropout(drop\_p)

self.fc = nn.Linear(n\_hidden, n\_output)

self.sigmoid = nn.Sigmoid()

Next, we must define our forward pass within our model class. Within this forward pass, we just chain together the outputs of one layer to become the input into our next layer. Here we see that our embedding layer takes input\_words as an input and outputs embedded words. Then our LSTM layer takes embedded words as an input and outputs lstm\_out. The only nuance here is that we use view() to reshape our tensors from the LSTM output to be the correct size for input into our fully connected layer. The same also applies for reshaping the output of our hidden layer to match that of our output node. Note that our output will return a prediction for class = 0 and class = 1 so we slice the output to only return prediction for class 1, ie. the probability that our sentence is positive:

def forward (self, input\_words):

embedded\_words = self.embedding(input\_words)

lstm\_out, h = self.lstm(embedded\_words)

lstm\_out = self.dropout(lstm\_out)

lstm\_out = lstm\_out.contiguous().view(-1, self.n\_hidden)

fc\_out = self.fc(lstm\_out)

sigmoid\_out = self.sigmoid(fc\_out)

sigmoid\_out = sigmoid\_out.view(batch\_size, -1)

sigmoid\_last = sigmoid\_out[:, -1]

return sigmoid\_last, h

We also define a function called init\_hidden which initializes our hidden layer with the dimensions of our batch size. This allows our model to train and predict on many sentences at once, rather than just training on one sentence at a time, if we so wish. Note that we define device as “cpu” here to run it on our local processor. However, it is possible to set this to a cuda-enabled GPU in order to train it on a GPU if you have one:

def init\_hidden (self, batch\_size):

device = "cpu"

weights = next(self.parameters()).data

h = (weights.new(self.n\_layers, batch\_size, self.n\_hidden).zero\_().to(device),

weights.new(self.n\_layers, batch\_size, self.n\_hidden).zero\_().to(device))

return h

We then initialize our model by creating a new instance of the SentimentLSTM class. We pass the size of our vocab, the size of our embeddings, the size of our hidden state as well as the output size and the number of layers in our LSTM:

n\_vocab = len(word\_to\_int\_dict)

n\_embed = 50

n\_hidden = 100

n\_output = 1

n\_layers = 2

net = SentimentLSTM(n\_vocab, n\_embed, n\_hidden, n\_output, n\_layers)

Now that we have defined our model architecture fully, it’s time to begin training our model.

Training the model

To train our model, we must first define our datasets. We will train out model using a training set of data, evaluate our trained model at each step on a validation set and then finally measure our model’s final performance using an unseen test set of data. The reason we use a test set separate from our validation training is because we may wish to fine tune our model hyperparameters based on the loss against the validation set. If we do this, we may end up picking the hyperparameters that are optimal in performance only for that particular validation set of data. We evaluate a final time against an unseen test set to make sure our model generalizes well to data it hasn’t seen before at any part of the training loop.

We have already defined our model inputs ([](https://www.codecogs.com/eqnedit.php?latex=X%250)) as encoded\_sentences, but we must also define our model output ( [](https://www.codecogs.com/eqnedit.php?latex=y%250)). We do this simply as:

labels = np.array([int(x) for x in data['Sentiment'].values])

Next, we define our training and validation ratios. In this case, we will train our model on 80% of the data, validate on a further 10% of the data and finally test on the remaining 10% of the data:

train\_ratio = 0.8

valid\_ratio = (1 - train\_ratio)/2

We then use these ratios to slice our data and transform them into Tensors and then Tensor Datasets:

total = len(encoded\_sentences)

train\_cutoff = int(total \* train\_ratio)

valid\_cutoff = int(total \* (1 - valid\_ratio))

train\_x, train\_y = torch.Tensor(encoded\_sentences[:train\_cutoff]).long(), torch.Tensor(labels[:train\_cutoff]).long()

valid\_x, valid\_y = torch.Tensor(encoded\_sentences[train\_cutoff : valid\_cutoff]).long(), torch.Tensor(labels[train\_cutoff : valid\_cutoff]).long()

test\_x, test\_y = torch.Tensor(encoded\_sentences[valid\_cutoff:]).long(), torch.Tensor(labels[valid\_cutoff:])

train\_data = TensorDataset(train\_x, train\_y)

valid\_data = TensorDataset(valid\_x, valid\_y)

test\_data = TensorDataset(test\_x, test\_y)

Then we use these datasets to create PyTorch DataLoader objects. DataLoader allows us to batch process our datasets with the batch\_size parameter, allowing different batch sizes to be easily passed to our model. In this instance we will keep it simple and set batch\_size = 1 which means our model will be trained on individual sentences, rather than using larger batches of data. We also opt to randomly shuffle our DataLoader so data is passed through our neural network in random order, rather than the same order each epoch, potentially removing any biases results from the training order:

batch\_size = 1

train\_loader = DataLoader(train\_data, batch\_size = batch\_size, shuffle = True)

valid\_loader = DataLoader(valid\_data, batch\_size = batch\_size, shuffle = True)

test\_loader = DataLoader(test\_data, batch\_size = batch\_size, shuffle = True)

Now that we have defined our DataLoader for each of our three datasets, we define our training loop. We first define a number of hyperparameters which will be used within our training loop. Most importantly we define our loss function as binary cross entropy (as we are dealing with predicting a single binary class) and we define our optimizer to be Adam with a learning rate of 0.001. We also define our model to run for a short number of epochs (to save time) and set clip = 5 to define our gradient clipping:

print\_every = 2400

step = 0

n\_epochs = 3

clip = 5

criterion = nn.BCELoss()

optimizer = optim.Adam(net.parameters(), lr = 0.001)

The main body of our training loop looks like this:

for epoch in range(n\_epochs):

h = net.init\_hidden(batch\_size)

for inputs, labels in train\_loader:

step += 1

net.zero\_grad()

output, h = net(inputs)

loss = criterion(output.squeeze(), labels.float())

loss.backward()

nn.utils.clip\_grad\_norm(net.parameters(), clip)

optimizer.step()

Here, we just train our model for a number of epochs and for every epoch we first initialize our hidden layer using the batch size parameter. In this instance batch\_size = 1 as we are just training our model one sentence at a time. For each batch of input sentences and labels within our train loader, we first zero our gradients (to stop them accumulating) and calculate our model outputs using the forward pass of our data using the model’s current state. Using this output we then calculate our loss using the predicted output from the model and the correct labels. We then perform a backward pass of this loss through our network to calculate the gradients at each stage. Next, we use the grad\_clip\_norm function to clip our gradients as this will stop our gradients from exploding as mentioned earlier in this chapter. We defined clip = 5 meaning the maximum gradient at any given node is 5. Finally we update our weights using the gradients calculated on our backward pass by calling optimizer.step()

If we run this loop by itself, we will train our model. However, we wish to evaluate our model performance after every epoch in order to determine its performance on a validation set of data. We do this as follows:

if (step % print\_every) == 0:

net.eval()

valid\_losses = []

for v\_inputs, v\_labels in valid\_loader:

v\_output, v\_h = net(v\_inputs)

v\_loss = criterion(v\_output.squeeze(), v\_labels.float())

valid\_losses.append(v\_loss.item())

print("Epoch: {}/{}".format((epoch+1), n\_epochs),

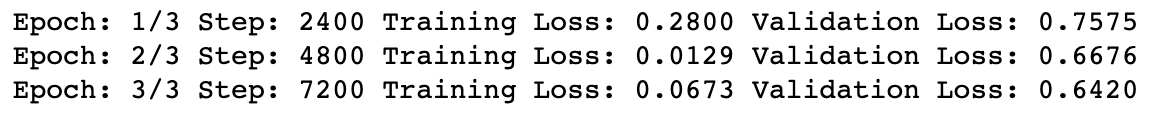
"Step: {}".format(step),

"Training Loss: {:.4f}".format(loss.item()),

"Validation Loss: {:.4f}".format(np.mean(valid\_losses)))

net.train()

This means at the end of every epoch, our model calls net.eval() to freeze the weights of our model and performs a forward pass using our data as before. Note that dropout is also not applied when we are in evaluation mode. However, this time instead of using the training data loader, we use the validation loader. By doing this we can calculate the total loss of the model’s current state over our validation set of data. Finally we print our results and call net.train() to unfreeze our model’s weights so we can train again on the next epoch. Our output looks something like this:



Finally, we can save our model for future use:

torch.save(net.state\_dict(), 'model.pkl')

After training our model for 3 epochs, we notice two main things. We’ll start with the good news first - our model is learning something! Not only has our training loss fallen , but we see that our loss on the validation set has fallen after each epoch. This means that our model is better at predicting sentiment on an unseen set of data after just 3 epochs! The bad news however is that our model is massively overfit. Our training loss is much lower than that of our validation loss, showing that while our model has learnt how to predict the training set of data very well, this doesn’t generalize as well to an unseen set of data. This was expected to happen as we are using a very small set of training data (just 2400 training sentences). As we are training a whole embedding layer, it is possible that many of the words occur just once in the training set and never in the validation set and vice versa, making it practically impossible for the model to generalize for all the different variety of words within our corpus.

In practice, we would hope to train our model on a much larger dataset to allow our model to learn how to generalize much better. We have also trained this model over a very short time period and have not performed hyperparameter tuning to determine the best possible iteration of our model. Feel free to try changing some of the parameters within the model (training time, hidden state size, embedding size etc.) in order to improve the performance of the model.

Although our model is overfit, it has still learned something. We now wish to evaluate our model on a final test set of data. We perform one final pass on the data using the test loader we defined earlier. Within this pass we loop through all of our test data and make predictions using our final model:

net.eval()

test\_losses = []

num\_correct = 0

for inputs, labels in test\_loader:

test\_output, test\_h = net(inputs)

loss = criterion(test\_output, labels)

test\_losses.append(loss.item())

preds = torch.round(test\_output.squeeze())

correct\_tensor = preds.eq(labels.float().view\_as(preds))

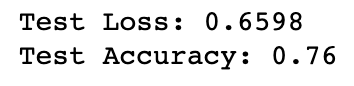
correct = np.squeeze(correct\_tensor.numpy())

num\_correct += np.sum(correct)

print("Test Loss: {:.4f}".format(np.mean(test\_losses)))

print("Test Accuracy: {:.2f}".format(num\_correct/len(test\_loader.dataset)))

Our performance on our test set of data is as follows:



We then compare our model predictions to our true labels to get correct\_tensor; a vector of whether each of our model’s predictions were correct or not. We then sum this vector and divide by its length to get our model’s total accuracy. Here, we get an accuracy of 76%. While our model is certainly far from perfect, given our very small training set and limited training time, this is not bad at all! This just serves to illustrate how useful LSTMs can be when it comes to learning from NLP data. Next, we will show how we can use our model to make predictions from new data.

Using our Model to Make Predictions

Now that we have a trained model, it should be possible to repeat our preprocessing steps on a new sentence, pass this into our model and get a prediction of it’s sentiment. We first create a function to preprocess our input sentence to predict:

def preprocess\_review(review):

review = review.translate(str.maketrans('', '', punctuation)).lower().rstrip()

tokenized = word\_tokenize(review)

if len(tokenized) >= 50:

review = tokenized[:50]

else:

review= ['0']\*(50-len(tokenized)) + tokenized

final = []

for token in review:

try:

final.append(word\_to\_int\_dict[token])

except:

final.append(word\_to\_int\_dict[''])

return final

We remove punctuation, trailing whitespace, convert to lowercase and tokenize our input sentence as before. We pad our sentence to a sequence of length 50 and then convert our tokens to numeric values using our precomputed dictionary. Note that our input may contain new words that our network hasn’t seen before. In this case our function treats these as an empty token.

Next we create our actual predict function. We preprocess our input review, convert it to a tensor and pass this into a data loader. We then loop through this data loader (even though it only contains one sentence) and pass our review through our network to obtain a prediction. Finally we evaluate our prediction and print whether it is a positive or negative review:

def predict(review):

net.eval()

words = np.array([preprocess\_review(review)])

padded\_words = torch.from\_numpy(words)

pred\_loader = DataLoader(padded\_words, batch\_size = 1, shuffle = True)

for x in pred\_loader:

output = net(x)[0].item()

msg = "This is a positive review." if output >= 0.5 else "This is a negative review."

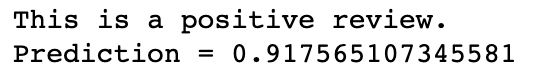
print(msg)

print('Prediction = ' + str(output))

Finally, we just call predict on our review to make a prediction:

predict("The film was good")

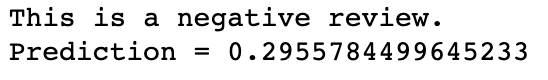
This results in the following output:



We also try predict on the negative value:

predict("It was not good")

This results in the following output:



We have now built an LSTM model to perform sentiment analysis from the ground up. Although our model is far from perfect, we have demonstrated how we can take some sentiment labelled reviews and train a model to be able to make predictions on new reviews. Next we will show how we can host our model on the Heroku cloud platform to so that other people can make predictions using your model

Deploying the application on Heroku

We have now trained our model on our local machine and we are able to use this to make predictions. However this isn’t necessarily any good if you want other people to be able to use your model to make predictions. If we host our model on a cloud-based platform like Heroku and create a basic API, other people will be able to make calls to the API in order to make predictions using our model.

Introducing Heroku

Heroku is a cloud based platform where you are able to host your own basic programmes. While the free-tier of Heroku has a maximum upload size of 500mb and limited processing power, this should be sufficient for us to host our model and create a basic API in order to make predictions using our model.

The first step is to create a free account on Heroku and install the Heroku app. Then, in the command line type the following command:

heroku login

To login using your account details. Then create a new heroku project by typing the following command:

heroku create sentiment-analysis-flask-api

Note that all project names are unique so you will need to pick a project name different to sentiment-analysis-flask-api.

Our first step is building a basic API using Flask.

Creating an API using Flask - File Structure

Creating an API is fairly simple using Flask as Flask contains a default template required to make an API.

First, in the command line, create a new folder for your flask API and navigate into it:

mkdir flaskAPI

cd flaskAPI

Then create a virtual environment within the folder. This will be the Python environment your API will use:

python3 -m venv vir\_env

Within your environment install all the packages you will need using pip. This includes all the packages that you use within your model program such as NLTK, Pandas, NumPy and PyTorch but also the packages you need to run the API like Flask and Gunicorn:

pip install nltk pandas numpy torch flask gunicorn

We then create a list of requirements that our API will use. Note that when we upload this to Heroku, Heroku will automatically download and install all the packages within this list. We can do this by typing the following:

pip freeze > requirements.txt

One adjustment we must make is to replace the Torch line within the requirements.txt file with the following:

<https://download.pytorch.org/whl/cpu/torch-1.3.1%2Bcpu-cp37-cp37m-linux_x86_64.whl>

This is a link to the wheel file of the version of PyTorch that only contains the CPU implementation. The full version of PyTorch that includes full GPU support is over 500mb in size so it will not run on the free Heroku cluster. Using this more compact version of PyTorch means that you will still be able to run your model using PyTorch on Heroku. Finally we create three more files within our folder, as well as a final directory for our models:

touch app.py

touch Procfile

touch wsgi.py

mkdir models

Now we have created all the files we will need for our Flash API and we are ready to start making adjustments to our file

Creating an API using Flask - API file

Within our app.py file, we can begin building our API. We first make all our imports and create a predict route. This allows us to call our API with the argument ‘predict’ in order to run a predict method within our API:

import flask

from flask import Flask, jsonify, request

import json

import pandas as pd

from string import punctuation

import numpy as np

import torch

from nltk.tokenize import word\_tokenize

from torch.utils.data import TensorDataset, DataLoader

from torch import nn

from torch import optim

app = Flask(\_\_name\_\_)

@app.route('/predict', methods=['GET'])

Next, we define our predict method within our app.py file. This is largely a rehash of our model file so to avoid repetition of code it is advised to look at the completed app.py within the GitHub. You will see that there are a few additional lines. Firstly, within our preprocess\_review function, we will see the lines:

with open('models/word\_to\_int\_dict.json') as handle:

word\_to\_int\_dict = json.load(handle)

This takes the word\_to\_int dictionary we computed within our main model notebook and loads it into our model. This is so our word indexing is consistent with our trained model. We then use this dictionary to convert our input text into an encoded sequence. Be sure to take the word\_to\_int\_dict.json file from the original notebook output and place it within the models directory.

Similarly, we must also load the weights from our trained model. We first define our SentimentLSTM class and the load our weights using torch.load . We will use the .pkl file from our original notebook so be sure to place this in the models directory as well:

model = SentimentLSTM(5401, 50, 100, 1, 2)

model.load\_state\_dict(torch.load("models/model\_nlp.pkl"))

We must also define the input and outputs of our API. We want our model to take the input from our API and pass this to our preprocess\_review function. We do this using request.get\_json():

request\_json = request.get\_json()

i = request\_json['input']

words = np.array([preprocess\_review(review=i)])

And to define our output, we return a JSON response consisting of the output from our model and a response code 200 which is what is returned by our predict function:

output = model(x)[0].item()

response = json.dumps({'response': output})

return response, 200

With the main body of our app completed, there are just two more additional things we must add in order to make our API run. We must first add the following to our wsi.py file:

from app import app as application

if \_\_name\_\_ == "\_\_main\_\_":

application.run()

Finally, add the following to our Procfile:

web: gunicorn app:app --preload

And that’s all that’s required for the app to run. We can test that our API runs by first starting the API locally using the following:

gunicorn — bind 0.0.0.0:8080 wsgi:application -w 1

Once the API is running locally, we can make a request to the API by passing it a sentence to predict the outcome:

curl -X GET http://0.0.0.0:8080/predict -H "Content-Type: application/json" -d '{"input":"the film was good"}'

If all is working correctly, you should receive a prediction back from the API. Now that we have our API making predictions locally, it is time to host it on Heroku in order to be able to make predictions in the cloud.

Creating an API using Flask - Hosting on Heroku

We must first commit our files to Heroku in a similar way to how we would commit files using GitHub. We define our working flaskAPI directory as a git folder by simply running the following command:

git init

Within the folder, we next add the following to the .gitignore file which will stop us adding unnecessary files to the Heroku repo:

vir\_env

\_\_pycache\_\_/

.DS\_Store

Finally, we add our first commit and push it to our heroku project:

git add . -A

git commit -m 'commit message here'

push push heroku master

This may take some time to compile as not only does the system have to copy all the file from your local directory to Heroku, but Heroku will automatically build your defined environment, installing all required packages and start running your API.

Now, if all has worked correctly, your API will be automatically running on the Heroku cloud. In order to make predictions, we simply make a request to the API using your own project name instead of sentiment-analysis-flask-api:

curl -X GET [https://](https://flask-ml-api-123.herokuapp.com/predict)sentiment-analysis-flask-api[.herokuapp.com/predict](https://flask-ml-api-123.herokuapp.com/predict) -H “Content-Type: application/json” -d '{"input”:”the film was good"}'

Whereby your API will return a prediction from the model. Congratulations, you have now learned to train an LSTM model from scratch, upload this to the cloud and make predictions using it. Going forward, hopefully this tutorial will serve as a basis for you to train your own LSTM models and deploy them on the cloud yourself.

Summary

In this chapter, we have discussed the fundamentals of recurrent neural networks and one of their main variations, the LSTM. We then showed how you can build your own RNN from scratch and deploy this on the cloud-based platform Heroku. While RNNs are often used for deep learning on NLP tasks, they are by no means the only neural network architecture suitable for this task. In the next chapter we will look at Convolutional Neural Networks and show how they can be used for NLP learning tasks.