Natural Language Processing with Tensorflow

Ashu Prasad Follow
Oct 20 · 19 min read

Photo by Kelly Sikkema on Unsplash

Hey all! In this post I attempt to summarize the course on Natural Language Processing in TensorFlow by Deeplearning.ai.

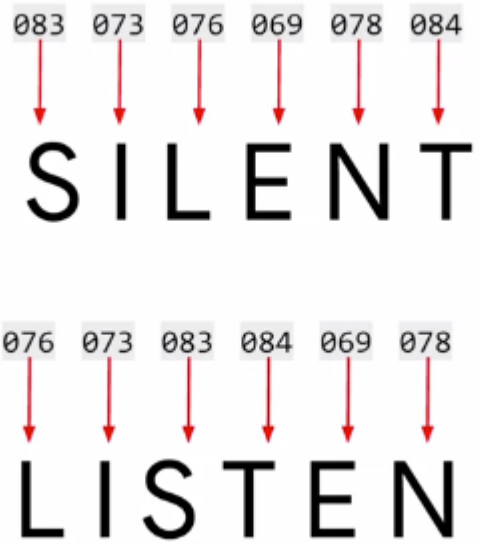
. . .

Week 1

When dealing with pictures, we already have pixel values which are numbers. However, when dealing with text, it has to be encoded so that it can be easily processed by a neural network.

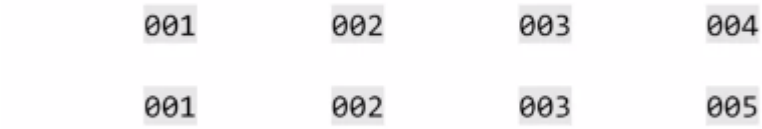
To encode the words, we could use their ASCII values. However, using ASCII values limits our semantic understanding of the sentence.

Eg:-



In the above two words, we have the same letters thus having the same ASCII values but each word is having a completely opposite meaning. Therefore, using ASCII values to extract meaning from the words is daunting task.

Now, instead of labelling each letter with a number i.e. ASCII values, we label each word. In the above sentences, we've labelled each word. The only difference is the last word. When we only view the labels, we observe a pattern.



We now begin to see similarity between the sentences. Now we can draw meaning out of this. From here on, we can begin to train a neural network which can understand the meaning of the sentences.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer

sentences = [
    'I love my dog',
    'I love my cat'
]

tokenizer = Tokenizer(num_words = 100)
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index
print(word_index)
```

Tokenizer will handle the heavy lifting in the code. Using tokenizer, we can label each word and provide a dictionary of the words being used in the sentences. We create an instance of tokenizer and assign a hyperparameter ***num_words*** to 100. This essentially takes the most common 100 words and tokenize them. For the above sentences, this is way too big as there are only 5 distinct words.

The ***fit_on_texts()*** method is used to encode the sentences.

The ***word_index*** method returns a dictionary of key value pairs where the key is the word in the sentence and the value is the label assigned to it. One can view this dictionary by printing it.

```
{'i': 1, 'my': 3, 'dog': 4, 'cat': 5, 'love': 2}
```

The ***word_index*** returns the above key value pairs. Notice that 'I' has been replaced by 'i'. This is what tokenizer does; it omits the punctuation.

```
sentences = [
    'I love my dog',
    'I love my cat',
    'You love my dog!'
]
```

The ***word_index*** of the above sentences returns the following dictionary.

```
{'i': 3, 'my': 2, 'you': 6, 'love': 1, 'cat': 5, 'dog': 4}
```

Notice, that **'dog!'** is not treated as a separate word just because there is an exclamation next to it. Instead, the exclamation being a punctuation is stripped and only the word is

included. **'You'** being another new word has been assigned a new value.

```
test_data = [
    'i really love my dog',
    'my dog loves my manatee'
]

test_seq = tokenizer.texts_to_sequences(test_data)
print(test_seq)

[[4, 2, 1, 3], [1, 3, 1]]

{'think': 8, 'amazing': 10, 'my': 1, 'love': 2, 'dog': 3, 'is': 9, 'you': 5, 'do': 7,
'cat': 6, 'i': 4}
```

Passing set of sentences to the ***'text_to_sequences()'*** method converts the sentences to their labelled equivalent based on the corpus of words passed to it.

If the corpus has a word missing that is present in the sentence, the word while being encoded to the label equivalent is omitted and the rest of the words are encoded and printed.

Eg:- In the above **test_data**, the word **'really'** is missing in the corpus. Hence, while encoding, the word **'really'** is omitted and instead the encoded sentence is **'i love my dog'.**

```
from tensorflow.keras.preprocessing.text import Tokenizer

sentences = [
    'I love my dog',
    'I love my cat',
    'You love my dog!',
    'Do you think my dog is amazing?'
]

tokenizer = Tokenizer(num_words = 100, oov_token="<OOV>")
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index

sequences = tokenizer.texts_to_sequences(sentences)

test_data = [
    'i really love my dog',
    'my dog loves my manatee'
]

test_seq = tokenizer.texts_to_sequences(test_data)
print(test_seq)
```

Similarly, for the second sentence, the words **'loves', 'manatee'** is missing in the word corpus. Hence, the encoded sentence is **'my dog my'**.

To overcome the problem faced in the above examples, we can either use a huge corpus of words or use a hyperparameter ***'oov_token'*** and assign it to a certain value which will be used to encode words previously unseen in the corpus. ***'oov_token'*** can be assigned to anything but one should assign a unique value to it so that it isn't confused with an original word.

```
[[5, 1, 3, 2, 4], [2, 4, 1, 2, 1]]



{'think': 9, 'amazing': 11, 'dog': 4, 'do': 8, 'i': 5, 'cat': 7,
 'you': 6, 'love': 3, '<OOV>': 1, 'my': 2, 'is': 10}
```

The output of the above code snippet. Notice that **'<00V>'** is now part of the ***word_index***. Any word not present in the sentences is replaced by the **'<00V>'** encoding.

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

sentences = [
    'I love my dog',
    'I love my cat',
    'You love my dog!',
    'Do you think my dog is amazing?'
]

tokenizer = Tokenizer(num_words = 100, oov_token="<OOV>")
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index

sequences = tokenizer.texts_to_sequences(sentences)

padded = pad_sequences(sequences)
print(word_index)
print(sequences)
print(padded)
```

When feeding training data to the neural network, a uniformity of the data must be maintained. For example, when feeding images for computer vision problems, all images

being fed are of similar dimensions.

Similarly, in NLP, while feeding training data in the form of sentences, padding is used to provide uniformity in the sentences.

```
{'do': 8, 'you': 6, 'love': 3, 'i': 5, 'amazing': 11, 'my': 2, 'is': 10, 'think': 9,
'dog': 4, '<OOV>': 1, 'cat': 7}


[[5, 3, 2, 4], [5, 3, 2, 7], [6, 3, 2, 4], [8, 6, 9, 2, 4, 10, 11]]


[[ 0  0  0  5  3  2  4]
 [ 0  0  0  5  3  2  7]
 [ 0  0  0  6  3  2  4]
 [ 8  6  9  2  4 10 11]]
```

As we can see, padding in the form of **'0'** is generated in the beginning of the sentence. Padding has been done with reference to the longest sentence.

```
padded = pad_sequences(sequences, padding='post',
                                  truncating='post', maxlen=5)
```

If padding is to be done after the sentence, the hyperparameter ***padding*** can be set to ***'post'***. Padding is generally done with reference to the longest sentence, however the hyperparameter ***maxlen*** can be provided to override it and define the maximum length of the sentence. Now, with the knowledge of ***maxlen*** one may wonder if information is lost as only a certain length of the sentence is taken. This is true but one can specify from where the words are omitted. Setting it to ***'post'*** allows one to loose words from the end of the sentence.

Week 2

Word Embedding: Words and associated words are clustered as vectors in a multi-dimensional space. Words are present in a sentence and ***often words with similar meanings are close to each other.***

Eg:- *"The movie was **dull** and **boring**."; "The movie was **fun** and **exciting**."*

Now imagine we pick up a vector in a higher dimensional space, suppose 16 dimensions and words that are found together are given similar vectors. Overtime, words of similar meaning begin to cluster together. The meaning of the words can come from labelling the dataset.

So taking the example of the above sentence, the words ***dull*** and ***boring*** show up a lot in the negative review, therefore they have a similar sentiment and they show up close to each other in a sentence, thus their vectors will be similar. As the neural network trains, it can learn these vectors and associate them with the labels to come up with something called and embedding i.e. the vectors of each word with their associated sentiment.

Similar words show up a lot in a negative review — — -> Similar sentiment

Similar words show up close to each other in a sentence — — → Similar vectors

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Now while building the neural network, we use the Embedding layer which gives an output of the shape of a 2D array with length of the sentence as one dimension and the embedding dimension, in our case 16 as the other dimension.

Therefore, we use the Flatten layer just as we used it in computer vision problems. In CNN based problems, a 2D array of pixels was needed to be flattened to feed it to the neural network. In a NLP based problem the 2D array of Embiddings is needed to be flattened.

Model summary:-

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_9 (Embedding)      (None, 120, 16)           160000
_________________________________________________________________
flatten_3 (Flatten)          (None, 1920)              0
_________________________________________________________________
dense_14 (Dense)             (None, 6)                 11526
_________________________________________________________________
```

```
dense_15 (Dense)             (None, 1)                 7
=================================================================
Total params: 171,533
Trainable params: 171,533
Non-trainable params: 0
```

Alternately, we can use ***GlobalAveragePooling1D*** layer which works in a similar fashion.

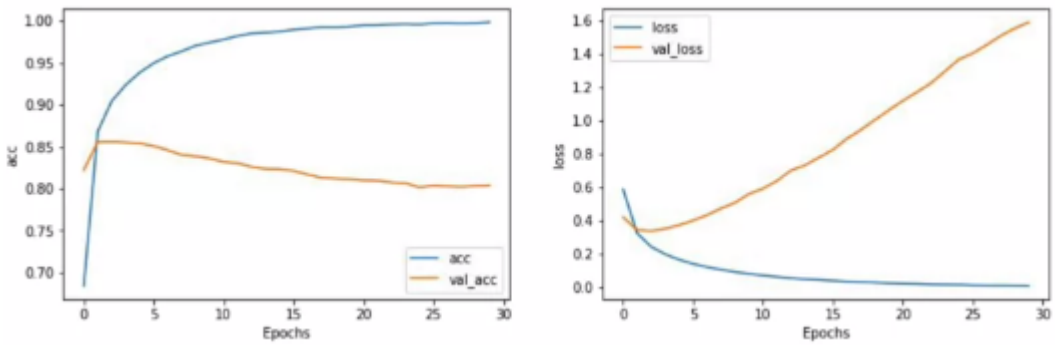
```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

The summary of the model now looks like:-

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_11 (Embedding)     (None, 120, 16)           160000
_________________________________________________________________
global_average_pooling1d_3 ( (None, 16)                0
_________________________________________________________________
dense_16 (Dense)             (None, 6)                 102
_________________________________________________________________
dense_17 (Dense)             (None, 1)                 7
=================================================================
Total params: 160,109
Trainable params: 160,109
Non-trainable params: 0
```

Now the model is simpler and therefore faster. However, upon analysis it was found that the above model even though being faster than the ***Flatten*** model, performed with a slightly lower accuracy.

Loss function

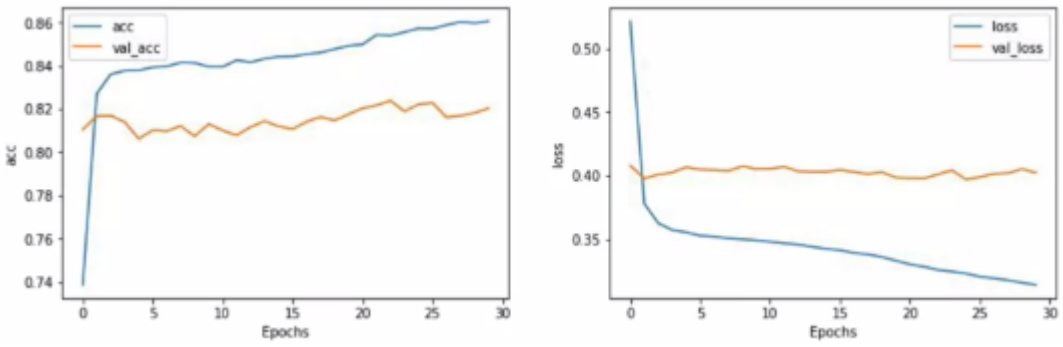


To understand the loss function, we need to treat it in terms of confidence in prediction. So even though the number of accurate predictions increased over time, an increase in the loss implies that the confidence per prediction effectively decreased. One needs to explore the differences in the loss between the training and validation set.

We now try to tweak the hyperparameters

```
vocab_size = 1000    (was 10,000)
embedding_dim = 16
max_length = 16      (was 32)
trunc_type='post'
padding_type='post'
oov_tok = "<OOV>"
training_size = 20000
```

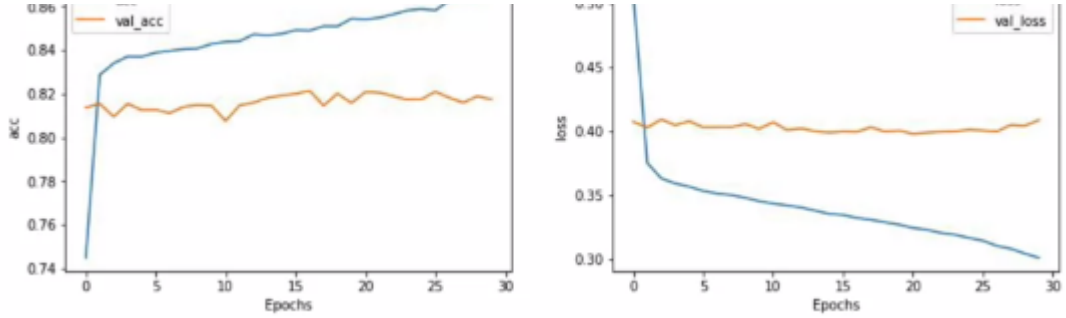
Now we receive the following results:-



We notice that the loss function is flattened out which is better than the previous result but the accuracy is not high.

Another tweak is performed to the hyperparameters where the number of dimensions used in the embedding vector is changed.

```
vocab_size = 1000    (was 10,000)
embedding_dim = 32   (was 16)
max_length = 16      (was 32)
trunc_type='post'
padding_type='post'
oov_tok = "<OOV>"
training_size = 20000
```



The result obtained is not much different to the previous one.

Summarizing the final code:

We first instantiate a tokenizer by providing the ***vocabulary size*** and the ***out of vocab (oov)*** token.

Next we fit the tokenizer on the sentences used for training using the ***fit_on_texts()*** method.

The ***word_index*** allows us to view what the individual words have been numbered or tokenized.

The ***text_to_sequences()*** encodes the entire sentences used for training in the numeric format.

Next we pad the sequences by specifying ***what we are padding, will the padding occur before or after the sentence and maximum length of the sentences being padded.***

```
tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(train_sentences)
word_index = tokenizer.word_index

train_sequences = tokenizer.texts_to_sequences(train_sentences)
train_padded = pad_sequences(train_sequences, padding=padding_type, maxlen=max_length)
```

Similar to encoding the training sentences, we encode the validation sentences and pad them.

```
validation_sequences = tokenizer.texts_to_sequences(validation_sentences)
validation_padded = pad_sequences(validation_sequences, padding = padding_type, maxlen = max_length)
```

Now we create a separate tokenizer called ***label_tokenizer*** to tokenize the labels and fit it on the labels to encode them.

Now we create a numpy array of encoded labels for both the training and validation labels.

```
label_tokenizer = Tokenizer()
label_tokenizer.fit_on_texts(labels)

training_label_seq = np.array(label_tokenizer.texts_to_sequences(train_labels))
validation_label_seq = np.array(label_tokenizer.texts_to_sequences(validation_labels))
```

The model is created and is trained for 30 epochs.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length = max_length),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(32, activation = 'relu'),
    tf.keras.layers.Dense(6, activation = 'softmax')
])
model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
model.summary()
```

```
num_epochs = 30
history = model.fit(train_padded, training_label_seq, epochs=num_epochs,
                    validation_data=(validation_padded, validation_label_seq), verbose=2)
```

Using the code below, one can plot the training set and validation set accuracy and loss.

```
import matplotlib.pyplot as plt


def plot_graphs(history, string):
  plt.plot(history.history[string])
  plt.plot(history.history['val_'+string])
  plt.xlabel("Epochs")
  plt.ylabel(string)
  plt.legend([string, 'val_'+string])
  plt.show()

plot_graphs(history, "acc")
plot_graphs(history, "loss")
```

We reverse the dictionary containing the encoded words using a helper function which facilitates us to plot the embeddings.

```
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])

def decode_sentence(text):
    return ' '.join([reverse_word_index.get(i, '?') for i in text])
```

We create the vector and meta files and store the meta data and vectorised embeddings.

```
import io

out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')
for word_num in range(1, vocab_size):
  word = reverse_word_index[word_num]
  embeddings = weights[word_num]
  out_m.write(word + "\n")
  out_v.write('\t'.join([str(x) for x in embeddings]) + "\n")
out_v.close()
out_m.close()
```

We can upload the vectorised and meta data files in the below mentioned link and view the word embeddings in a higher dimensional space by plotting it.

Embedding projector - visualization of high-dimensional data

Visualize high dimensional data.

projector.tensorflow.org

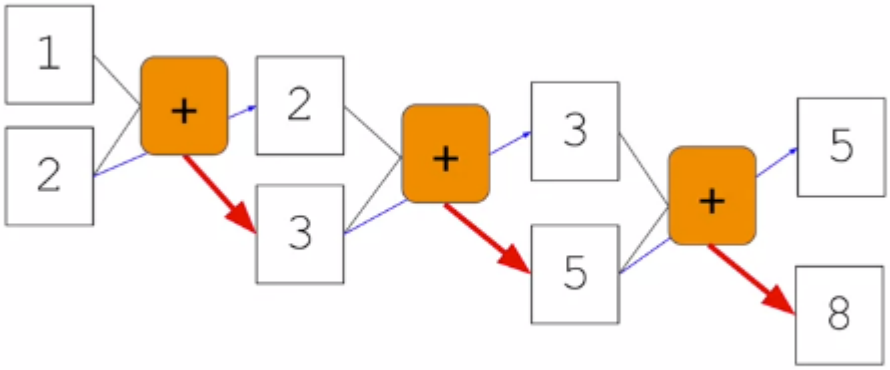
Week 3

In the previous weeks, we tried to implement a classifier that attempted to classify sentences on the basis of text. We attempted to do this by tokenizing the words and

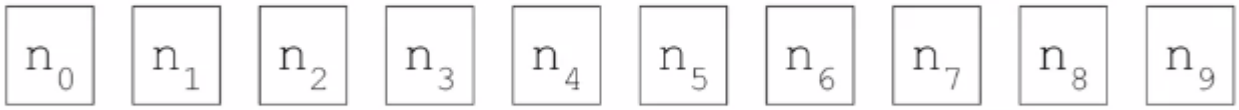
noticed our classifier failed to get any meaningful results. The reason for this was that it was hard to understand the context of the words when it was broken down into subwords. Understanding the sequence in which the subwords occur is necessary to understand their meaning.

Eg:-

The below diagram is the sequence of a fibonacci series.



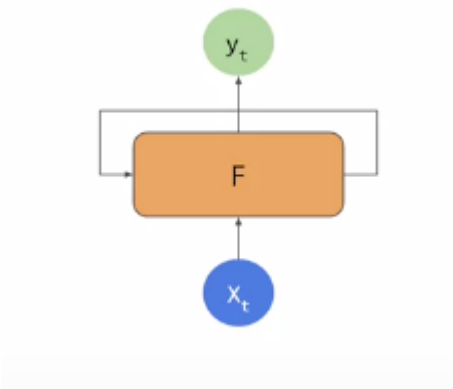
The above sequence works in a recurrence, i.e.



$$n_x = n_{x-1} + n_{x-2}$$

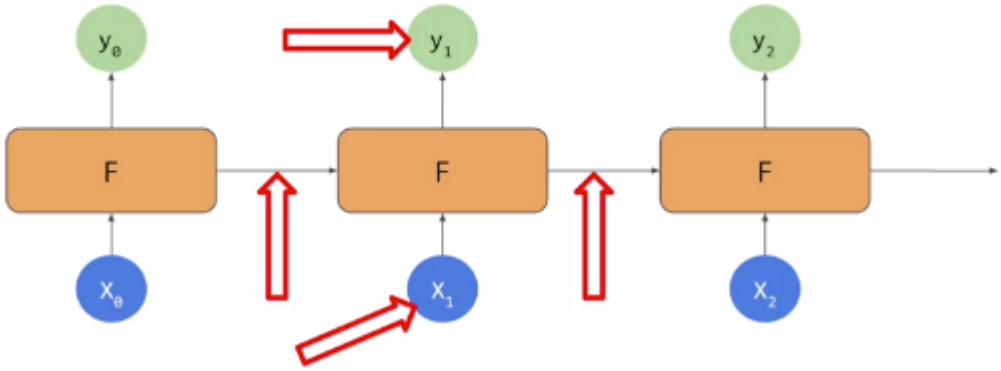
Analyzing the recurrence, one can identify the sequence in which the numbers follow. This sequence is not explicitly mentioned. Data and it's labels are provided and the

sequence is derived through the neural network.



The recurrent function of the fibonacci series can be represented by the above diagram where **Xt** is the initial numbers for the series. The function then outputs **yt,** i.e. the sum of the first two numbers. The sum value is carried to the next iteration where it gets added to the second number and outputs another value. The sequence goes on.

The above recurrent function when unwrapped would look like this:-



X0 -> 1, 2; F -> (1+2); y0 -> 3

X1 -> 2, 3; F -> (2+3); y1 -> 5

X2 -> 3, 5; F -> (3+5); y2 -> 8

We observe that the ***current output is highly dependent on the immediate previous step and is least dependent on the initial steps*** if the series is particularly large, i.e. **y2** is highly dependent on the previous step (**X1, F, y1**) and is less dependent on (**X0**, **F, y0**). Similarly, **y1** is highly dependent on the previous step (**X0**, **F, y0**) and would have been less dependent on the initial steps if it were existing in the series.

This forms the basis of a recurrent neural network (RNN).

This brings up a new challenge when trying to classify text.

Suppose in the below example, we need to predict the word after blue.

Today has a beautiful blue <...>

Today has a beautiful blue sky

When looking at the sentence, we can predict that when talking in context about a beautiful blue something, we quite likely mean “the sky”.

In this case the context word that helps us to predict the next word is very close to the word we are interested in i.e. the word “blue” is next to the word we are interested in “sky”.

However, we may also encounter cases where the *context words* required to predict the *interested word* is present perhaps at the beginning of the sentence. Here the concept of an RNN can fail as it tries to predict the interested word by taking into consideration the words immediately preceding it.

Eg:

I lived in Ireland, so at school they made me learn how to speak <...>

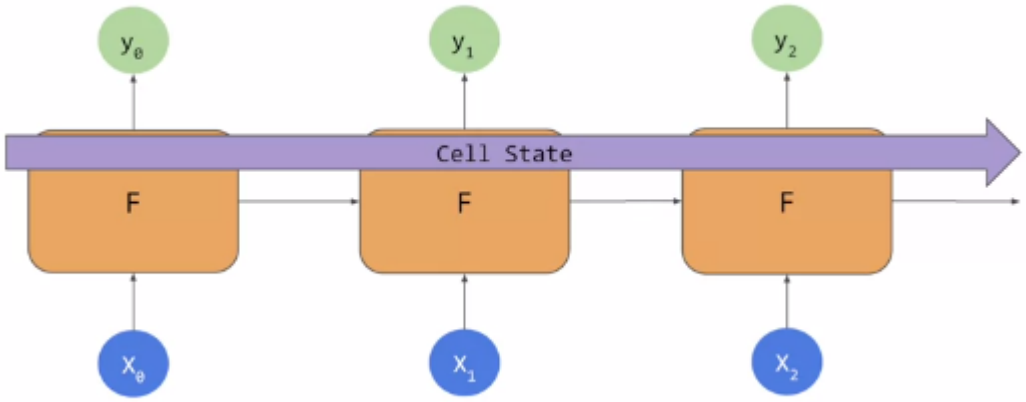
I lived in Ireland, so at school they made me learn how to speak Gaelic

In the above sentence, we can observe that context word “Ireland” appears much earlier in the sentence while the interested word “Gaelic” appears later. An RNN would most

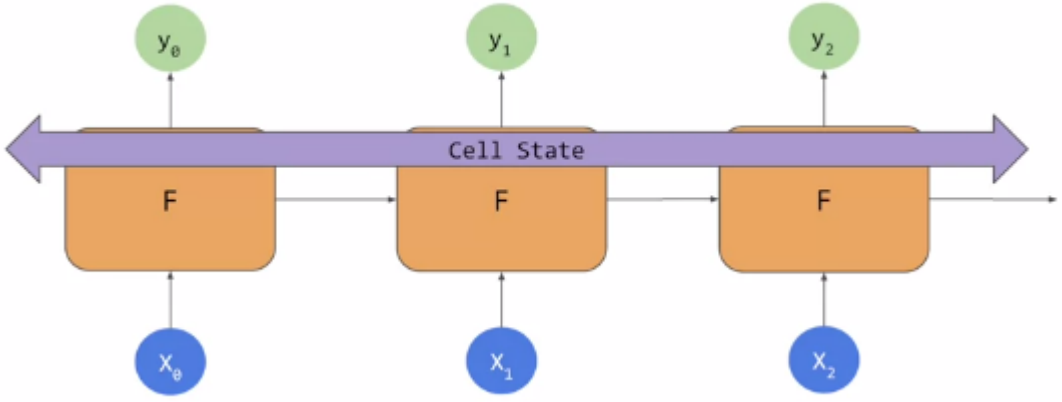
likely attempt to predict the word “Gaelic” by taking into account the words immediately preceding it i.e. “speak”, “to”, “how”; but none of these words would facilitate the prediction of the word “Gaelic”.

In such cases we need a modification of the RNN.

LSTM (Long Short Term Memory)



In these type of networks, in addition to the context being passed like in an RNN, the LSTM have an additional pipeline of context called ***Cell State*** which passes through the network. This helps to keep the context from earlier tokens or steps relevant in later ones to overcome the challenge just discussed in the above example



Cell State can be bidirectional so that tokens appearing later in a sentence can impact the earlier tokens.

Implementing LSTMs in code

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

The LSTM layer is implemented using the above code. The parameter passed is the number of outputs that is desired from that layer. In this case it is **64**.

We wrap the LSTM layer in a bidirectional format, it'll make the ***Cell State*** go in both directions.

The model summary therefore looks like this.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, None, 64)          523840
_________________________________________________________________
bidirectional_1 (Bidirection (None, 128)               66048
_________________________________________________________________
dense_4 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 65
=================================================================
Total params: 598,209
Trainable params: 598,209
Non-trainable params: 0
```

Notice that the output shape of the bidirectional layer is **128** even though we had passed **64** as the parameter. This happens due to the bidirectional layer which effectively doubles the output of the LSTM layer.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(64, activation='relu'),
```

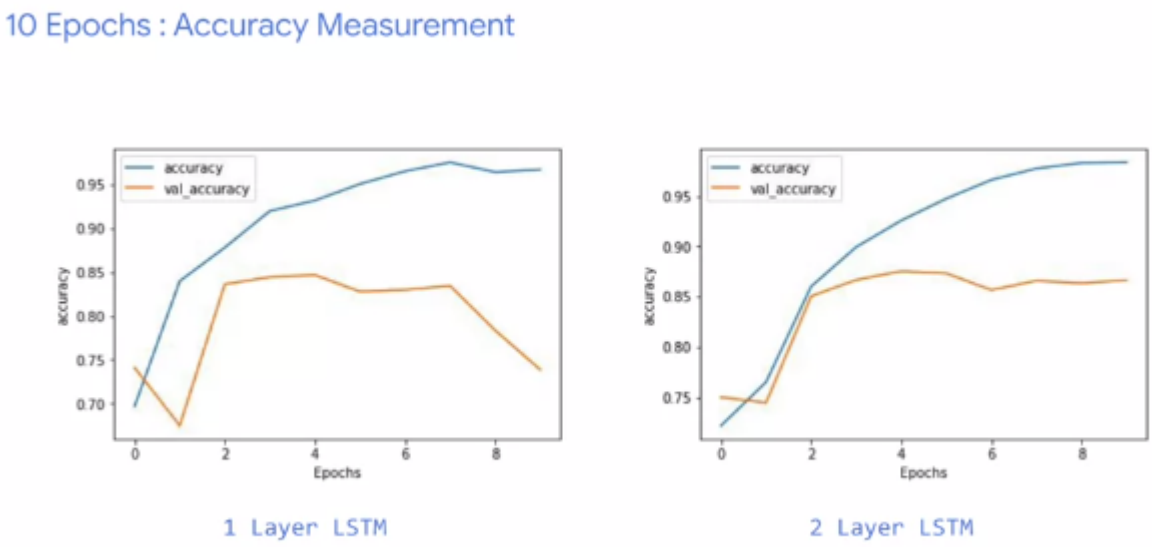
```
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

We can also stack LSTM layers but we need to ensure that the ***return_sequences = True*** is present. This allows us to match the output of the current LSTM layer with the next LSTM layer.

The summary of the model is this:

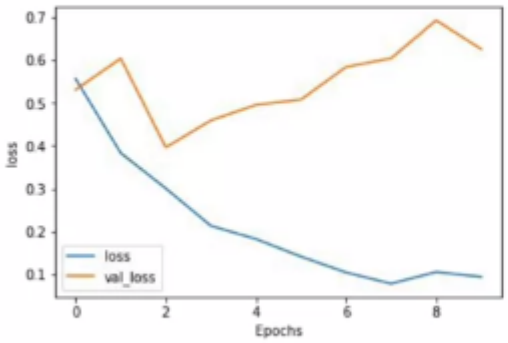
```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 64)          523840
_________________________________________________________________
bidirectional_2 (Bidirection (None, None, 128)         66048
_________________________________________________________________
bidirectional_3 (Bidirection (None, 64)                41216
_________________________________________________________________
dense_6 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 65
=================================================================
Total params: 635,329
Trainable params: 635,329
Non-trainable params: 0
```

Comparing accuracy and loss

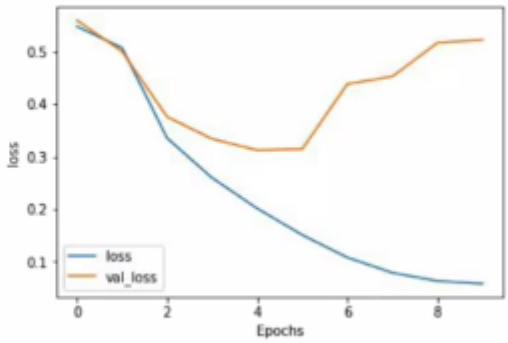


We notice that the 1 layer LSTM network's training accuracy appears to be uneven while the 2 layer LSTM network's training accuracy is much smoother. Often the occurrence of this unevenness is an indication that the model needs improvement.

10 Epochs : Loss Measurement



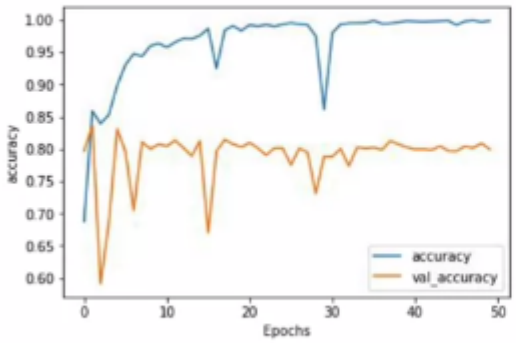
1 Layer LSTM



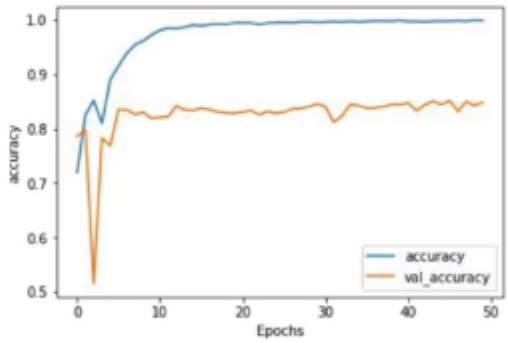
2 Layer LSTM

We notice a similar result while plotting the loss function. The 1 layer LSTM network seems to have a jaggedy surface while that of the 2 layer LSTM network has a smooth surface.

50 Epochs : Accuracy Measurement



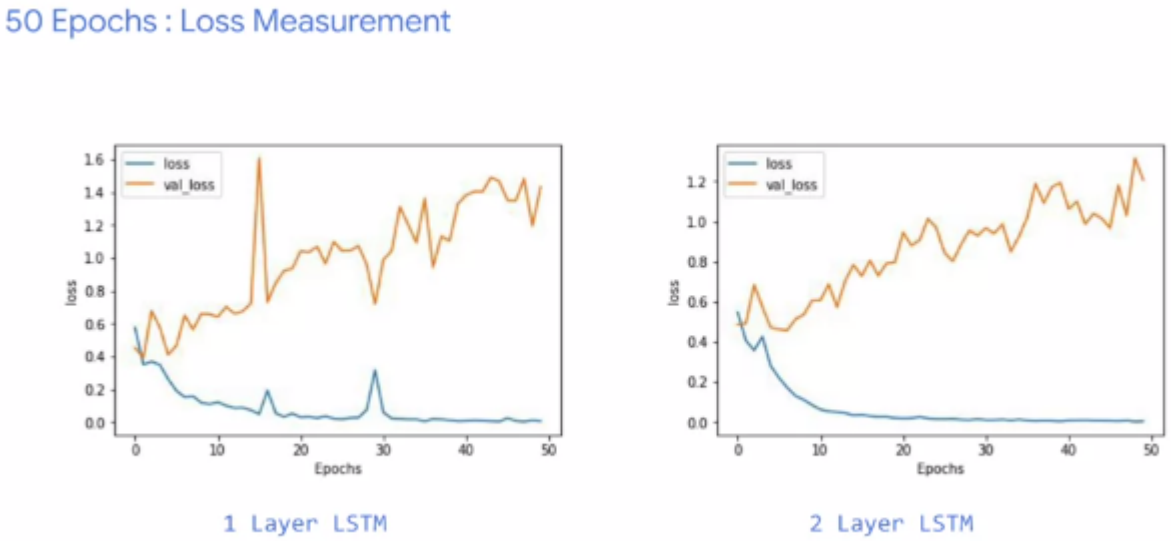
1 Layer LSTM



2 Layer LSTM

When we train the networks for 50 epochs, we notice that the 1 layer LSTM is prone to some pretty sharp dips. Even though the final accuracy is good, the presence of these

dips makes us suspicious of the model. In contrast, the 2 layer LSTM has a smooth curve and achieves a similar result but since it is smooth, the model is much more reliable.



A similar trend can be observed while plotting the loss function. The 2 layer LSTM network's curve appears much smoother than the 1 layer LSTM network. The loss gradually increases in both the curves and is to be monitored closely to check if it flattens out in later epochs as it would be desired.

Comparing Non-LSTMs with LSTMs

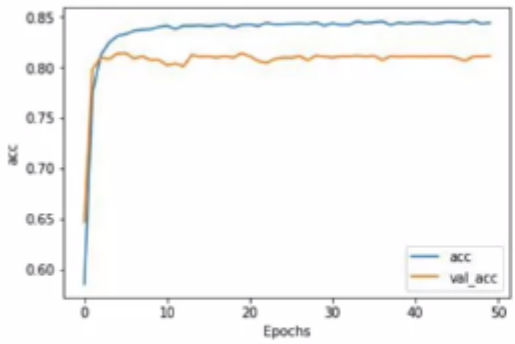
```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              input_length=max_length),
    tf.keras.layers.Flatten(),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

i) Without LSTM

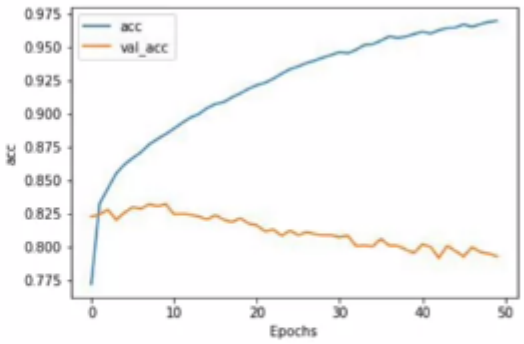
```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              input_length=max_length),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(24, activation='relu'),
```

```
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

ii) With LSTM



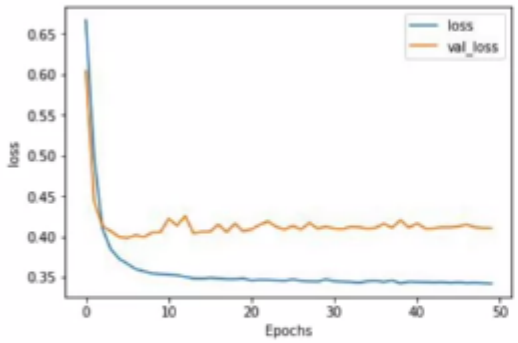
Without LSTM



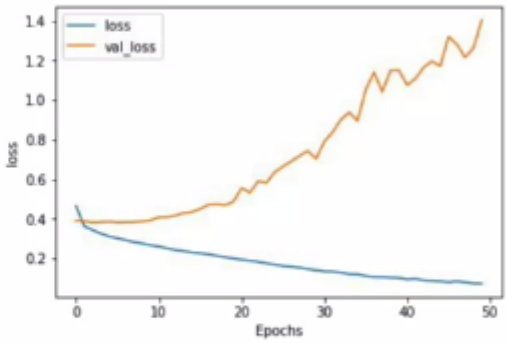
With LSTM

When using a combination of pooling and flattening, we quickly got to an 85% accuracy and then it flattened out. The validation set was a bit less accurate but the curve appears to be in sync with the training accuracy, flattening at an accuracy of 80%.

On the other hand when using a LSTM layer in the network, we quickly got to an accuracy of 85% and it continued to rise up upto an accuracy of 97.5%. The validation set accuracy increased upto 82.5% but then dropped to 80% similar to that of the previous network. The drop in the accuracy hints at some kind of overfitting that must be going on. A little bit of tweaking in the model is required to overcome this issue.



Without LSTM



With LSTM

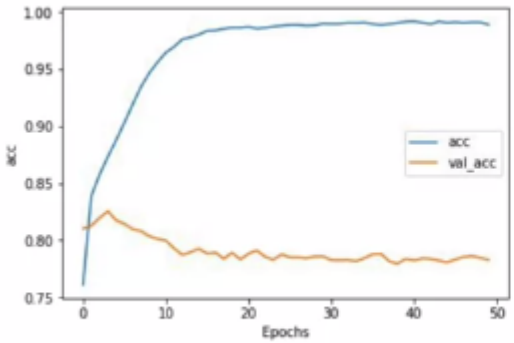
A similar trend is observed when comparing the loss function. The training loss fell quickly and then flattened out. Validation accuracy appears to behave similarly. Whereas in the LSTM network, the training accuracy dropped nicely but the validation accuracy increased thereby hinting at the possible overfitting that might be taking place in the model. Intuitively, this means that while the accuracy of the model increased, the confidence in it decreased.

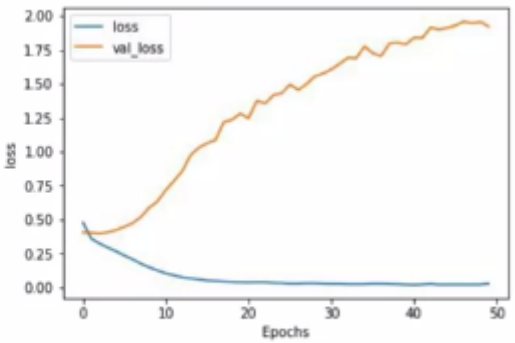
Using Convolutional Layers

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              input_length=max_length),
    tf.keras.layers.Conv1D(128, 5, activation='relu'),
    tf.keras.layers.GlobalMaxPooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

A convolutional layer is applied in the network and now **words will be grouped in the size of the filter i.e. 5** and **convolutions will be learnt that can map the word classification to the desired output**.

Number of filters -> 128; Size of filter -> 5

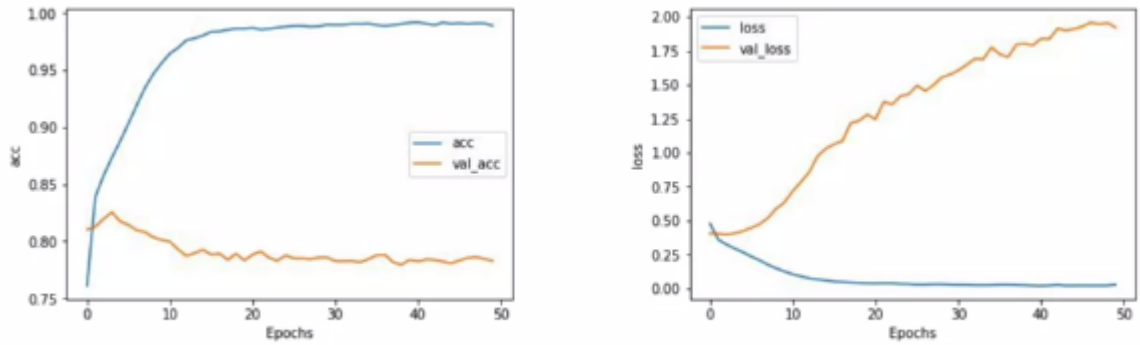




We observe that our model performs even better than the previous one approaching almost a 100% accuracy on the training set and around 80% accuracy on the validation

set but as before our loss increases on the validation set indicating overfitting and consequently drop in the prediction confidence.

If we go back to our model and explore the parameters of the convolutional layer, we'll notice that **we have 128 filters for every group of 5 words**.



On viewing the model summary, we notice that input **length of the sentence is 120** and the **filter size is 5**, consequently **2 words have been shaved off from the front and 2 from the back** leaving us with a **sentence size of 116**. Since we have used a **128 convolutions**, we have the output dimensions of **(116, 128)**.

Week 4

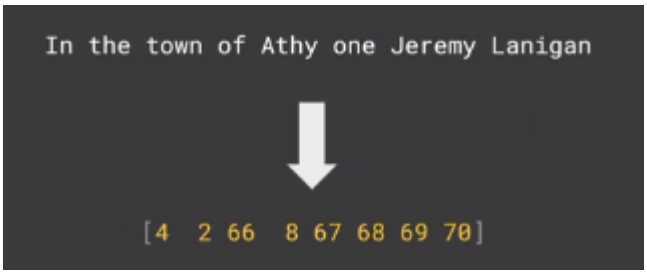
```
tokenizer = Tokenizer()

data="In the town of Athy one Jeremy Lanigan \n Battered away ... ..."
corpus = data.lower().split("\n")

tokenizer.fit_on_texts(corpus)
total_words = len(tokenizer.word_index) + 1
```

The tokenizer is initialized and the data in the form of sentences separated by a **"\n"** is provided. The sentences are then converted to a lower case and individual sentences are retrieved and stored as list items in the ***"corpus"*** list using the split method. The words in the list are tokenized and labeled using the ***fit_on_texts*** method. The total unique words present in the data is stored in the ***total_words*** variable. The extra **1** is due to the existence of the **oov_token**.

```
input_sequences = []
for line in corpus:
```

```
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)
```

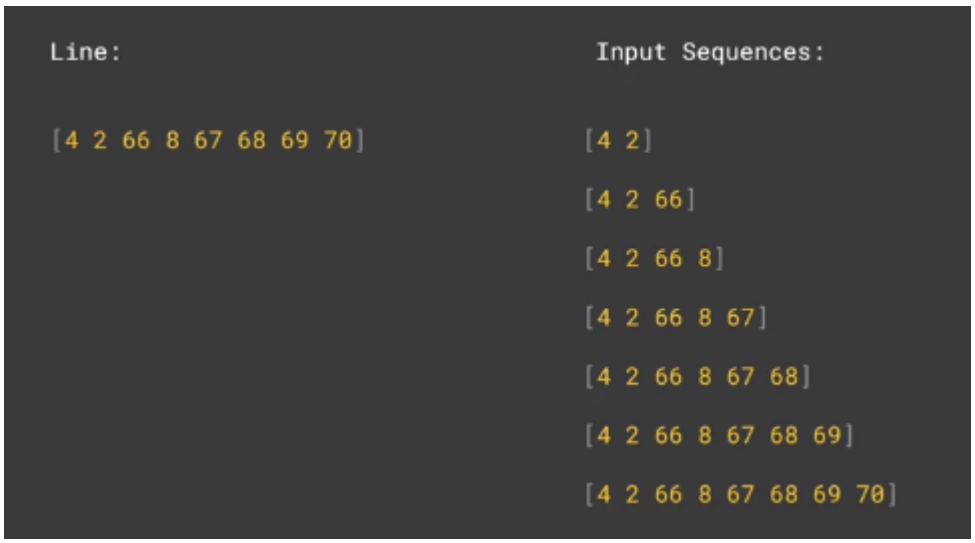
The above highlighted code gives the below output. The first line of texts in the ***corpus*** list is retrieved and is encoded.



The next loop takes the entire encoded sentence and breaks it down into an ***n_gram_sequence*** and appends each sequence into the ***input_sequences*** list.

```
input_sequences = []
for line in corpus:
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)
```

Each row in the "**Input Sequences"** in the image below is an **n_gram_sequence.**



We then try to identify the length of the largest sentence by iterating through all the available **Input Sequences**.

```
max_sequence_len = max([len(x) for x in input_sequences])
```

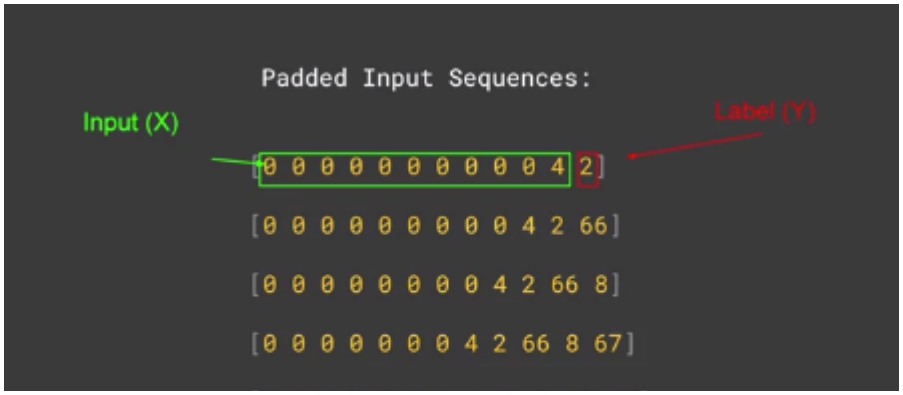
Once the length of the largest sentence is identified, we **pre pad** the **Input Sequences**.

```
input_sequences =
    np.array(pad_sequences(input_sequences, maxlen=max_sequence_len, padding='pre'))
```

The above code gives us the following output.

```
Line:                          Padded Input Sequences:

[4 2 66 8 67 68 69 70]         [0 0 0 0 0 0 0 0 0 0 4 2]
                               [0 0 0 0 0 0 0 0 0 4 2 66]
                               [0 0 0 0 0 0 0 0 4 2 66 8]
                               [0 0 0 0 0 0 0 4 2 66 8 67]
                               [0 0 0 0 0 0 4 2 66 8 67 68]
                               [0 0 0 0 0 4 2 66 8 67 68 69]
                               [0 0 0 0 4 2 66 8 67 68 69 70]
```

The reason we **pre pad** the above **Input Sequences** is so that we can have the entire training data to the left side of each **Input Sequence** and the label representing it to the right. In this case, since we are predicting the word at the end of each sentence, we consider the last word of each **Input Sequence** as the target label that is to be predicted.



```
[0 0 0 0 0 0 4 2 66 8 67 68]

[0 0 0 0 0 4 2 66 8 67 68 69]

[0 0 0 0 4 2 66 8 67 68 69 70]
```

Now we begin to segregate the **Input Sequences** by collecting all the encoded words following upto the last word as the input to the model and the last encoded word as the target label and store both of them in separate lists as shown below.

```
xs = input_sequences[:,:-1]
labels = input_sequences[:,-1]
```

Since this is a multi-class classification model, we **one-hot** encode it using the code below.

```
ys = tf.keras.utils.to_categorical(labels, num_classes=total_words)
```

As stated before, given the encoded sentence, all the encoded words leading upto the second last encoded word is stored as an input **X** for the model. In the case shown below, all encoded words upto **69** is stored as **X** and last encoded word **70** is stored as **Label**.

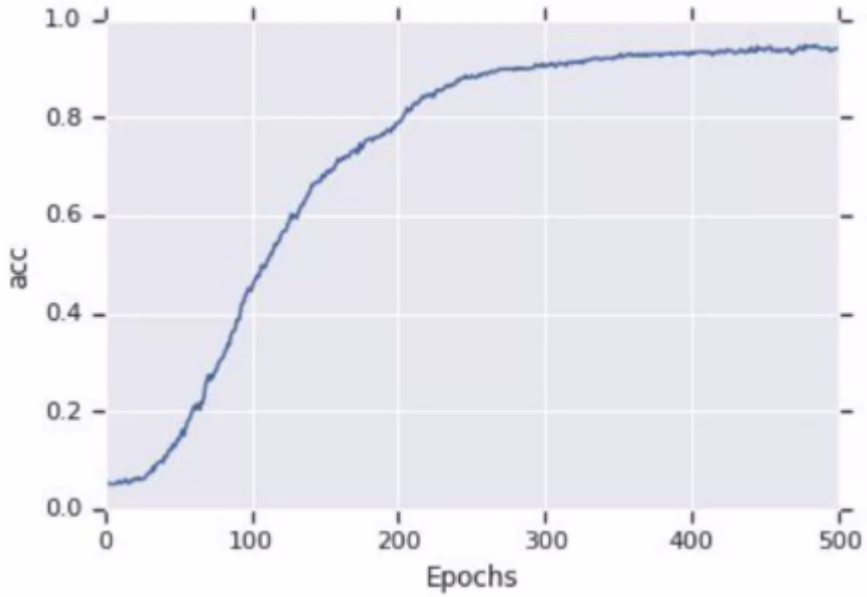
The **one-hot** encoding for the **Label** is shown below as **Y**. The 70th element is stored as **1** because the encoded label corresponds to it, while the rest are **0**.

```
Sentence: [0 0 0 0 4 2 66 8 67 68 69 70]

       X: [0 0 0 0 4 2 66 8 67 68 69]

   Label: [ 70 ]

       Y: [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
            0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
```

The model for the problem is shown below. We use the **adam** optimizer as it happens to perform particularly well in such cases.

```
model = Sequential()
model.add(Embedding(total_words, 64, input_length=max_sequence_len - 1))
model.add((LSTM(20)))
model.add(Dense(total_words, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(xs, ys, epochs=500, verbose=1)
```

The performance accuracy for the above model is plotted below.



We notice that certain words that have been predicted tend to get repeated at the end. This is because the LSTM layer used in the model is uni-directional and the same word once predicted continues to pass forward influencing the words being predicted later in the sentence.

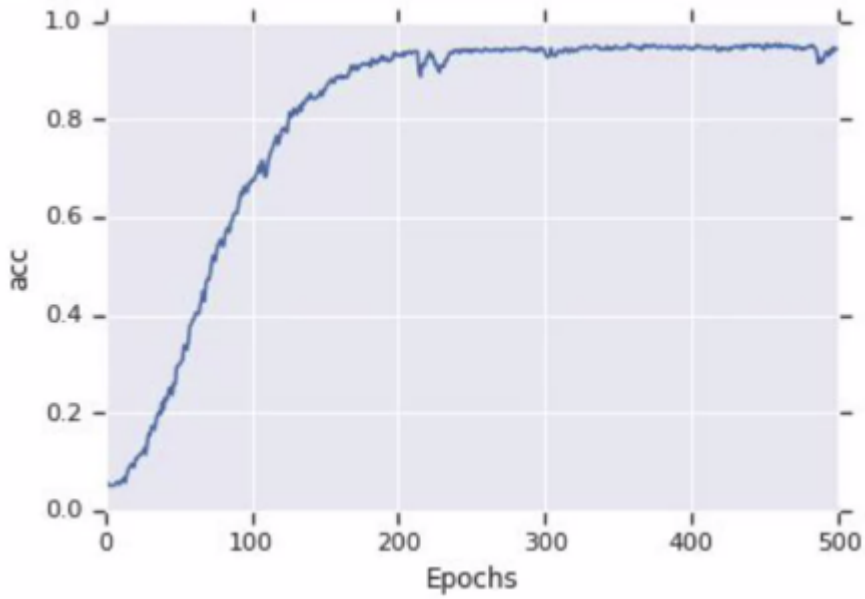
```
Laurence went to dublin round the plenty as red wall me for wall wall
Laurence went to dublin odaly of the nice of lanigans ball ball ball hall
Laurence went to dublin he hadnt a minute both relations hall new relations youd
```

To overcome the above problem, we use a bi-directional LSTM layer so that the words present even after the target word influences the prediction.

```
model = Sequential()
model.add(Embedding(total_words, 64, input_length=max_sequence_len - 1))
model.add(Bidirectional(LSTM(20)))
model.add(Dense(total_words, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
model.fit(xs, ys, epochs=500, verbose=1)
```

The accuracy on including the bi-directional layer is plotted below.



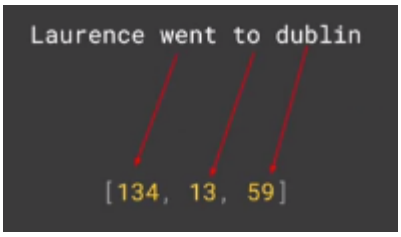
The output on including the bi-directional layer is shown below. We observe that repetitions of words do continue to exist but it's frequency is reduced. That being said, the text below is a part of a poem where words need to rhyme and therefore must follow some sequence. The repetitions therefore may not be due to a fault in the model, rather it may be due to the inherent structure of the sentence.

```
Laurence went to dublin think and wine for lanigans ball entangled in nonsense me
Laurence went to dublin his pipes bellows chanters and all all entangled all kinds
Laurence went to dublin how the room a whirligig ructions long at brooks fainted
```

We now try to generate a poem by giving it an initial set of words.

```
seed_text = "Laurence went to dublin"
next_words = 10

for _ in range(next_words):
  token_list = tokenizer.texts_to_sequences([seed_text])[0]
  token_list = pad_sequences([token_list], maxlen=max_sequence_len - 1, padding='pre')
  predicted = model.predict_classes(token_list, verbose=0)
  output_word = ""
  for word, index in tokenizer.word_index.items():
    if index == predicted:
      output_word = word
      Break
  seed_text += " " + output_word
print(seed_text)
```

Since the word Laurence is not present in the corpus, it's not encoded.



After padding the sentence, we end up with a sequence mentioned below.

```
[  0   0   0   0   0   0   0 134  13  59]
```

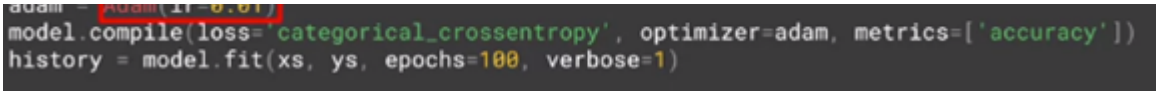
Trying to predict a large number of words following an input sentence is not advisable as the word predicted initially is itself based on some probability. The probability keeps on decreasing and hence the quality of prediction keeps deteriorating as more words are predicted until the predicted words are no longer relevant and gibberish comes out as output.

Solution to the above problem is using a bigger corpus of words.

```
Laurence went to dublin round a cask cask cask cask cask
squeezed forget tea twas make eyes glisten mchugh mchugh
lanigan lanigan glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
```

We are aware of the fact that **adam** optimizer tends to perform well in such problems. To tune it better, we can instantiate the optimizer explicitly and experiment with the learning rate.

```
model = Sequential()
model.add(Embedding(total_words, 100, input_length=max_sequence_len-1))
model.add(Bidirectional(LSTM(150)))
model.add(Dense(total_words, activation='softmax'))
```



Since the English language has more words than the individual alphabets, we can train models by providing them encoded alphabets and train it to predict the next alphabet. This way we don't have to worry about having an extra large corpus of text.

. . .

Thanks for reading this blog. I would appreciate hearing your thoughts on this.

Machine Learning Deep Learning Naturallanguageprocessing TensorFlow Keras

About Help Legal