```
In the town of Athy one Jeremy Lanigan
Battered away til he hadnt a pound.
His father died and made him a man again
Left him a farm and ten acres of ground.

He gave a grand party for friends and relations
Who didnt forget him when come to the wall,
And if youll but listen Ill make your eyes glisten
Of the rows and the ructions of Lanigan's Ball.

Myself to be sure got free invitation,
For all the nice girls and boys I might ask,
And just in a minute both friends and relations
Were dancing round merry as bees round a cask.

Judy ODaly, that nice little milliner,
She tipped me a wink for to give her a call,
And I soon arrived with Peggy McGilligan
Just in time for Lanigans Ball.
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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)
```

```
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)
```

For each line in the corpus, we'll generate a token list using the tokenizers, texts to sequences method.

This will convert a line of text like, "In the he town of Athy one Jeremy Lanigan," into a list of the tokens representing the words.

In the town of Athy one Jeremy Lanigan

[4 2 66 8 67 68 69 70]

```
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)
```

iterate over this list of tokens and create a number of n-grams sequences, namely the first two words in the sentence or one sequence, then the first three are another sequence etc.

Line:

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

Input Sequences:

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

The result of this will be, for the first line in the song, the following input sequences that will be generated. The same process will happen for each line, but as you can see, the input sequences are simply the sentences being broken down into phrases, the first two words, the first three words, etc.

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

We next need to find the length of the longest sentence in the corpus. To do this, we'll iterate over all of the sequences and find the longest one with code like this.

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

Once we have our longest sequence length, the next thing to do is pad all of the sequences so that they are the same length. We will pre-pad with zeros to make it easier to extract the label, you'll see that in a few moments.

Line:

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

So now, our line will be represented by a set of padded input sequences that looks like this

Padded Input Sequences:

```
[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]
```

Padded Input Sequences:

Input (X)

Label (Y)

```
[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]
```

Now, that we have our sequences, the next thing we need to do is turn them into x's and y's, our input values and their labels. When you think about it, now that the sentences are represented in this way, all we have to do is take all but the last character as the x and then use the last character as the y on our label. We do that like this, where for the first sequence, everything up to the four is our input and the two is our label.

By this point, it should be clear why we did pre-padding, because it makes it much easier for us to get the label simply by grabbing

So now, we have to split our sequences into our x's and our y's.

To do this, let's grab the first n tokens, and make them our x's. We'll then get the last token and make it our label. Before the label becomes a y, there's one more step, and you'll see that shortly. Python makes this really easy to do with it's less syntax.

So to get my x's, I just get all of the input sequences sliced to remove the last token. To get the labels, I get all of the input sequence sliced to keep the last token

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

Now, I should one-hot encode my labels as this really is a classification problem. Where given a sequence of words, I can classify from the corpus, what the next word would likely be. So to one-hot encode, I can use the contrast utility to convert a list to a categorical. I simply give it the list of labels and the number of classes which is my number of words, and it will create a one-hot encoding of the labels.

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

```
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 ]
```

So for example, if we consider this list of tokens as a sentence, then the x is the list up to the last value, and the label is the last value which in this case is 70. The y is a one-hot encoded array whether length is the size of the corpus of words and the value that is set to one is the one at the index of the label which in this case is the 70th element.

```
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.
```

We'll start with an embedding layer. We'll want it to handle all of our words, so we set that in the first parameter. The second parameter is the number of dimensions to use to plot the vector for a word. Feel free to tweak this to see what its impact would be on results, but I'm going to keep it at 64 for now. Finally, the size of the input dimensions will be fed in, and this is the length of the longest sequence minus 1. We subtract one because we cropped off the last word of each sequence to get the label, so our sequences will be one less than the maximum sequence length.

```
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)
```

```
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)
```

```
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)
```

Finally there's a dense layer sized as the total words, which is the same size that we used for the one-hot encoding. Thus this layer will have one neuron, per word and that neuron should light up when we predict a given

```
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)
```

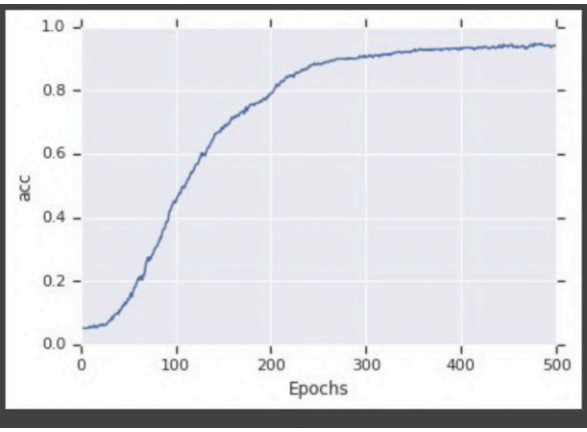
We're doing a categorical classification, so we'll set the laws to be categorical cross entropy.

```
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)
```

we'll use the atom optimizer, which seems to work particularly well for tasks like this one

```
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)
```

we'll train for a lot of epoch, say about 500, as it takes a while for a model like this to converge, particularly as it has very little data. So if we train the model for 500 epochs, it will look like this.



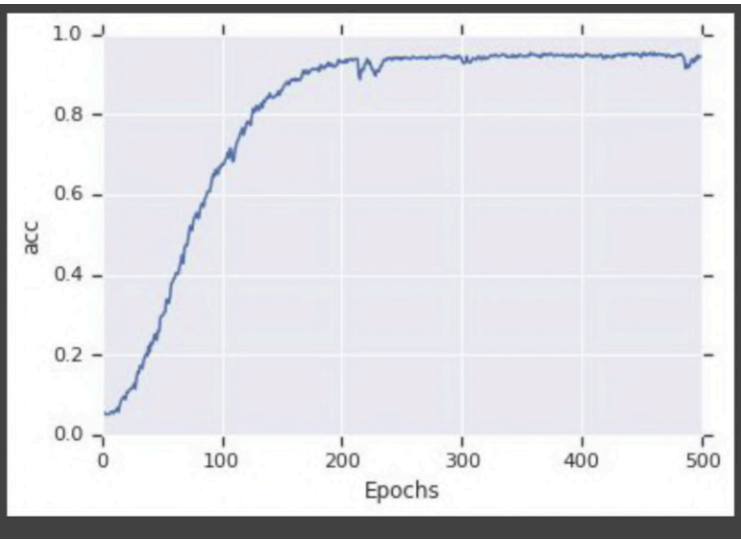
```
Laurence went to dublin round the plenty as red wall me for wall wall
Laurence went to dublin odaly of the nice of lanigans ball ball ball hall
Laurence went to dublin he hadnt a minute both relations hall new relations youd
```

```
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
```

```
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
```

Notice that there's a lot of repetition of words.

```
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)
```



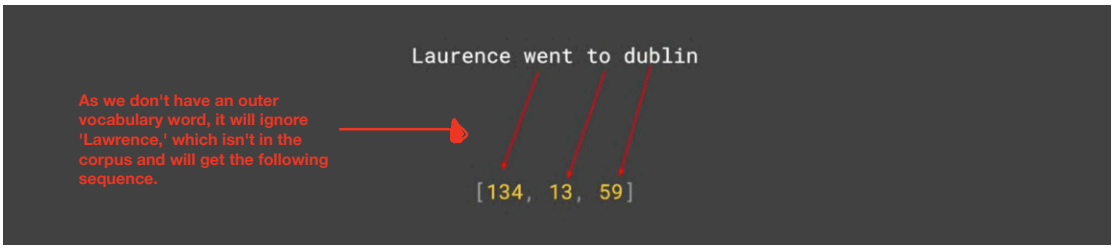
This is because our LSTM was only carrying context forward. Let's take a look at what happens if we change the code to be bidirectional. By adding this line simply defining the LSTM is bidirectional, and then retraining

I can see that I do converge a bit quicker as you'll see in this chart

```
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
```

```
Laurence went to dublin
```

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




```
token_list = pad_sequences([token_list], maxlen=max_sequence_len - 1, padding='pre')
```

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

This code will then pad the sequence so it matches the ones in the training set.

```
predicted = model.predict_classes(token_list, verbose=0)
```

So we end up with something like this which we can pass to the model to get a prediction back.

This will give us the token of the word most likely to be the next one in the sequence

```
for word, index in tokenizer.word_index.items():
    if index == predicted:
        output_word = word
        break
seed_text += " " + output_word
```

Now, we can do a reverse lookup on the word index items to turn the token back into a word and to add that to our seed texts

```
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)
```

Here's the complete code to do that 10 times and you can tweak it for more. But do you know that the more words you predict, the more likely you are going to get gibberish? Because each word is predicted, so it's not 100 per cent certain, and then the next one is less certain, and the next one, etc

```
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
```

```
data = open('/tmp/irish-lyrics-eof.txt').read()
```

```
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'))
adam = Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=adam, metrics=['accuracy'])
history = model.fit(xs, ys, epochs=100, verbose=1)
```

```
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'))
adam = Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=adam, metrics=['accuracy'])
history = model.fit(xs, ys, epochs=100, verbose=1)
```

```
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'))
adam = Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=adam, metrics=['accuracy'])
history = model.fit(xs, ys, epochs=100, verbose=1)
```

```
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'))
adam = Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=adam, metrics=['accuracy'])
history = model.fit(xs, ys, epochs=100, verbose=1)
```

https://www.tensorflow.org/tutorials/sequences/text_generation