

4.1 Introduction

We've seen classification of text over the last few lessons. But what about if we want to generate new text. Now this might sound like new unbroken ground, but when you think about it, you've actually covered everything that you need to do this already. Instead of generating new text, how about thinking about it as **a prediction problem**. Remember when for example you had a bunch of pixels for a picture, and you trained a neural network to classify what those pixels were, and it would predict the contents of the image, like maybe a fashion item, or a piece of handwriting. Well, text prediction is very similar. We can get a body of texts, extract the full vocabulary from it, and then create datasets from that, where we make it phrase the Xs and the next word in that phrase to be the Ys. For example, consider the phrase, Twinkle, Twinkle, Little, Star. What if we were to create training data where the Xs are Twinkle, Twinkle, Little, and the Y is star. Then, whenever neural network sees the words Twinkle, Twinkle, Little, the predicted next word would be star. Thus given enough words in a corpus with a neural network trained on each of the phrases in that corpus, and the predicted next word, we can come up with some pretty sophisticated text generation and this week, you'll look at coding that.

4.2 Looking into the code

```
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.
```



So let's start with a simple example. I've taken a traditional Irish song and here's the first few words of it, and here's the beginning of the code to process it.

```
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
```

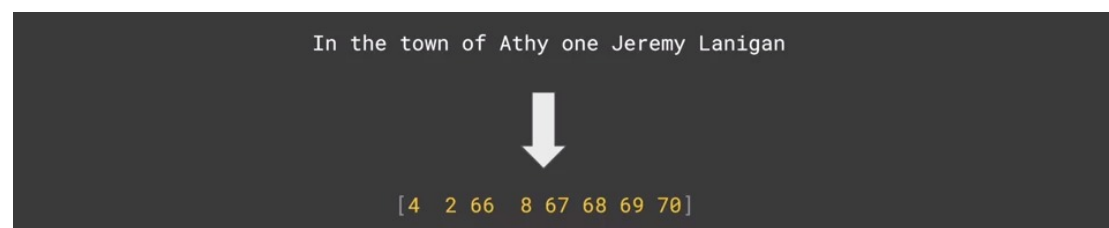
In this case to keep things simple, I put the entire song into a single string. You can see that string here and I've denoted line breaks with `\n`. Then, by calling the `split` function on `\n`, I can create a Python list of sentences from the data and I'll convert all of that to lowercase. Using the tokenizer, I can then call `fit_on_texts` to this corpus of work and it will create the dictionary of words and the overall corpus. This is a key value pair with the key being the word and the value being the token for that word. We can find the total number of words in the corpus, by getting the length of its word index. We'll add one to this, to consider outer vocabulary words.

4.3 Training the data

```
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)
```

So now, let's look at the code to take this corpus and turn it into training data. Here's the beginning, I will unpack this line by line.

First of all, our training x's will be called input sequences, and this will be a Python list.



Then 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 town of Athy one Jeremy Lanigan," into a list of the tokens representing the words.

Line:	Input Sequences:
[4 2 66 8 67 68 69 70]	[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]

Then we'll 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. 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:	Padded Input Sequences:
[4 2 66 8 67 68 69 70]	[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]

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

Line:	Padded Input Sequences:
[4 2 66 8 67 68 69 70]	[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]

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. Similarly, here for the second sequence where the input is two words and the label is the third word, tokenized to 66. Here, the input is three words and the label is eight, which was the fourth word in the sentence. 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 the last token.

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

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.

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

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

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.

[illegible][illegible][illegible][illegible][illegible]

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 where 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. Okay. You now have all of the data ready to train a network for prediction. Hopefully, this was useful for you. You'll see the neural network in the next video. But first, let's see your screen cast of processing the data, using the methods that you saw in this lesson.

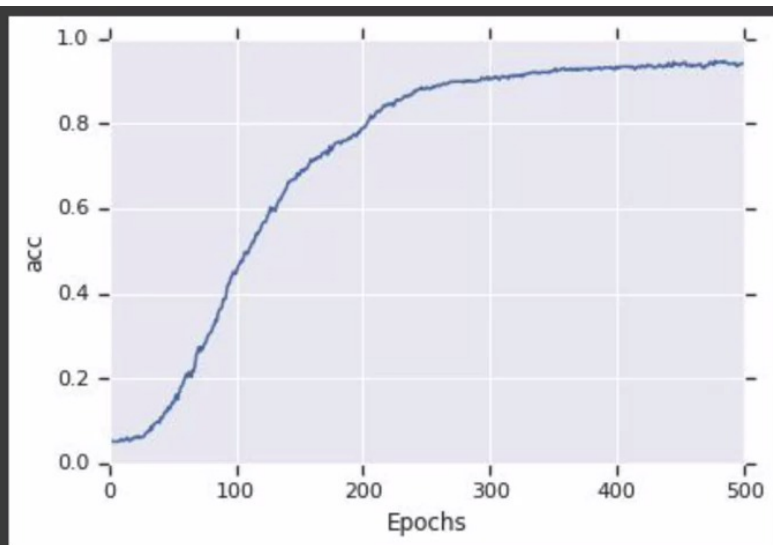
4.4 Notebook for lesson 1

4.5 Finding what the next word should be

In the previous video we looked at the data, a string containing a single song, and saw how to prepare that for generating new text. We saw how to tokenize the data and then create sub-sentence engrams that were labelled with the next word in the sentence. We then one-hot encoded the labels to get us into a position where we can build a neural network that can, given a sentence, predict the next word. Now that we have our data as xs and ys, it's relatively simple for us to create a neural network to classify what the next word should be, given a set of words.

```
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)
```

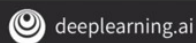
Here's the code. We'll start with an embedding layer. We'll want it to handle all of our words, so we set that in the first parameter. The second parameter is the number of dimensions to use to plot the vector for a word. Feel free to tweak this to see what its impact would be on results, but I'm going 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. Next we'll add an LSTM. As we saw with LSTMs earlier in the course, their cell state means that they carry context along with them, so it's not just next door neighbor words that have an impact. I'll specify 20 units here, but again, you should feel free to experiment. 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 word. We're doing a categorical classification, so we'll set the laws to be categorical cross entropy. And we'll use the atom optimizer, which seems to work particularly well for tasks like this one. Finally, 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.

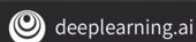
4.6 Example

```
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
```



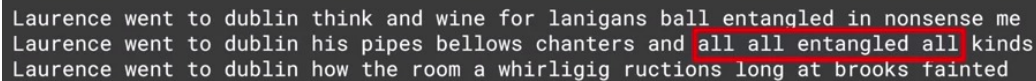
Here are a few phrases that were generated when I gave the neural network the sentence Lawrence went to Dublin, and I asked it to predict the next 10 words. We'll see the code for that in a moment, but notice that there's a lot of repetition of words. In this sentence, three of the last five words are wall, and here three of the last four are ball, and even in this one the word relations gets repeated. This is because our LSTM was only carrying context forward.

```
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)
```

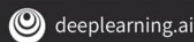


Let's take a look at what happens if we change the code to be bidirectional. By adding this line

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 chanterers and all all entangled all kinds
Laurence went to dublin how the room a whirligig ructions long at brooks fainted
```



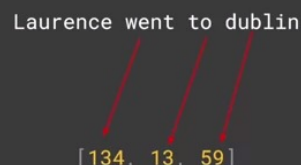
After training and testing, I now get these sentences. They make a little bit more sense, but there's still some repetition. That being said, remember this is a song where words rhyme such as ball, all and wall, et cetera, and as such many of them are going to show up.

4.7 Predicting a word

So now, let's take a look at how to get a prediction for a word and how to generate new text based on those predictions. So let's start with a single sentence. For example, '**Lawrence went to Dublin.**' I'm calling this sentence the seed.

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

If I want to predict the next 10 words in the sentence to follow this, then this code will use the tokenizer that for me using the text to sequences method on the tokenizer.



```
Laurence went to dublin
           /  /  /
          [134, 13, 59]
```

As we don't have an outer vocabulary word, it will ignore 'Lawrence,' which isn't in the corpus and will get the following sequence. This code will then pad the sequence so it matches the ones in the training set.

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

```
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
```

So 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, and that's it.

```
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
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
```

So for example, if you try the same seed and predict 100 words, you'll end up with something like this. Using a larger corpus we'll help, and then the next video, you'll see the impact of that, as well as some tweaks that a neural network that will help you create poetry.

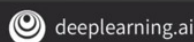
4.8 Poetry

In the previous video, we used the single song to generate text. We got some text from it but once we tried to predict beyond a few words, it rapidly became gibberish. So in this video, we'll take a look at adapting that work for a larger body of words to see the impact. The good news is that it will require very little code changes, so you'll be able to get it working quite

quickly. I've prepared a file with a lot of songs that has 1,692 sentences in all to see what the impact would be on the poetry that a neural network would create. To download these lyrics, you can use this code.

4.9 Looking into the code

```
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)
```



Now instead of hard-coding the song into a string called data, I can read it from the file like this. I've updated the model a little bit to make it work better with a larger corpus of work but please feel free to experiment with these hyper-parameters. Three things that you can experiment with. First, is the dimensionality of the embedding, 100 is purely arbitrary and I'd love to hear what type of results you will get with different values.

Similarly, I increase the number of LSTN units to 150. Again, you can try different values or you can see how it behaves if you remove the bidirectional. Perhaps you want words only to have forward meaning, where big dog makes sense but dog big doesn't make so much sense. Perhaps the biggest impact is on the optimizer. Instead of just hard coding Adam as my optimizer this time and getting the defaults, I've now created my own Adam optimizer and set the learning rate on it. Try experimenting with different values here and see the impact that they have on convergence. In particular, see how different convergences can create different poetry. And of course, training for different epochs will always have an impact with more generally being better but eventually you'll hit the law of diminishing returns.

4.10 Laurence the port

In a co-lab with this data and these parameters, using a GPU, it typically takes about 20 minutes to train a model. Once it's done, try a a seed sentence and get it to give you 100 words. Note that there are no line breaks in the prediction, so you'll have to add them manually to turn the word stream into poetry. Here's a simple example. I used a famous

quote from a very famous movie and let's see if you can recognize it. And I tried that to see what type of poem it would give me and I got this.

```
Help Me Obi-Wan Kenobi, you're my only hope  
my dear  
and hope as i did fly with its flavours  
along with all its joys  
but sure i will build  
love you still  
gold it did join  
do mans run away cross our country  
are wedding i was down to  
off holyhead wished meself  
down among the pigs  
played some hearty rigs  
me embarrass  
find me brother  
me chamber she gave me  
who storied be irishmen  
to greet you  
lovely molly  
gone away from me home  
home to leave the old tin cans  
the foemans chain one was shining  
sky above i think i love
```

And it almost make sense, [LAUGH] but that's poetry for you. Help me Obi-Wan Kenobi, you're my only hope. My dear and hope as I did fly with its flavors, along with all its joys. But sure I will build, love you still, gold it did join. Do mans run away cross our country, our wedding I was down to. Off holyhead wished myself down among the pigs. Played some hearty Riggs, me embarrass. Find me brother, me chambers she gave me her story. Be Irishmen to greet you lovely Molly. Gone away from me home, home to leave the old tin cans. The foreman's chain once was shining. Sky above I think I love.

4.11 You next task

Now, this approach works very well until you have very large bodies of text with many many words. So for example, you could try the complete works of Shakespeare and you'll likely hit memory errors, as assigning the one-hot encodings of the labels to matrices that have over 31,477 elements, which is the number of unique words in the collection, and there are over 15 million sequences generated using the algorithm that we showed here. So the labels alone would require the storage of many terabytes of RAM. So for your next task, you'll go through a workbook by yourself that uses character-based prediction. The full number of unique characters in a corpus is far less than the full number of unique words, at least in English. So the same principles that you use to predict words can be used to apply here. The workbook is at this URL, so try it out, and once you've done, that you'll be ready for this week's final exercise.