**Part 1**

LAURENCE MORONEY: Hi, and welcome to this series on Zero to Hero for natural language processing using TensorFlow.

If you're not an expert on AI or ML, don't worry.

We're taking the concepts of NLP and teaching them from first principles.

In this first lesson, we'll talk about how to represent words in a way that a computer can process them, with a view to later training a neural network that can understand their meaning.

This process is called tokenization.

So let's take a look.

Consider the word "listen," as you can see here.

It's made up of a sequence of letters.

These letters can be represented by numbers using an encoding scheme.

A popular one called ASCII has these letters represented by these numbers.

This bunch of numbers can then represent the word listen.

But the word silent has the same letters, and thus the same numbers, just in a different order.

So it makes it hard for us to understand sentiment of a word just by the letters in it.

So it might be easier, instead of encoding letters, to encode words.

Consider the sentence I love my dog.

So what would happen if we start encoding the words in this sentence instead of the letters in each word? So, for example, the word "I" could be one, and then the sentence "I love my dog" could be 1, 2, 3, 4.

Now, if I take another sentence, for example, "I love my cat," how would we encode it? Now we see "I love my" has already been given 1, 2, 3, so all I need to do is encode "cat." I'll give that the number 5.

And now, if we look at the two sentences, they are 1, 2, 3, 4 and 1, 2, 3, 5, which already show some form of similarity between them.

And it's a similarity you would expect, because they're both about loving a pet.

Given this method of encoding sentences into numbers, now let's take a look at some code to achieve this for us.

This process, as I mentioned before, is called tokenization, and there's an API for that.

We'll look at how to use it with Python.

So here's your first look at some code to tokenize these sentences.

Let's go through it line by line.

First of all, we'll need the tokenize our APIs, and we can get these from TensorFlow Keras like this.

We can represent our sentences as a Python array of strings like this.

It's simply the "I love my dog" and "I love my cat" that we saw earlier.

Now the fun begins.

I can create an instance of a tokenizer object.

The num\_words parameter is the maximum number of words to keep.

So instead of, for example, just these two sentences, imagine if we had hundreds of books to tokenize, but we just want the most frequent 100 words in all of that.

This would automatically do that for us when we do the next step, and that's to tell the tokenizer to go through all the text and then fit itself to them like this.

The full list of words is available as the tokenizer's word index property.

So we can take a look at it like this and then simply print it out.

The result will be this dictionary showing the key being the word and the value being the token for that word.

So for example, my has a value of 3.

The tokenizer is also smart enough to catch some exceptions.

So for example, if we updated our sentences to this by adding a third sentence, noting that "dog" here is followed by an exclamation mark, the nice thing is that the tokenizer is smart enough to spot this and not create a new token.

It's just "dog." And you can see the results here.

There's no token for "dog exclamation," but there is one for "dog." And there is also a new token for the word "you." If you want to try this out for yourself, I've put the code in the Colab here.

Take it for a spin and experiment.

You've now seen how words can be tokenized, and the tools in TensorFlow that handle that tokenization for you.

Now that your words are represented by numbers like this, you'll next need to represent your sentences by sequences of numbers in the correct order.

You'll then have data ready for processing by a neural network to understand or maybe even generate new text.

You'll see the tools that you can use to manage this sequencing in the next episode, so don't forget to hit that subscribe button.

**Part 2**

[MUSIC PLAYING] LAURENCE MORONEY: Welcome to episode 2 of this series of Zero to Hero with Natural Language Processing.

In the last video, you learned about how to tokenize words using TensorFlow's tools.

In this one, you'll take that to the next step, creating sequences of numbers from your sentences and using tools to process them to make them ready for teaching neural networks.

Last time, we saw how to take a set of sentences and use the tokenizer to turn the words into numeric tokens.

Let's build on that now by also seeing how the sentences containing those words can be turned into sequences of numbers.

We'll add another sentence to our set of texts, and I'm doing this because the existing sentences all have four words, and it's important to see how to manage sentences, or sequences, of different lengths.

The tokenizer supports a method called texts to sequences which performs most of the work for you.

It creates sequences of tokens representing each sentence.

Let's take a look at the results.

At the top, you can see the list of word-value pairs for the tokens.

At the bottom, you can see that the sequences that texts to sequences has returned.

We have a few new words such as amazing, think, is, and do, and that's why this index looks a little different than before.

And now, we have the sequences.

So for example, the first sequence is 4, 2, 1, 3, and these are the tokens for I, love, my, and dog in that order.

So now, we have the basic tokenization done, but there's a catch.

This is all very well for getting data ready for training a neural network, but what happens when that neural network needs to classify texts, but there are words in the text that it has never seen before? This can confuse the tokenizer, so we'll look at how to handle that next.

Let's now look back at the code.

I have a set of sentences that I'll use for training a neural network.

The tokenizer gets the word index from these and create sequences for me.

So now, if I want to sequence these sentences, containing words like manatee that aren't present in the word index, because they weren't in my initial set of data, what's going to happen? Well, let's use the tokenizer to sequence them and print out the results.

We see this, I really love my dog.

A five-word sentence ends up as 4, 2, 1, 3, a four-word sequence.

Why? Because the word really wasn't in the word index.

The corpus used to build it didn't contain that word.

And my dog loves my manatee ends up as 1, 3, 1, which is my, dog, my, because loves and manatee aren't in the word index.

So as you can imagine, you'll need a really big word index to handle sentences that are not in the training set.

But in order not to lose the length of the sequence, there is also a little trick that you can use.

Let's take a look at that.

By using the OOV token property, and setting it as something that you would not expect to see in the corpus, like angle bracket, OOV, angle bracket, the tokenizer will create a token for that, and then replace words that it does not recognize with the Out Of Vocabulary token instead.

It's simple, but effective, as you can see here.

Now, the earlier sentences are encoded like this.

We've still lost some meaning, but a lot less.

And the sentences are at least the correct length.

That's a handy little trick, right? And while it helps maintain the sequence length to be the same length as the sentence, you might wonder, when it comes to needing to train a neural network, how it can handle sentences of different lengths? With images, they're all usually the same size.

So how would we solve that problem? The advanced answer is to use something called a RaggedTensor.

That's a little bit beyond the scope of this series, so we'll look at a different and simpler solution, padding.

OK.

So here's the code that we've been using, but I've added a couple of things.

First is to import pad sequences from pre-processing.

As its name suggests, you can use it to pad our sequences.

Now, if I want to pad my sequences, all I have to do is pass them to pad sequences, and the rest is done for me.

You can see the results of our sentences here.

First is the word index, and then is the initial set of sequences.

The padded sequence is next.

So for example, our first sentence is 5, 3, 2, 4.

And in the padded sequence, we can see that there are three 0s preceding it.

Well, why is that? Well, it's because our longest sentence had seven words in it.

So when we pass this corpus to pad sequence, it measured that and ensured that all of the sentences would have equally-sized sequences by padding them with 0s at the front.

Note that OOV isn't 0.

It's 1.

0 means padding.

Now, you might think that you don't want the 0s in front, and you might want them after the sentence.

Well, that's easy.

You just set the padding parameter to post like this, and that's what you'll get.

Or if you don't want the length of the padded sentences to be the same as the longest sentence, you can then specify the desired length with the maxlen parameter like this.

But wait, you might ask what happens if sentences are longer than the specified maxlen? Well, then, you can specify how to truncate either chopping off the words at the end, with a post truncation, or from the beginning with a pre-truncation.

And here's what a post looks like.

But don't take my word for it.

Check out the Codelab at this URL, and you can try out all of the code in this video for yourself.

Now that you've seen how to tokenize your text and organize it into sequences, in the next video, we'll take that data and train a neural network with text data.

We'll look at a data set with sentences that are classified as sarcastic and not sarcastic, and we'll use that to determine if sentences contain sarcasm.

Really? No, no.

I mean, really.

**Part 3**

[MUSIC PLAYING] LAURENCE MORONEY: Hi, and welcome back to this series on Zero to Hero with TensorFlow, where we're looking at Natural Language Processing.

In the last couple of episodes, you saw how to tokenize text into numeric values, and how to use tools in TensorFlow to regularize and pad that text.

Now that we've gotten the preprocessing out of the way, we can next look at how to build a classifier to recognize sentiment in text.

We'll start by using a dataset of headlines, where the headline has been categorized as sarcastic or not.

We'll train a classifier on this, and it can then tell us afterwards if a new piece of text looks like it might be sarcastic.

We'll use Rishabh Misra's dataset from Kaggle, and you can find details on it here.

The data is nice and simple.

The is\_sarcastic field is 1 if it's sarc-y, and 0 otherwise.

There is a headline where the text will train on, and then there's a URL to the article if you're interested in reading it.

But we're not going to use this, just the headline text.

The data is stored in JSON format like this, pretty straightforward.

We'll have to convert it to Python format for training, so it will look like this.

Every JSON element becomes a Python list element, and it's all encapsulated in square brackets.

Python has a JSON toolkit that can achieve this.

And here's the complete code.

We'll go through it step by step.

First of all, we'll import the JSON library.

Then, we can load in the sarcasm JSON file using the JSON library.

We can then create lists for the labels, headlines, and article URLs.

And when we iterate through the JSON, we can load the requisite values into our Python list.

Now that we have three lists, one with our labels, one with the text, and one with the URLs, we can start doing a familiar preprocessing on the text.

Here's the code.

By calling tokenizer.fit on texts with the headline, we'll create tokens for every word in the corpus.

And then, we'll see them in the word index.

You can see an example of some of the words here.

So "underwood" has been tokenized at 24127, and "skillingsbolle"-- what is that, anyway-- to 23055.

So now, we can turn our sentences into sequences of tokens, and pad them to the same length with this code.

If we want to inspect them, we can simply print them out.

Here you can see one tokenized sentence and the shape of the entire corpus.

That's 26,709 sequences, each with 40 tokens.

Now, there's a problem here.

We don't have a split in the data for training and testing.

We just have a list of 26,709 sequences.

Fortunately, Python makes it super easy for us to slice this up.

Let's take a look at that next.

So we have a bunch of sentences in a list and a bunch of labels in a list.

To slice them into training and test sets is actually pretty easy.

If we pick a training size, say 20,000, we can cut it up with code like this.

So the training sentences will be the first 20,000 sliced by this syntax, and the testing sentences will be the remaining slice, like this.

And we can do the same for the labels to get a training and a test set.

But there's a bit of a problem.

Remember earlier we used the tokenizer to create a word index of every word in the set? That was all very good.

But if we really want to test its effectiveness, we have to ensure that the neural net only sees the training data, and that it never sees the test data.

So we have to rewrite our code to ensure that the tokenizer is just fit to the training data.

Let's take a look at how to do that now.

Here's the new code to create our training and test sets.

Let's look at it line by line.

We'll first instantiate a tokenizer like before, but now, we'll fit the tokenizer on just the training sentences that we split out earlier, instead of the entire corpus.

And now, instead of one overall set of sequences, we can now create a set of training sequences, and pad them, and then do exactly the same thing for the test sequences.

It's really that easy.

But you might be wondering at this point, we've turned our sentences into numbers, with the numbers being tokens representing words.

But how do we get meaning from that? How do we determine if something is sarcastic just from the numbers? Well, here's where the context of embeddings come in.

Let's consider the most basic of sentiments.

Something is good or something is bad.

We often see these as being opposites, so we can plot them as having opposite directions like this.

So then what happens with a word like "meh"? It's not particularly good, and it's not particularly bad.

Probably a little more bad than good.

So you might plot it a bit like this.

Or the phrase, "not bad," which is usually meant to plot something as having a little bit of goodness, but not necessarily very good.

So it might look like this.

Now, if we plot this on an x- and y-axis, we can start to determine the good or bad sentiment as coordinates in the x and y.

Good is 1, 0.

Meh is minus 0.4, 0.7, et cetera.

By looking at the direction of the vector, we can start to determine the meaning of the word.

So what if you extend that into multiple dimensions instead of just two? What if words that are labeled with sentiments, like sarcastic and not sarcastic, are plotted in these multiple dimensions? And then, as we train, we try to learn what the direction in these multi-dimensional spaces should look like.

Words that only appear in the sarcastic sentences will have a strong component in the sarcastic direction, and others will have one in the not-sarcastic direction.

As we load more and more sentences into the network for training, these directions can change.

And when we have a fully trained network and give it a set of words, it could look up the vectors for these words, sum them up, and thus, give us an idea for the sentiment.

This concept is known as embedding.

So going back to this diagram, consider what would have happened if I said something was "not bad, a bit meh." If we were to sum up the vectors, we'd have something that's 0.7 on y and 0.1 on x.

So its sentiment could be considered slightly on the good side of neutral.

So now, let's take a look at coding this.

Here's my neural network code.

The top layer is an embedding, where the direction of each word will be learned epoch by epoch.

After that, we pool with a global average pooling, namely adding up the vectors, as I demonstrated earlier.

This is then fed into a common or garden deep neural network.

Training is now as simple as model.fit, using the training data and labels, and specifying the testing padded and labels for the validation data.

At this URL, you can try it out for yourself.

And here, you can see the results that I got training it for just 30 epochs.

While it was able to fit the training data to 99% accuracy, more importantly, with the test data, that is words that the network has never seen, it's still got 81% to 82% accuracy, which is pretty good.

So how do we use this to establish sentiment for new sentences? Here's the code.

Let's create a couple of sentences that we want to classify.

The first one looks a little bit sarcastic, and the second one's quite plain and boring.

We'll use the tokenizer that we created earlier to convert them into sequences.

This way, the words will have the same tokens as the training set.

We'll then pad those sequences to be the same dimensions as those in the training set and use the same padding type.

And we can then predict on the padded set.

The results are like this.

The first sentence gives me 0.91, which is very close to 1, indicating that there's a very high probability of sarcasm.

The second is 5 times 10 to the minus 6, indicating an extremely low chance of sarcasm.

It does seem to be working.

All of this code is runnable in a Colab at this URL.

So give it a try for yourself.

You've now built your first text-classification model to understand sentiment in text.

Give it a try for yourself, and let us know what kind of classifiers you built.

I hope you've enjoyed this short series, and there's more on the way.

So don't forget to hit that Subscribe button and get the latest and greatest in AI videos right here on the TensorFlow Channel.

**Part 4**

LAURENCE MORONEY: Hi, and welcome to this episode in "Natural Language Processing, Zero to Hero" with TensorFlow.

In the previous videos in this series, you saw how to tokenize text and how to use sequences of tokens to train a neural network.

In particular, you saw how to create a neural network that classified text by sentiments.

And in this case, you trained a classifier on sarcasm headlines.

But the next step I'm often asked when it comes to text is, what about generating text? Can a neural network create text based on the corpus that it's trained on, and can we get an AI to write poetry? Well, the answer to this is yes.

And over the next few videos, I'll show you a simple example on how you can achieve this.

Before we can do that, though, an important concept that you'll need to understand is recurrent neural networks.

This type of neural network takes the sequence of data into account when it's learning.

So for example, in the case of a classifier for text that we just saw, the order in which the words appear in the sentence doesn't really matter.

What determined the sentiment was the vector that resulted in adding up all of the individual vectors for the individual words.

The direction of that vector roughly gave us the sentiments.

But if we're going to generate text, the order does matter.

For example, consider this sentence.

"Today the weather is gorgeous, and I see a beautiful blue"-- something.

If you were trying to predict the next word-- and the concept of creating text really boils down to predicting the next word-- you'd probably say, "sky," because that comes after "beautiful" and "blue," and the context is the weather, which we saw earlier in the sentence.

So how do we fit this to neural networks? Let's take a look at what's involved in changing from sequence list data to sequential data.

Neural networks for classification or regression tend to look like this.

It's kind of like a function that you feed in data and labels, and it infers the rules that fits the data to the labels.

But you could also express it like this.

The f of data and labels equals the rules.

But there's no sequence inherent in this.

So let's take a look at some numeric sequences and explore the anatomy of them.

And here's a very famous one called the Fibonacci sequence.

To describe the rules that make this sequence, let's describe the numbers using a variable.

So for example, we can say n0 for the first number, n1 for the next, and so on.

And the rule that then defines the sequence is that any number in the sequence is the sum of the two numbers before it.

So if we start with 1 and 2, the next number is 1 plus 2, which is 3.

The next number is 5, which is 2 plus 3, and so on.

We could also try to visualize it like this on a computation graph.

If the function is plus, we feed in 1 and 2 to get 3.

We also pass this answer and the second parameter, which in this case was 2, onto the next computation.

This gives us 2 plus 3, which is 5.

This gets fed into the next computation along with the second parameter, so 5 plus 3 get added to get 8, and so on.

So every number is in essence contextualized into every other number.

We started with 1, and added it to 2 to get 3.

The 1 and the 3 still exists.

And when added to 2 again, we get 5.

That 1 still continues to exist throughout the series.

Thus, a numeric value can recur throughout the life of the series.

And this is the basis of the concept of a recurrent neural network.

Let's take a look at this type of network in a little more detail.

Typically, a recurrent neuron is drawn like this.

There's a function that gets an input value that produces an output value.

In addition to the output, it also produces another feed-forward value that gets passed to the next neuron.

So a bunch of them together can look like this.

And reading from left to right, we can feed x0 into the neuron, and it calculates a result, y0, as well as a value that gets passed to the next neuron.

That gets x1 along with the fed-forward value from the previous neuron and calculates y1.

And its output is combined with x to get y2 and a feed-forward value to the next neuron, and so on.

Thus, sequence is encoded into the outputs, a little bit like the Fibonacci sequence.

This recurrence of data gives us the name recurrent neural networks.

So that's all very well.

And you may have seen a little catch in how this could work with natural language processing.

A simple RNN like the one that I've just shown is a bit like the Fibonacci sequence in that the sequence can be very strong, but it weakens as the context spreads.

The number at the position 1 has very little impact on the number at the position 100, for example.

It's there, but it's tiny.

And that could be useful for predicting text where the signal to determine the text is close by, for example, the beautiful blue something that we mentioned earlier.

It's easy for us to see that "sky" is the next word.

But what about a sentence like this? "I lived in Ireland, so they taught me how to speak"-- something.

Now, you might think it's "Irish," but the correct answer is "Gaelic." But think about how you predicted that word.

The key word that dictated it was much further back in the sentence, and it's the word "Ireland." If we were only predicting based on the words that are close to the desired one, we'd miss that completely, and we'd get a bad prediction.

The key there is to go beyond the very short-term memory of a recurrent neural network with a longer short-term memory and a network type not surprisingly called long short-term memory, or LSTM.

You'll see that in the next video, so don't forget to hit that Subscribe button for more great episodes of "Coding TensorFlow at Home." [MUSIC PLAYING]

**Part 5**

LAURENCE MORONEY: Hi, and welcome to episode 5 of our Natural Language Processing with TensorFlow series.

In this video, we're going to take a look at how to manage the understanding of context in language across longer sentences, where we can see that the impact of a word early in the sentence can determine the meaning and semantics of the end of the sentence.

We'll use something called an LSTM, or Long Short Term Memory, to achieve this.

So for example, if we're predicting text and the text looks like this-- today has a beautiful blue something-- it's easy to predict that the next word is probably sky, because we have a lot of context close to the word, and most notably the word blue.

But what about a sentence like this one-- I lived in Ireland, so I learned how to speak something? How do we predict the something? The correct answer, of course, is Gaelic, not Irish, but that's close enough.

And you and I could interpret that, but how do we do that? What's the keyword that determines this answer? Of course, it's the word Ireland, because in this case, the country determines the language.

But the word is very far back in the sentence.

So when using a recurrent neural network, this might be hard to achieve.

Remember, the recurrent neural networks we've been looking at are a bit like this, where there's a neuron that can learn something and then pass context to the next timestamp.

But over a long distance, this context can be greatly deluded, and we might not be able to see how meanings in faraway words dictate overall meaning.

The LSTM architecture might help here, because it introduces something called a cell state, which is a context that can be maintained across many timestamps, and which can bring meaning from the beginning of the sentence to bear.

It can learn that Ireland denotes Gaelic as the language.

What's fascinating is that it can also be bi-directional, where it might be that later words in the sentence can also provide context to earlier ones so that we can learn the semantics of the sentence more accurately.

I won't go into the specifics of LSTMs in this video, but if you want to learn how they work in depth, the deep learning specialization from Deep Learning AI is a great place to go.

So we've seen in theory how they work.

But what does this look like in code? Let's dive in and take a look.

Let's consider how we would use an LSTM and a classifier like the sarcasm classifier we saw in an earlier video.

It's really quite simple.

We first define that we want an LSTM-style layer.

This takes a numeric parameter for the number of hidden nodes within it, and this is also the dimensionality of the output space from this layer.

If you wanted to be bi-directional, you can then wrap this layer in a bi-directional like this, and you're good to go.

Remember that this will look at your sentence forwards and backwards, learn the best parameters for each, and then merge them.

It might not always be best for your scenario, but it is worth experimenting with.

LSTMs can use a lot of parameters, as a quick look at this model summary can show you.

Note that there are 128 in the LSTM layer, because we're doing a bi-directional using 64 in each direction.

You can, of course, also stack LSTM layers so that the outputs of one layer get fed into the next, a lot like with dense layers.

Just be sure to set return sequences to true on all layers that are feeding another.

So in a case like this, where we have two, the first should have it.

If you have three LSTM layers stacked, the first two should have it, and so on.

And a summary of this model will show the extra parameters that the extra LSTMs give.

So now you've seen the basis of recurrent neural networks, including long short term memory ones.

You've also seen the steps in pre-processing text for training a neural network.

In the next video, you'll put all of this together and start with a very simple neural network for predicting and thus creating original text.

I'll see you there.

And for more videos on AI in TensorFlow, don't forget to hit that Subscribe button.

**Part 6**

LAURENCE MORONEY: Through this series so far, you've been learning the basics of NLP using TensorFlow.

You saw how to tokenize and then sequence text, preparing it to train neural networks.

You saw how sentiment in text can be represented with embeddings and how the semantics of text over long stretches might be learned using recurrent neural networks and LSTMs.

In this video, we'll put all of that together into a fun scenario.

We'll create a model and train it on the lyrics of traditional Irish songs.

From that, you'll see then if it can write its own poetry using those words.

Let's look at the steps involved.

First of all, this is our text.

Within the entire corpus are the lyrics to lots of Irish songs.

One of them, "Lanigan's Ball," is listed here, and you can see these words have a very distinctive style.

If we were to read them in, we could do it something like this.

And for simplicity, I'll just use this one song for now.

It's stored as a single string with slash n's to give new lines.

That, I can then break into a number of sentences by splitting the string by that new line character, and this will form my corpus of text.

Later, you'll see how to change it to read the full corpus off of disk, but the methods will be exactly the same.

I can then fit my tokenizer to the corpus to get a word index.

As I'm using an out of vocabulary token, I'll add 1 to the length of the word index just to cater for that.

Now, you might wonder, why not just encode with an out of vocabulary token? There's a subtle difference here when generating text over the previous scenario when we were classifying text.

When generating text, we don't need a validation data set.

We're going to use every bit we have to try to spot the patterns of where and how words occur.

So if we tokenize our entire corpus, there will be, by definition, no out of vocabulary token.

However, in a moment you'll see where we will start to pad subsentences from the full corpus, and for that, we'll need some kind of a zero token.

Hence, we'll add one here, and counting that token as a valid word.

Now that we have a list of sentences and we've tokenized them, we can turn them into a set of training data.

Now, there's a key difference here between what we've seen previously for classification and what we'll use for generation.

So let's go over this line by line so it's clear.

First of all, I'll create an empty list of input sequences.

We'll populate this as we go along.

Now, for each line in the corpus, we'll create the list of tokens.

Note that we're not doing text to sequences for the entire body.

We're going to do it one line at a time.

So this will give me the text to sequences for the current line.

Now for example, this will sequence just the first line the first time through the loop.

And "In the town of Athy one Jeremy Lanigan" will be tokenized into the numbers as shown.

Next, we're going to go through this list and generate n grams from that.

What does that mean? It's best if we look at it like this.

The line that we tokenized is represented by a list of numbers, but we can split that into a number of other lists.

The first two, the first three, the first four, and so on.

The reason for that is that we want to train a model to predict the likely next word.

So for each sentence we have, we can train it for when you see this word, this one is next.

When you see these two words, this one is next.

When you see these three words, this one is next, and so on.

Now that we've split the sentence into multiple lists, we'll need to pad it.

So we'll start by getting the length of the longest of the sentences and then pad everything with a 0 up to the length of the maximum sentence.

So now our line of eight words has formed the same seven lists, but each one is now padded with 0's to begin.

Thus, we can see our set of input sequences for this one line just looks like this.

And this is ideal for giving us features and labels or X's and Y's.

We can take everything but the last value as our X, and we can use the last value as our Y.

So when we see a bunch of 0's followed by a 4, the label for that will be 2.

Similarly, when we see a bunch of 0's ending with a 4 and then a 2, the label for that will be 66.

Similarly, 4 to 66 will be labeled as 8.

Python makes it super simple for us to slice our lists like this.

We can simply use code like this to generate our X's and now our labels.

Finally, we'll want our Y to be categorical and one hot encoded, so that when we train, we'll be able to predict across all of the words in our corpus which one is the most likely word to be next in the sequence given the current set of words.

And then we can use the keras to categorical to achieve this.

So for example, given the above sentence, we'll split it into X and label, where X is the beginning of the list, and the label is 70.

We can then 1, hot encode the label to get the Y.

And if you look closely, you'll see that the seventieth element in the y list is a one while everything else is 0.

So now we have our features and our labels, let's train a neural network with all of the data.

And here's a very simple model architecture to achieve that.

This is completely unoptimized, particularly in the middle layers, so please feel free to experiment and improve it.

It starts with a sequential, adds an embedding at the top like we saw earlier.

As there's a massive variation of words, I gave it a lot of dimensions.

And in this case, it's 240.

The first parameter is the number of unique words in the corpus.

The input length is the maximum sequence length minus 1, because we lopped off the final value in each sequence to make a label.

After that, we've just got a single LSTM, but we'll make it bi-directional.

And then importantly, our output is a dense with the total number of words.

Remember that the labels were 1 hot encoded, so we want an output that is representative of this.

It's then a matter of defining your loss function and optimizer.

Remember, as this is categorical with lots of classes, you'll need a categorical loss function such as categorical cross entropy here.

And once you've done that, you just fit the X's to the Y's.

As you're training, you might see the initial accuracy is really small like 0.05 or 0.06.

Don't worry, it will go up with time.

This is very unstructured data, and it's trying to figure out the rules that match your X's to your Y's.

When it's done, you'll have a model that you can pass it a sequence, and it will give you the predicted next value.

You can use this to then generate poetry, take a sequence, and get the next value, add that to the sequence, pass that to the model, get the next value, add that to the sequence, and so on.

With the simple model architecture above, it ends up with an accuracy around 70 to 75%.

And that means that given a sequence of words, it will pick the correct word right about 70% of the time.

If it gets a sequence of words it hasn't previously seen, it can make a rough prediction for what the next word could be.

So to get it to generate text, we can seed it with some words and predict the next value.

We'll add that to our string of words and get it to predict the next value, and so on.

And here's the code for that.

And when seeded with the words, "I made a poetry machine," I got the following sequence generated for me.

It's not bad, though if anybody can explain "shed love raw boo," please let me know in the comments below.

Experiment with different architectures and run times, and let me know what you come up with.

Now, that brings us to the end of this series on NLP.

I hope you've enjoyed it, and if you want more, please let us know in the comments, and don't forget to hit that Subscribe button for more great TensorFlow content.

Thank you.