Tokenizing

**Tokenizing Text**

Neural networks utilize numbers as their inputs, so we need to convert our input text into numbers. Tokenization is the process of assigning numbers to our inputs, but there is more than one way to do this - should each letter have its own numerical token, each word, phrase, etc.

As you saw in the video, tokenizing based on letters with our current neural networks doesn’t always work so well - anagrams, for instance, may be made up of the same letters but have vastly different meanings. So, in our case, we’ll start by tokenizing each individual word.

**Tokenizer**

With TensorFlow, this is done easily through use of a Tokenizer, found within tf.keras.preprocessing.text. If you wanted only the first 10 most common words, you could initialize it like so:

tokenizer = Tokenizer(num\_words=10)

**Fit on Texts**

Then, to fit the tokenizer to your inputs (in the below case a list of strings called sentences), you use .fit\_on\_texts():

tokenizer.fit\_on\_texts(sentences)

**Text to Sequences**

From there, you can use the tokenizer to convert sentences into tokenized sequences:

tokenizer.texts\_to\_sequences(sentences)

**Out of Vocabulary Words**

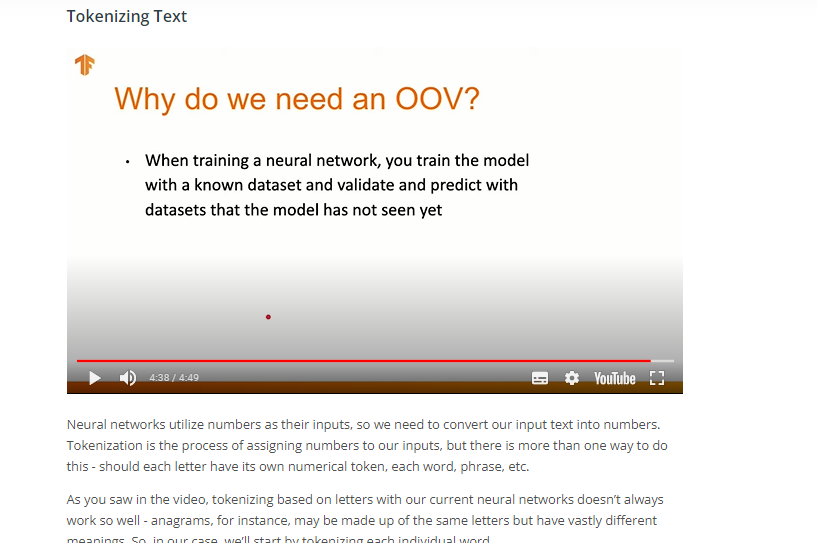
However, new sentences may have new words that the tokenizer was not fit on. By default, the tokenizer will just ignore these words and not include them in the tokenized sequences. However, you can also add an “out of vocabulary”, or OOV, token to represent these words. This has to be specified when originally creating the Tokenizer object.

tokenizer = Tokenizer(num\_words=20, oov\_token=’OOV’)

**Viewing the Word Index**

Lastly, if you want to see how the tokenizer has mapped numbers to words, use the tokenizer.word\_index property to see this mapping.

**QUESTION 1 OF 2**



Text to Sequences:

Even after converting sentences to numerical values, there’s still an issue of providing equal length inputs to our neural networks - not every sentence will be the same length!

There’s two main ways you can process the input sentences to achieve this - padding the shorter sentences with zeroes, and truncating some of the longer sequences to be shorter. In fact, you’ll likely use some combination of these.

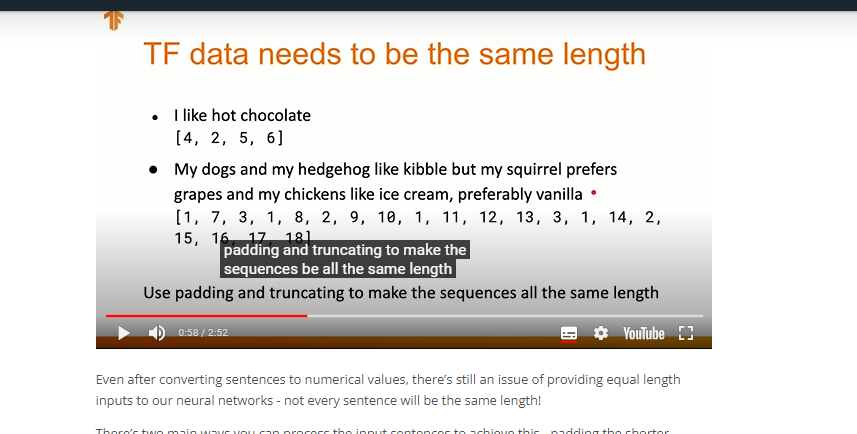
With TensorFlow, the pad\_sequences function from tf.keras.preprocessing.sequence can be used for both of these tasks. Given a list of sequences, you can specify a maxlen (where any sequences longer than that will be cut shorter), as well as whether to pad and truncate from either the beginning or ending, depending on pre or post settings for the padding and truncating arguments. By default, padding and truncation will happen from the beginning of the sequence, so set these to post if you want it to occur at the end of the sequence.

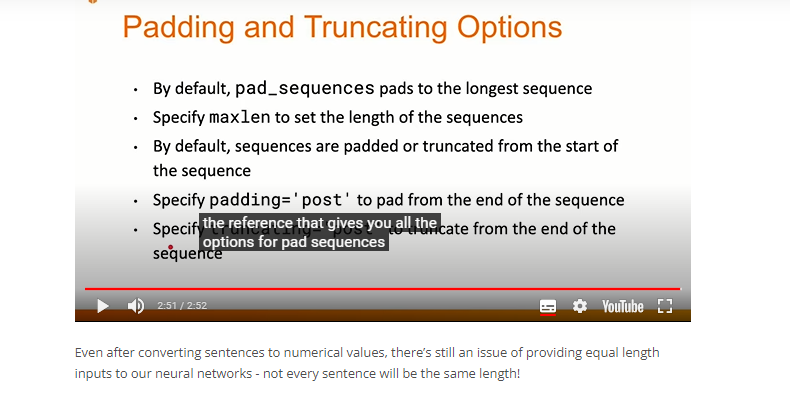
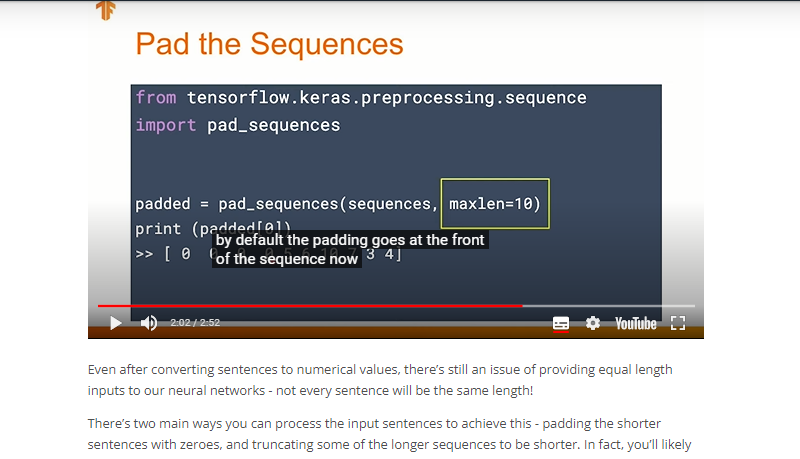
If you wanted to pad and truncate from the beginning, you could use the following:

padded = pad\_sequences(sequences, maxlen=10)

### Further Research

Head [here](https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/pad_sequences) if you’d like to check out the full TensorFlow documentation for pad\_sequences.

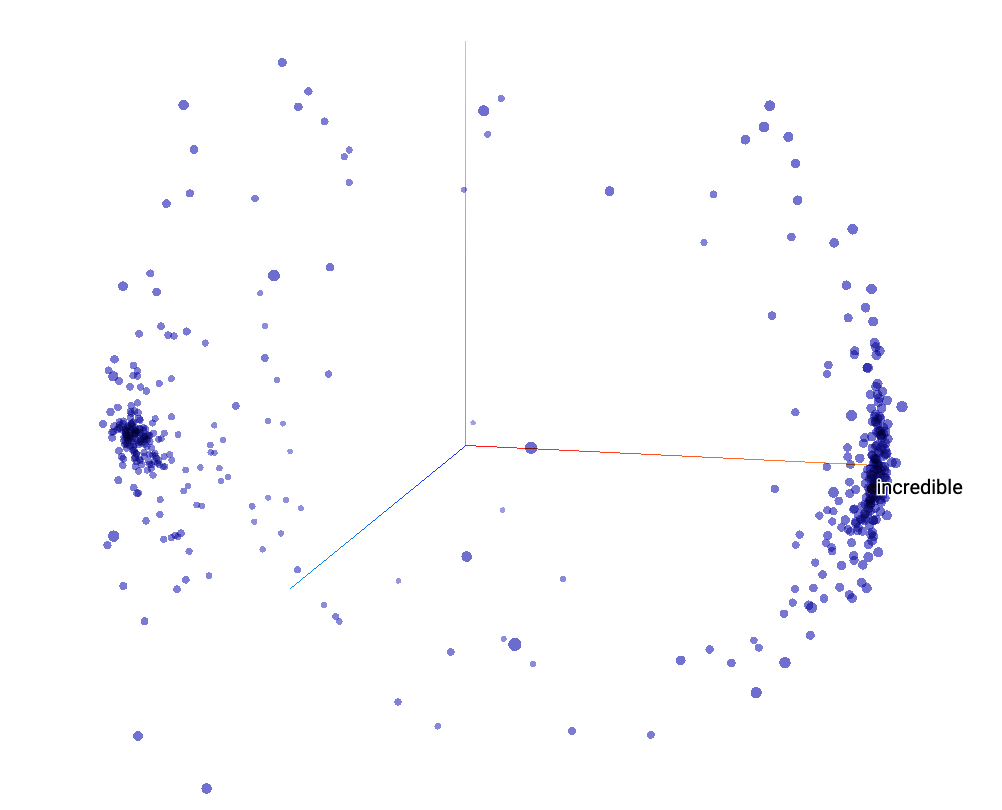




**Word Embeddings**

Embeddings are clusters of vectors in multi-dimensional space, where each vector represents a given word in those dimensions. While it’s difficult for us humans to think in many dimensions, luckily the [TensorFlow Projector](http://projector.tensorflow.org/" \t "_blank) makes it fairly easy for us to view these clusters in a 3D projection (later Colabs will generate the necessary files for use with the projection tool).

This can be very useful for sentiment analysis models, where you’d expect to see clusters around either more positive or more negative sentiment associated with each word.



An example of a post-training embedding projection, with clear distinctions between positive and negative sentiments.

## Building a Basic Sentiment Model

To create our embeddings, we’ll first use an embeddings layer, called [tf.keras.layers.Embedding](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding" \t "_blank). This takes three arguments: the size of the tokenized vocabulary, the number of embedding dimensions to use, as well as the input length (from when you standardized sequence length with padding and truncation).

The output of this layer needs to be reshaped to work with any fully-connected layers. You can do this with a pure Flatten layer, or use GlobalAveragePooling1D for a little additional computation that sometimes creates better results.

In our case, we’re only looking at positive vs. negative sentiment, so only a single output node is needed (0 for negative, 1 for positive). You’ll be able to use a binary cross entropy loss function since the result is only binary classification.

### QUIZ QUESTION

### A Note on Embedding Networks

The TensorFlow team has two additional suggestions for you, on top of what we show in the video and what we included in the Colab coming up.

They suggest that the final network does not use a sigmoid activation layer when working with embeddings, especially when using just the two classes like we are for sentiment analysis:

tf.keras.layers.Dense(1)

Additionally, they suggest instead of using the string ”binary\_crossentropy” as the loss function, you use tf.keras.losses.BinaryCrossentropy(from\_logits=True).

## Visualizing Embeddings

We’ve given you the code to create the files for input into the [projector](http://projector.tensorflow.org/). This will download two files: 1) the vectors, and 2) the metadata.

The projector will already come with a pre-loaded visualization, so you’ll need to use the “Load” button on the left and upload each of the two files. In some cases, there may be a small difference in the number of tensors present in the vector file and the metadata file (usually with a message appearing after uploading the metadata); if this appears, wait for a few seconds for the error message to disappear, and then click outside the window. Typically, the visualization will still load just fine.

Make sure to click the checkbox for “Sphereize data”, which will better show whether there is separation between positive and negative sentiment (or not).

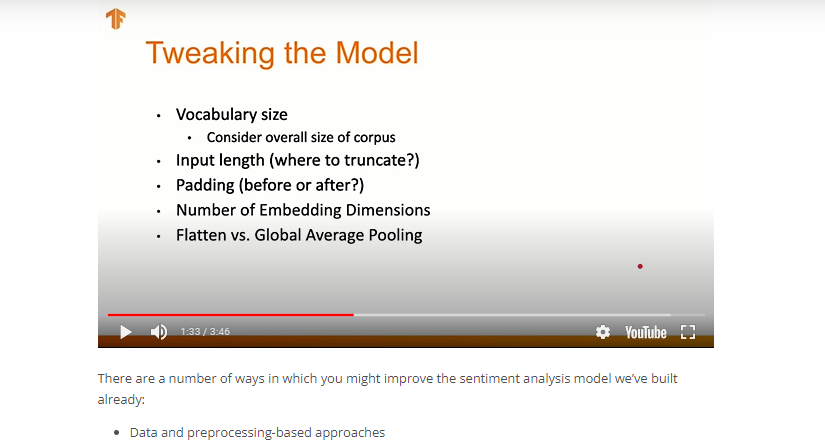
## Tweaking the Model

There are a number of ways in which you might improve the sentiment analysis model we’ve built already:

* Data and preprocessing-based approaches
  + More data
  + Adjusting vocabulary size (make sure to consider the overall size of the corpus!)
  + Adjusting sequence length (more or less padding or truncation)
  + Whether to pad or truncate pre or post (usually less of an effect than the others)
* Model-based approaches
  + Adjust the number of embedding dimensions
  + Changing use of Flatten vs. GlobalAveragePooling1D
  + Considering other layers like Dropout
  + Adjusting the number of nodes in intermediate fully-connected layers

These are just some of the potential things you might tweak to better predict sentiment from text.

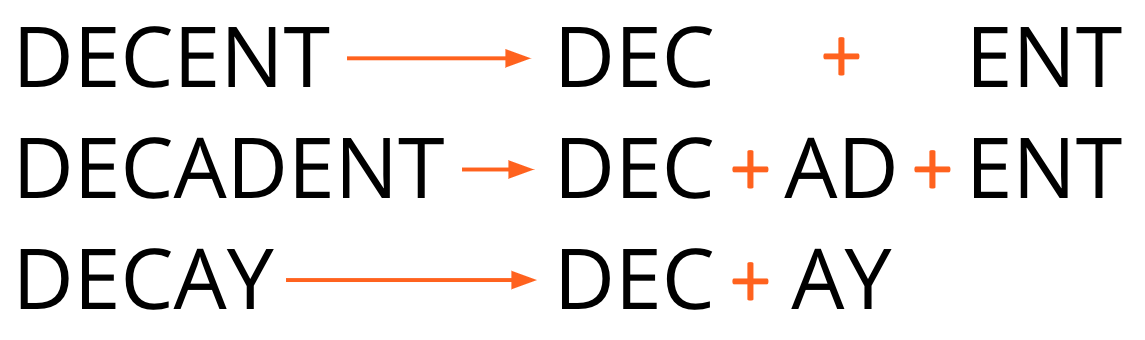
**Tweaking the Model Quiz**



## What's in a (sub)word?

We’ve worked with full words before for our sentiment models, and Jocelyn had shown us some issues right at the start of the lesson when using character-based tokenization. Subwords are another approach, where individual words are broken up into the more commonly appearing pieces of themselves. This helps avoid marking very rare words as OOV when you use only the most common words in a corpus.

As shown in the video, this can further expose an issue affecting all of our models up to this point, in that they don’t understand the full context of the sequence of words in an input. The next lesson on recurrent neural networks will help address this issue.



Our example subwords using Decent, Decadent and Decay.

### Subword Datasets

There are a number of already created subwords datasets available online. If you check out the [IMDB dataset on TFDS](https://www.tensorflow.org/datasets/catalog/imdb_reviews), for instance, by scrolling down you can see datasets with both 8,000 subwords as well as 32,000 subwords in a corpus (along with regular full-word datasets).

However, I want you to know how to create these yourself as well, so we’ll use TensorFlow’s SubwordTextEncoder and [its build\_from\_corpus function](https://www.tensorflow.org/datasets/api_docs/python/tfds/features/text/SubwordTextEncoder#build_from_corpus) to create one from the reviews dataset we used previously.

You’ve already learned an amazing amount of material on Natural Language Processing with TensorFlow in this lesson.

You started with Tokenization by:

* Tokenizing input text
* Creating and padding sequences
* Incorporating out of vocabulary words
* Generalizing tokenization and sequence methods to real world datasets

From there, you moved onto Embeddings, where you:

* transformed tokenized sequences into embeddings
* developed a basic sentiment analysis model
* visualized the embeddings vector
* tweaked hyperparameters of the model to improve it
* and diagnosed potential issues with using pre-trained subword tokenizers when the network doesn’t have sequence context

In the next lesson, you’ll dive into Recurrent Neural Networks, which will be able to understand the sequence of inputs, and you'll learn how to generate new text.