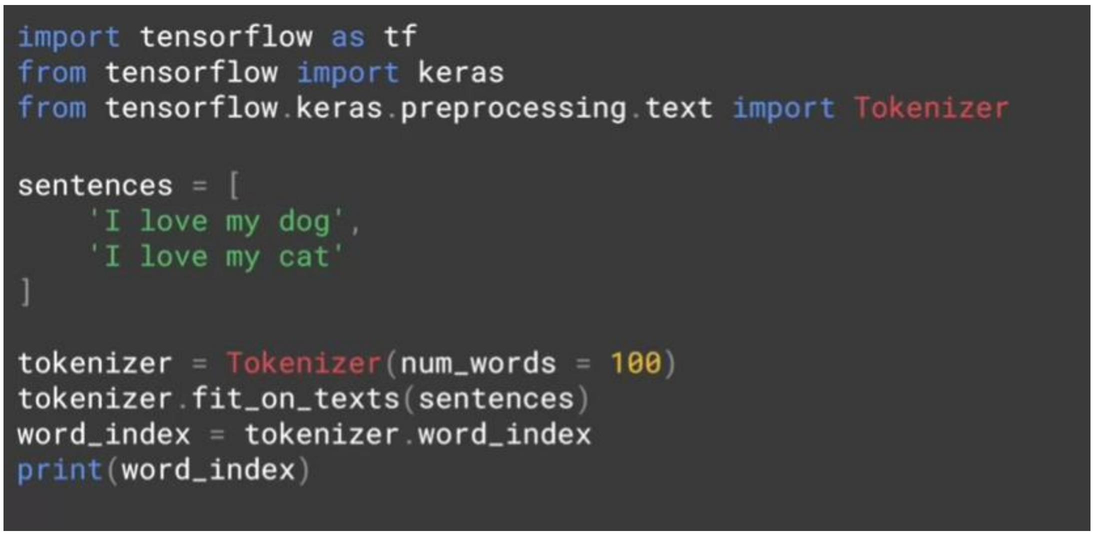
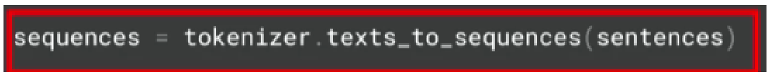


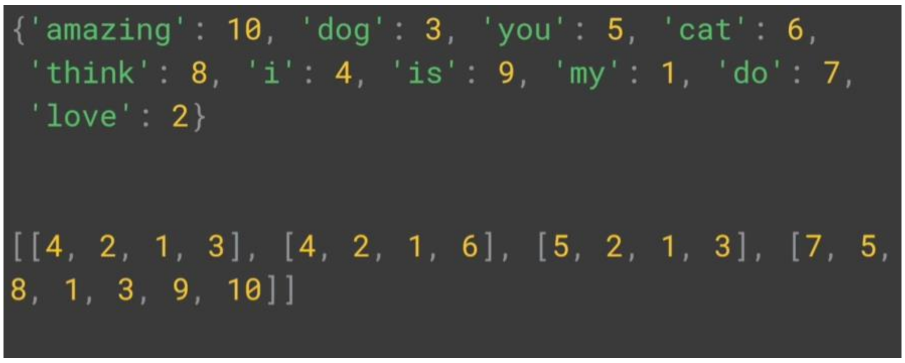
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer

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: Tensorflow and keras give us a number of ways to encode words, one way is the tokenizer. This will generate the dictionary of word encodings and creating vectors out of the sentences. The fit on texts method of the tokenizer then takes in the data and encodes it. The tokenizer provides a word index property which returns a dictionary containing key value pairs, where the key is the word, and the value is the token for that word. (strips punctuations and is case insensitive).

TEXT TO SEQUENCE: Remember when we were doing images, we defined an input layer with the size of the image that we're feeding into the neural network. In the cases where images where differently sized, we would resize them to fit (to make every sentence the same length).



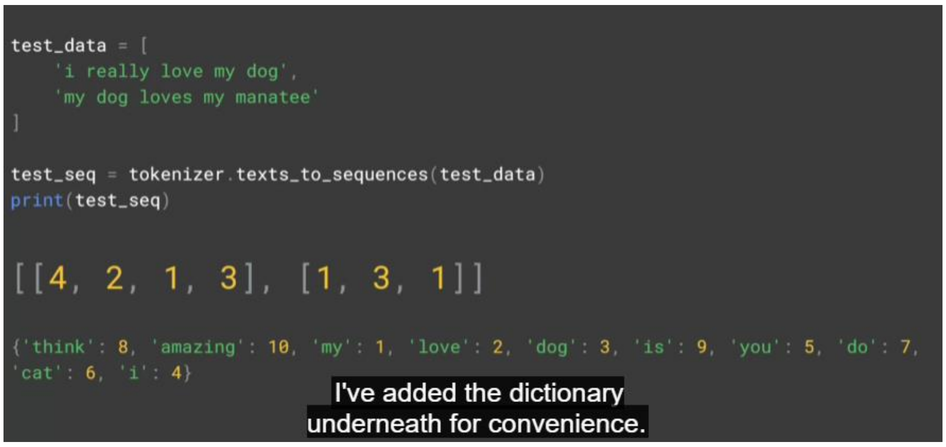
```
sequences = tokenizer.texts_to_sequences(sentences)
```



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


[[4, 2, 1, 3], [4, 2, 1, 6], [5, 2, 1, 3], [7, 5,
8, 1, 3, 9, 10]]
```

for example, I love my dog becomes 4, 2, 1, 3.

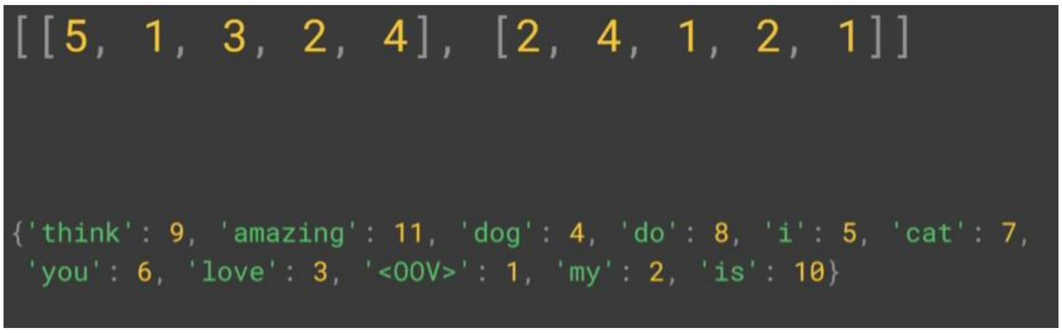
The text to sequences called can take any set of sentences, so it can encode them based on the word set that it learned from the one that was passed into fit on texts. If you train a neural network on a corpus of texts, and the text has a word index generated from it, then when you want to do inference with the train model, you'll have to encode the text that you want to infer on with the same word index, otherwise it would be meaningless.



```
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}
```

I've added the dictionary
underneath for convenience.

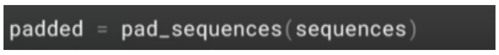
“really” being lost as it was not a part of the dictionary previously. Hence, we need a broad vocabulary and not to instead of just ignoring unseen words, to put a special value in when an unseen word is encountered. Property **<OOV> Token** is used i.e. out of vocabulary for words that aren’t in index.



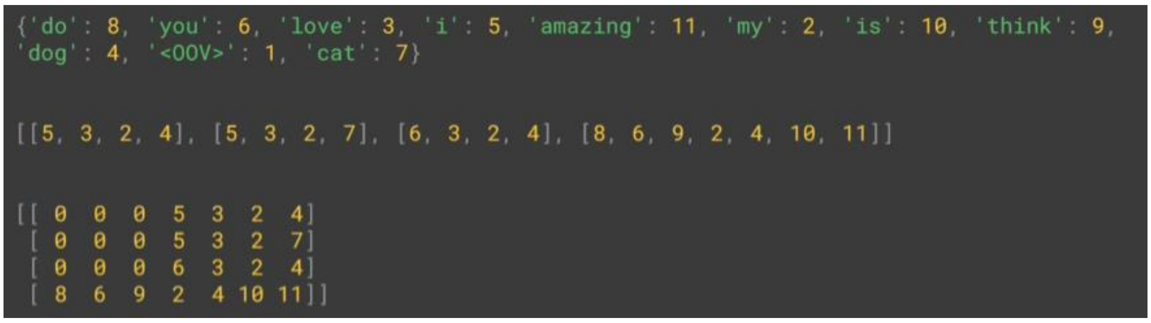
```
[[5, 1, 3, 2, 4], [2, 4, 1, 2, 1]]



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

The first sentence will be, i <OOV> love my dog.
The second will be, my dog <OOV> my <OOV>. Still not syntactically correct, but better coverage.

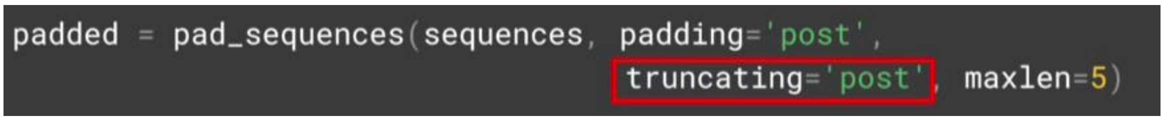
PADDING: Before feeding training texts, we need some level of uniformity in size, so padding is used. from tensorflow.keras.preprocessing.sequence import pad_sequences



```
padded = pad_sequences(sequences)
```



```
{'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]]
```

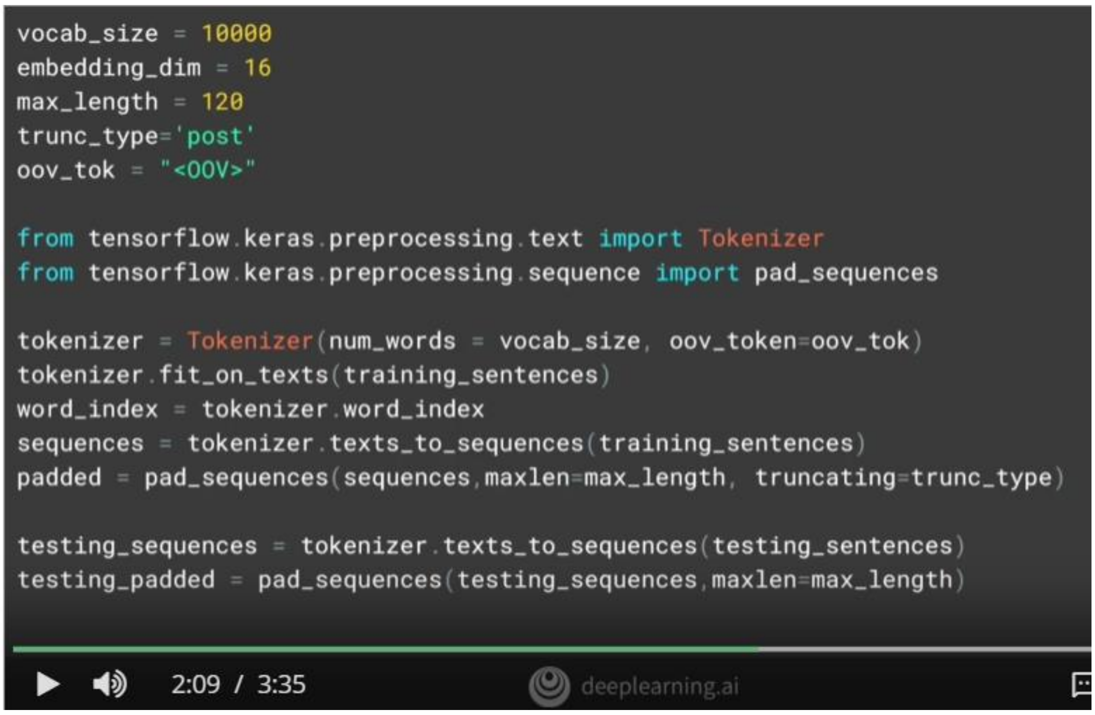


```
padded = pad_sequences(sequences, padding='post',
                        truncating='post', maxlen=5)
```

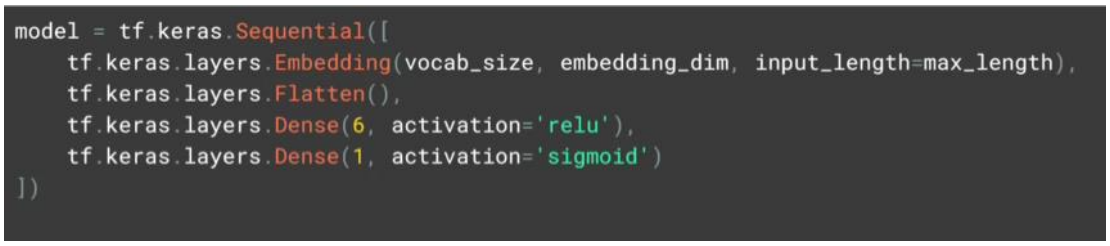
‘post’ to add padding after the sentence and not before. Matrix width can be customised with maxlen parameter i.e. the maximum no. of words in a sentence. If I have sentences longer than the maxlength, then I'll lose information, but from where? Like with the padding the default is pre, which means that you will lose from the beginning of the sentence. If you want to override this so that you lose from the end instead, you can do so with the truncating parameter like this.

WORD EMBEDDING: a type of word representation that allows words with similar meaning to have a similar representation. it's like a vector in n-dimensional space. So for example the word dog, might be a vector pointing in a particular direction and then the word "canine", could be learned as a vector pointing in a very similar direction, and we know they have very similar semantic meaning off of that.

Right now, it's still just a string of numbers representing words. So how would one actually get sentiment? Well, that's something that can be learned from a corpus of words in much the same way as features were extracted from images. This process is called embedding, with the idea being that words and associated words are clustered as vectors in a multi-dimensional space.



```
vocab_size = 10000
embedding_dim = 16
max_length = 120
trunc_type='post'
oov_tok = "<OOV>"

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

tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(training_sentences)
word_index = tokenizer.word_index
sequences = tokenizer.texts_to_sequences(training_sentences)
padded = pad_sequences(sequences,maxlen=max_length, truncating=trunc_type)

testing_sequences = tokenizer.texts_to_sequences(testing_sentences)
testing_padded = pad_sequences(testing_sequences,maxlen=max_length)
```



```
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')
])
```

The meaning of the words can come from the labelling of the dataset. So in this case, we say a negative review and the words dull and boring show up a lot in the negative review so that they have similar sentiments, and they are close to each other in the sentence. Thus their vectors will be similar. The results of the embedding will be a 2D array with the length of the sentence and the embedding dimension for example 16 as its size. So we need to flatten it out in much the same way as we needed to flatten out our images.

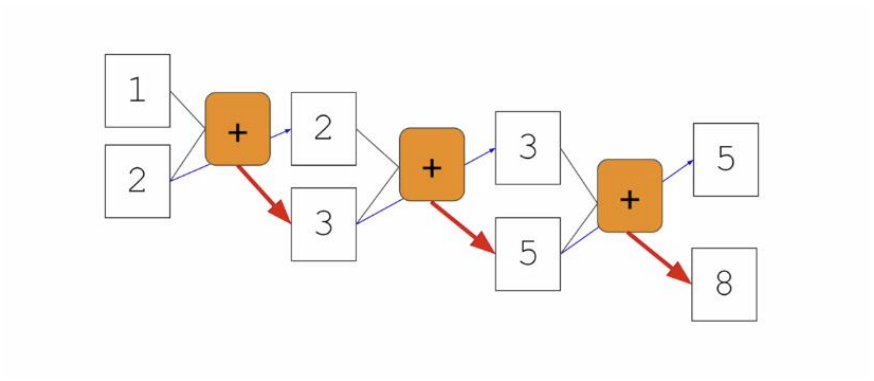
Often in natural language processing, a different layer type than a flatten is used, and this is a global average pooling 1D. The reason for this is the size of the output vector being fed into the Dense.

IMDB Reviews are either positive or negative. Type of loss function used: “Binary Categorical”

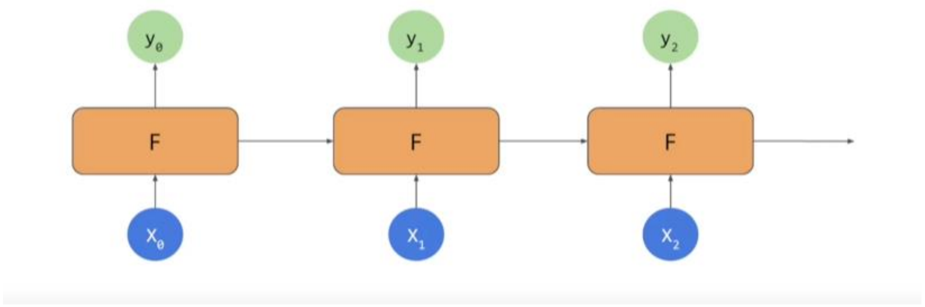
SUBWORDS: Tokenizing sub words, the punctuation is maintained and case sensitive. When using IMDB Sub Words dataset, our results in classification were poor. Why? Sequence becomes much more important when dealing with subwords, but we’re ignoring word positions. not only do the meanings of the words matter but also the sequence in which they are found.

SEQUENCE MODELLING

RNN – RECURRENT NEURAL NETWORK: to take into account the ordering of the words, people now use specialized Neural Network Architectures, things like an RNN, or GIO, or LSTM.



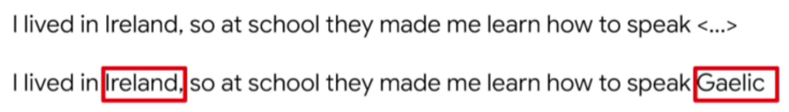
the Fibonacci sequence, example of RNN.



You have your x as in input and your y as an output. But there's also an element that's fed into the function from a previous function. These help to understand the meaning of the sentence better as they carry meaning from one cell to the next.

LSTMs – LONG SHORT TERM MEMORY:

Irish is correct, but to be more accurate answer is Gaelic(It's what the language is called). in the ML context is the key word that gives us the details about the language. That's the word Ireland, which appears much earlier in the sentence.



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

So, if we're looking at a sequence of words we might lose that context. With that in mind an update to RNNs is called LSTM, long short - term memory has been created. In addition to the context being passed as it is in RNNs, LSTMs have an additional pipeline of contexts called cell state. This can pass through the network to impact it. This helps keep context from earlier tokens relevance in later ones so issues like the one that we just discussed can be avoided. Cell states can also be bidirectional. So later contexts can impact earlier ones as we'll see when we look at the code. LSTM help understand meaning when words that qualify each other aren't necessarily beside each other in a sentence.

APPLICATION OF SEQUEMCE MODEL:

One of the most fun applications of sequence models, is that they can read the body of text, so train on the certain body of text, and then generate or synthesize new texts, that sounds like it was written by similar author or set of authors.

What is a major drawback of word-based training for text generation instead of character-based generation? -> Because there are far more words in a typical corpus than characters, it is much more memory intensive.

What function do we use to create one-hot encoded arrays of the labels? ->tf.keras.utils.to_categorical