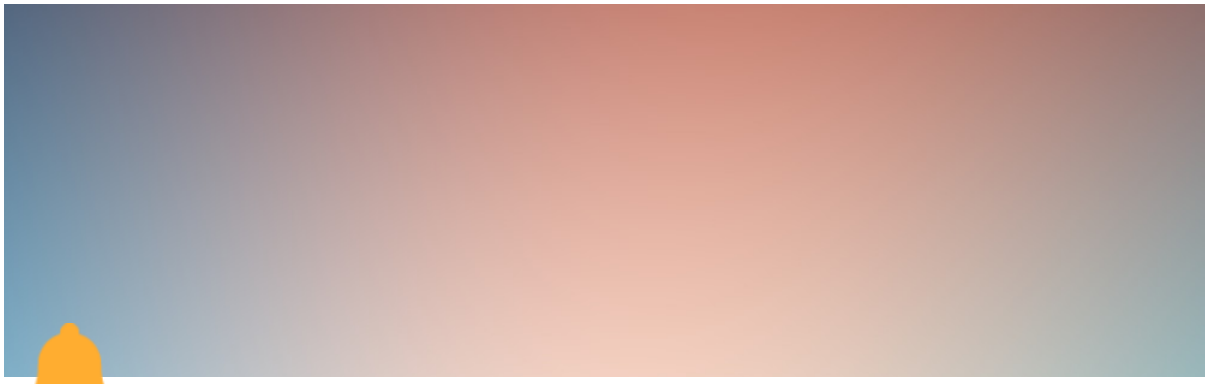
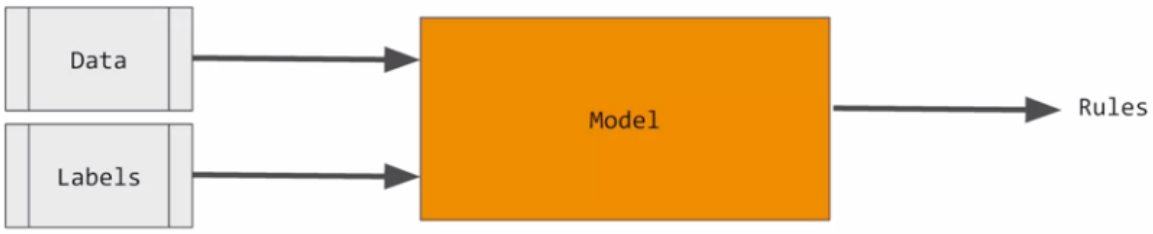


Natural Language Processing in TensorFlow (byDeeplearning.AI)

Week 3: Sequence Models for Natural Language Processing

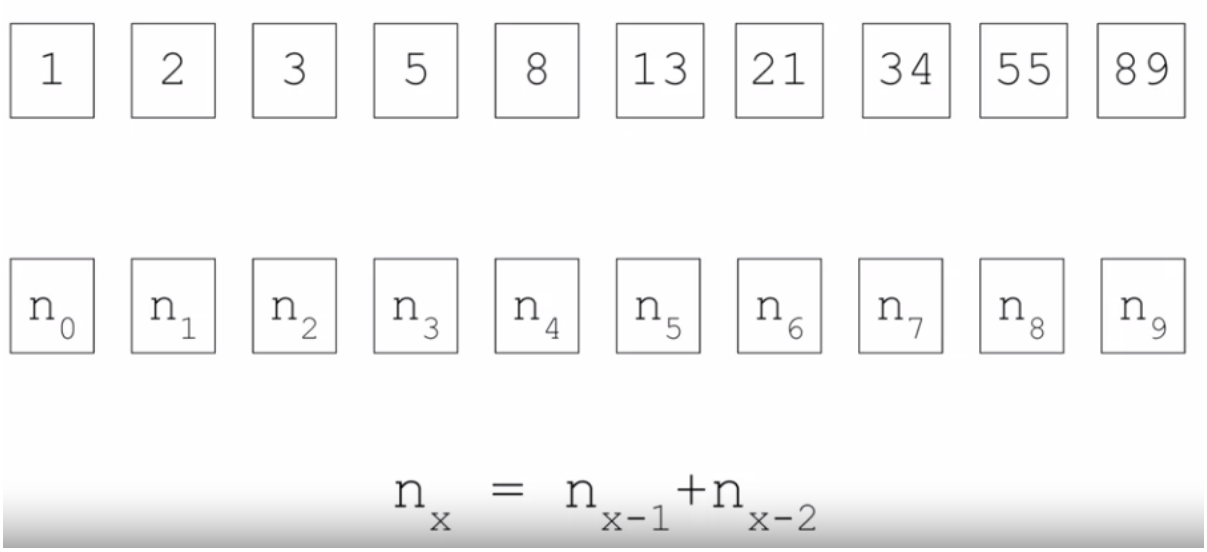
Sequence models are important in NLP since the context of words is hard to follow when the words were broken down into sub-words and the sequence in which the tokens for the sub-words appear becomes very important in understanding their meaning.

The neural network is like a function that when you feed it in data and labels, it infers the rules from these, and then you can use those rules.

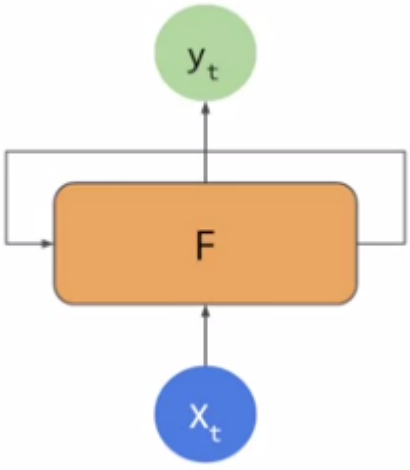


So it could be seen as a function, you take the data and you take the labels, and you get the rules. But this doesn't take any kind of sequence into account.

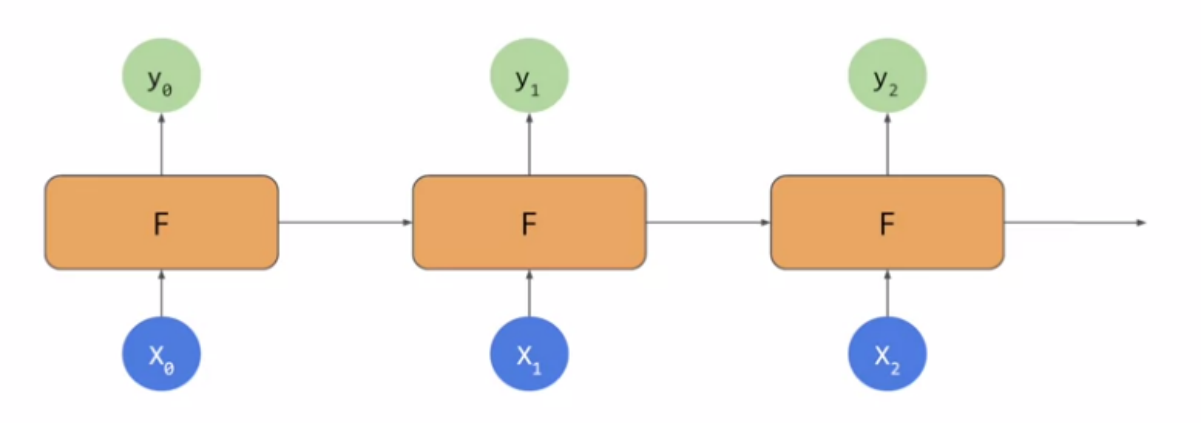
To understand why sequences can be important, consider this set of numbers.



If you've never seen them before, they're called the Fibonacci sequence. So let's replace the actual values with variables such as n_0, n_1 and n_2, etc., to denote them. Then the sequence itself can be derived where a number is the sum of the two numbers before it. So 3 is 2 plus 1, 5 is 2 plus 3, 8 is 3 plus 5, etc. Our n_x equals n_x minus 1, plus n_x minus 2, where x is the position in the sequence. This is similar to the basic idea of a **recurrent neural network or 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. That becomes a little more clear when you chain them together.

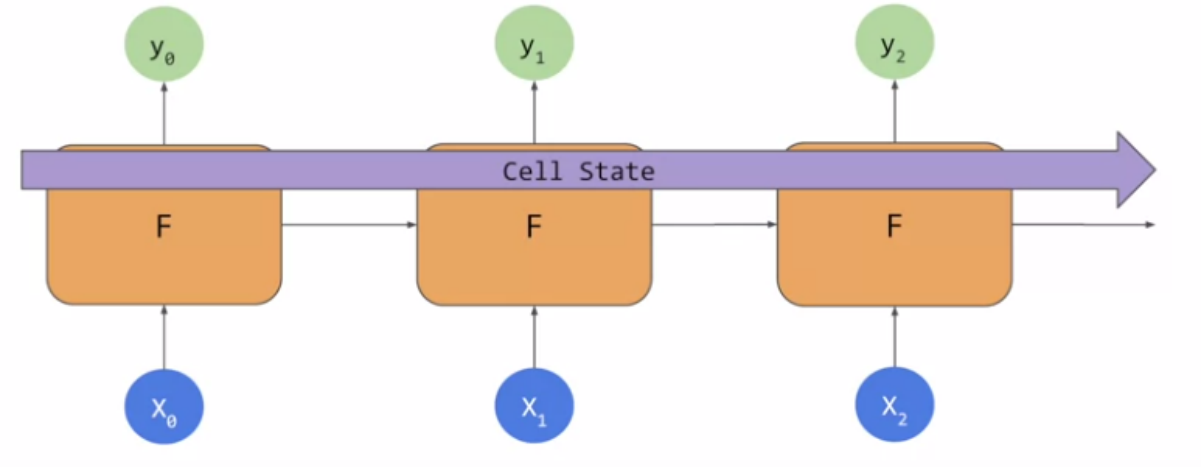


x_0 is fed into the function returning y_0. An output from the function is then fed into the next function, which gets fed into the function along with x_2 to get y_2, producing an output and continuing the sequence. As you can see, there's an element of x_0 fed all the way through the network, similar with x_1 and x_2 etc. This forms the basis of the recurrent neural network or RNN.

If you want to learn more about RNN and LSTM you can check those links: An Introduction to Recurrent Neural Networks & LSTMs , Long Short Term Memory (LSTM) and Deep RNNs

LSTMs

LSTMs, or Long Short-Term Memory Networks, are an update to RNNs which, in addition to the context being passed from RNNs, pass another pipeline of context called the **cell state.**



The cell state helps keep context from earlier tokens so that they can be accessed later on. Cell states can also be bidirectional, which means that later contexts can impact earlier ones.

You can also learn more about LSTMs from Andrew Ng's course on Sequence Models. Below is an example of a model that uses an LSTM in the second layer:

```
model = tf.keras.Sequential([
                            tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
                            tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
                            tf.keras.layers.Dense(64, activation='relu'),
                            tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Let's explain it line by line:

We added the second layer as an LSTM, to do so we use this line:

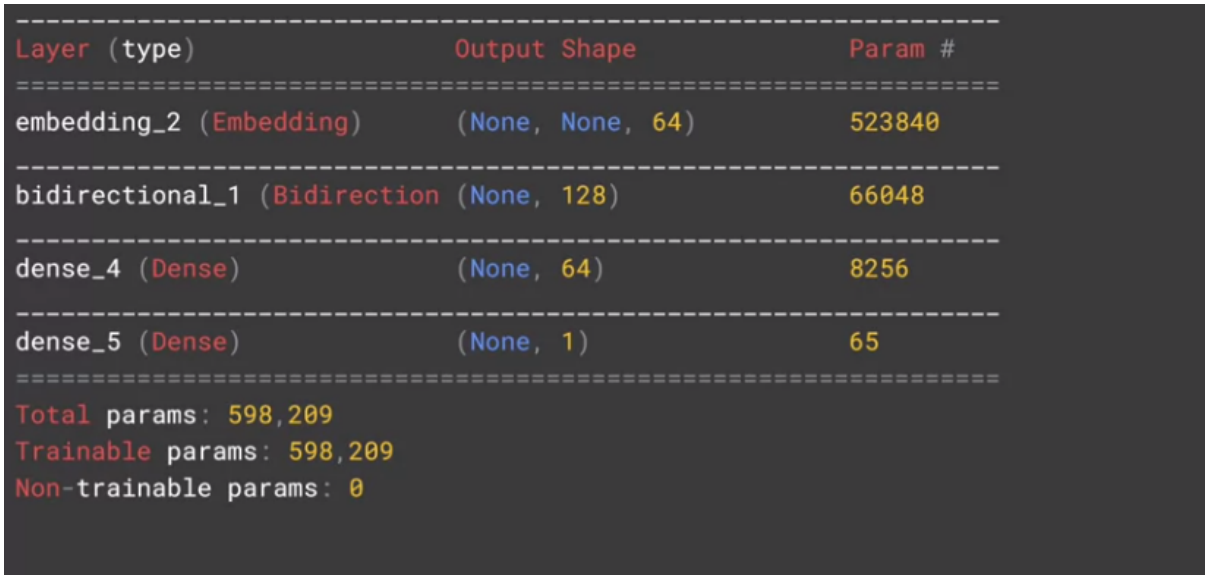
```
tf.keras.layers.LSTM(64)
```

The parameter passed in is the number of outputs that I desire from that layer, in this case it's 64.

If I wrap that with tf.keras.layers.Bidirectional, it will make my cell state go in both directions.

```
 tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64))
```

Let's explore the model summary:



We have our embedding and our bidirectional containing the LSTM, followed by the two dense layers. If you notice the output from the bidirectional is now a 128, even though we told our LSTM that we wanted 64, the bidirectional doubles this up to a 128.

You can also stack LSTMs like any other keras layer by using code like this:

```
model = tf.keras.Sequential([
                            tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
                            tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, retur
                            tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
                            tf.keras.layers.Dense(64, activation='relu'),
                            tf.keras.layers.Dense(1, activation='sigmoid')
])
```

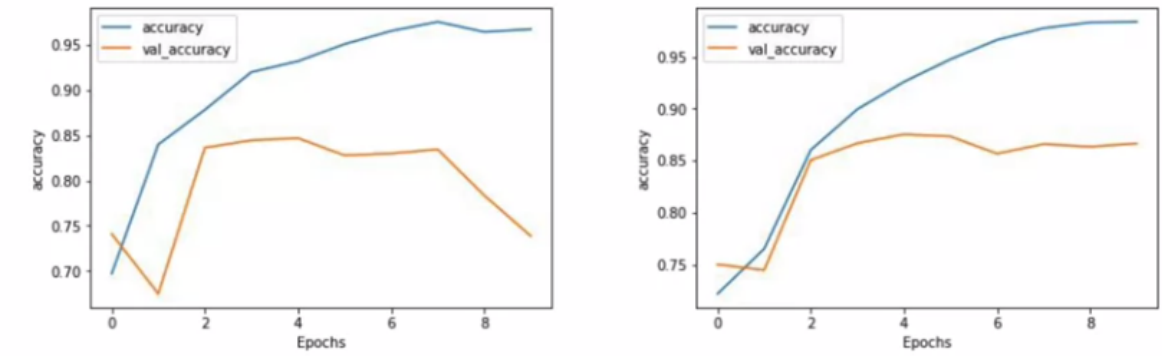
But when you feed an LSTM into another one, you do have to put the **return sequences** equal true parameter into the first one. This ensures that the outputs of the LSTM match the desired inputs of the next one. The summary of the model will look like this:

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 64)          523840
_________________________________________________________________
bidirectional_2 (Bidirection (None, None, 128)         66048
_________________________________________________________________
bidirectional_3 (Bidirection (None, 64)                41216
_________________________________________________________________
dense_6 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 65
=================================================================
Total params: 635,329
Trainable params: 635,329
Non-trainable params: 0
```

Let's look at the impact of using an LSTM on the model that we looked at in the last module, where we had subword tokens.

Accuracy and Loss

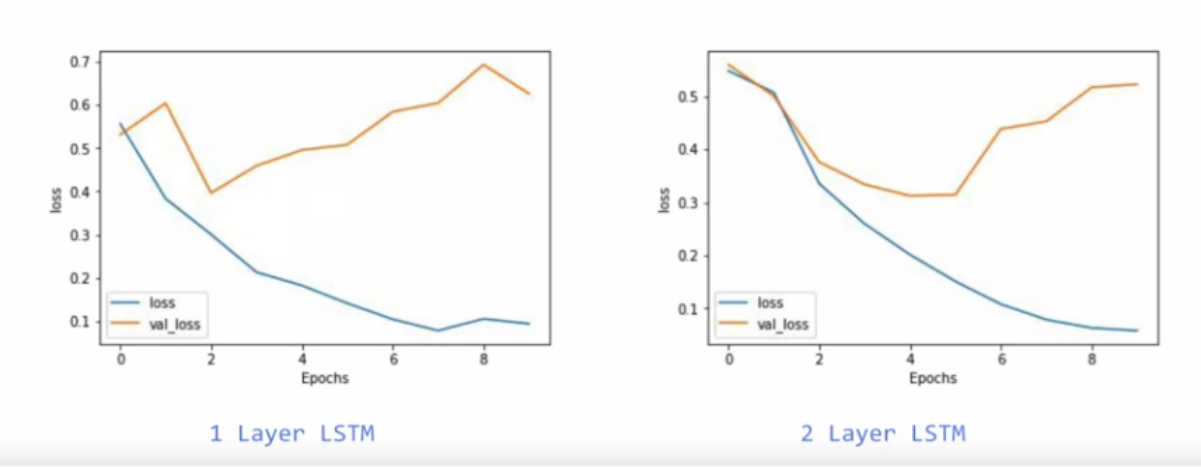
10 Epochs: Accurancy Measurement



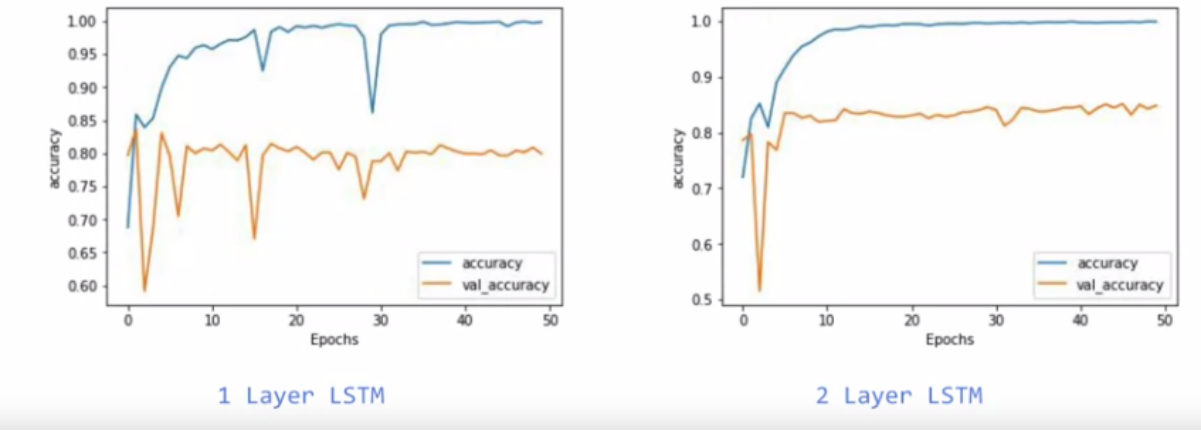
1 Layer LSTM 2 Layer LSTM

Here's the comparison of accuracies between the one layer LSTM and the two layer one over 10 epochs. There's not much of a difference except the nosedive and the validation accuracy. But notice how the training curve is smoother. I found from training networks that jaggedness can be an indication that your model needs improvement, and the single LSTM that you can see here is not the smoothest.

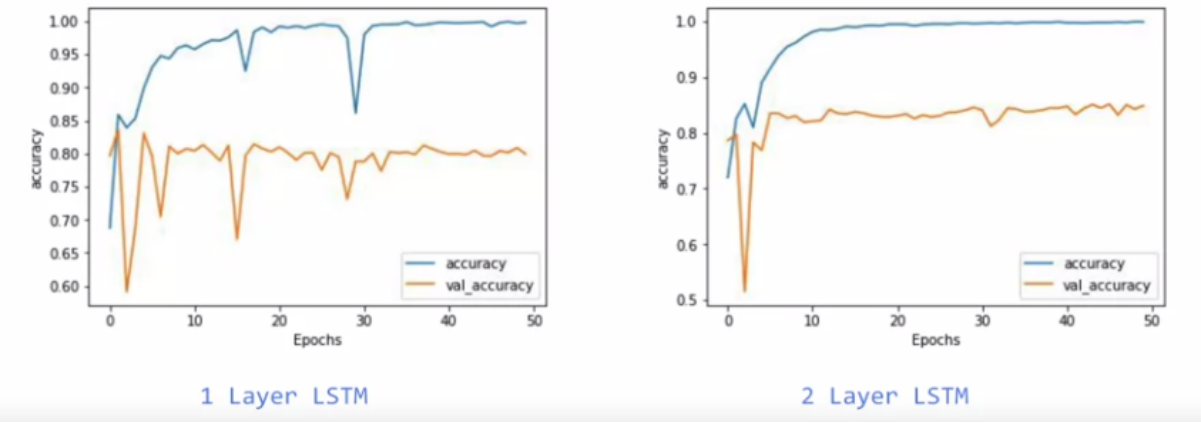
10 Epochs: Loss Measurement



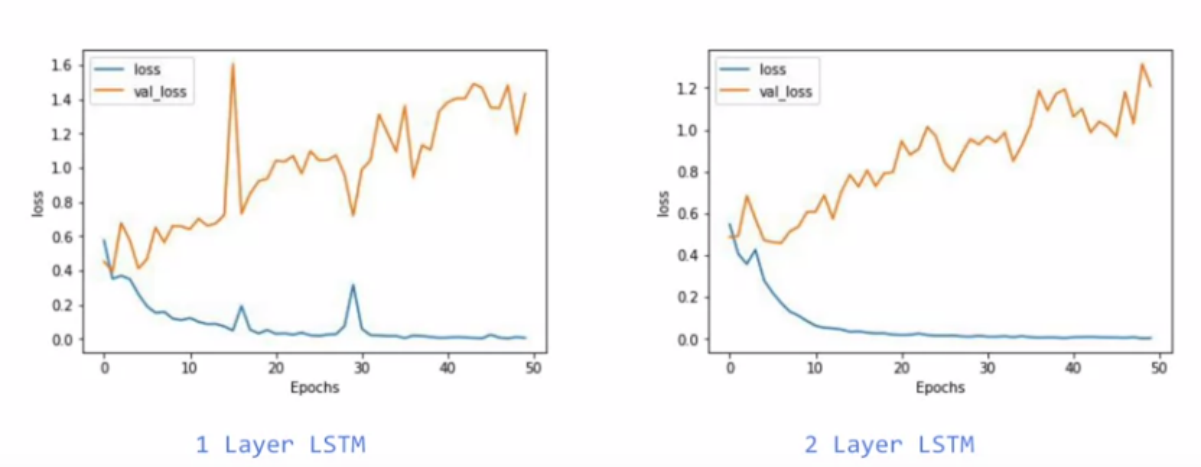
10 Epochs: Accurancy Measurement



50 Epochs: Accurancy Measurement



50 Epochs: Loss Measurement

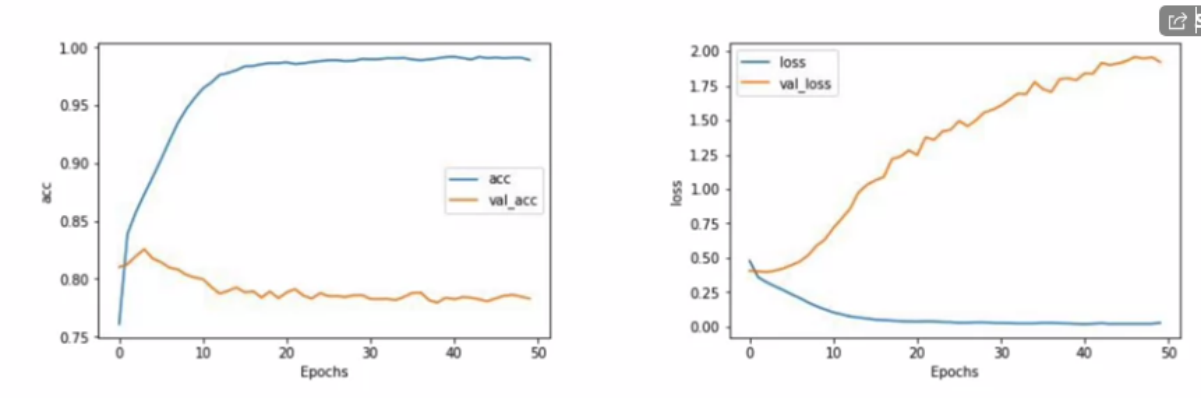


Using a convolutional network

Another type of layer that you can use is a convolution, in a very similar way to what you did with images. You specify the number of convolutions that you want to learn, their size, and their activation function. The effect of this will then be the same. Now words will be grouped into the size of the filter in this case 5. And convolutions will learned that can map the word classification to the desired output.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              input_length=max_length),
    tf.keras.layers.Conv1D(128, 5, activation='relu'),
    tf.keras.layers.GlobalMaxPooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

If we train with the convolutions now, we will see that our accuracy does even better than before with close to about 100% on training and around 80% on validation.



But as before, our loss increases in the validation set, indicating potential overfilling. As I have a super simple network here, it's not surprising, and it will take some experimentation with different combinations of conversational layers to improve on this. If we go back to the model and explore the parameters, we'll see that we have 128 filters each for 5 words.

```
max_length = 120

tf.keras.layers.Conv1D(128, 5, activation='relu'),


Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 120, 16)           16000
_________________________________________________________________
conv1d (Conv1D)              (None, 116, 128)          10368
_________________________________________________________________
global_max_pooling1d (Global (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 24)                3096
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 25
=================================================================
Total params: 29,489
Trainable params: 29,489
Non-trainable params: 0
```

And an exploration of the model will show these dimensions. As the size of the input was 120 words, and a filter that is 5 words long will shave off 2 words from the front and back, leaving us with 116. The 128 filters that we specified will show up here as part of the convolutional layer.