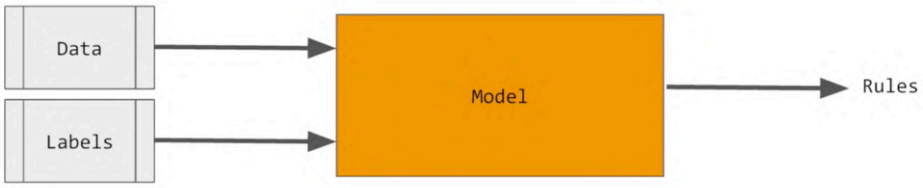
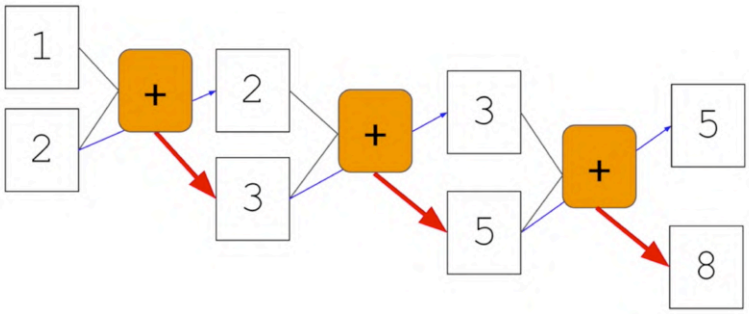
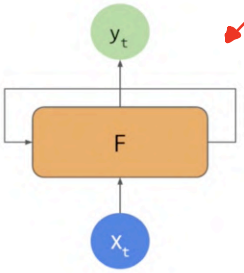


1	2	3	5	8	13	21	34	55	89

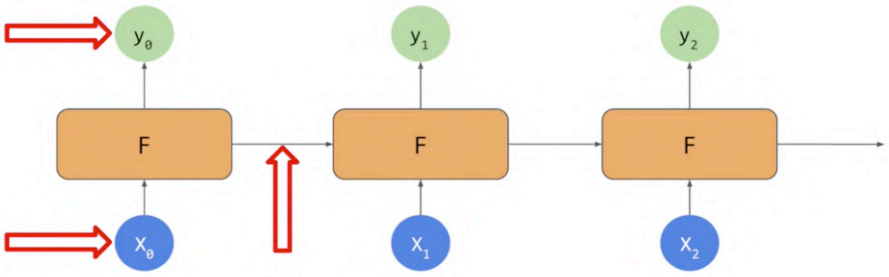
n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9

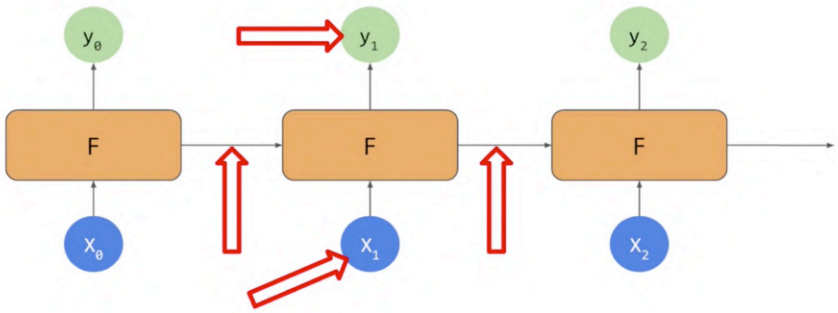
$$n_x = n_{x-1} + n_{x-2}$$

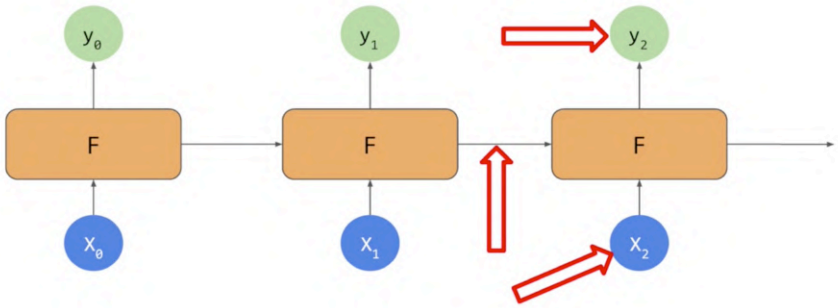


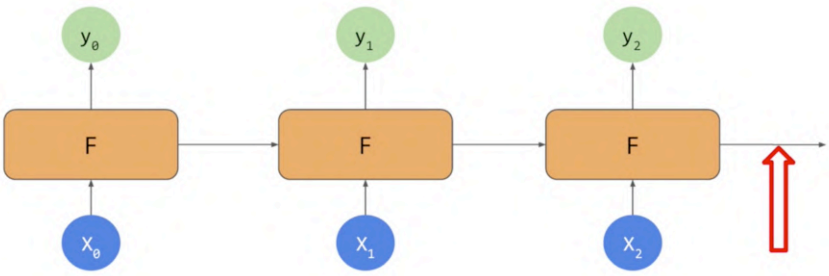


The basic idea of a recurrent neural network or RNN, which is often drawn a little like this. You have your x as in input and your y as an output. But there's also an element that's fed into the function from a previous function.




y0
y1
y2
F
F
F
x0
x1
x2


y0
y1
y2
F
F
F
x0
x1
x2


y0
y1
y2
F
F
F
x0
x1
x2

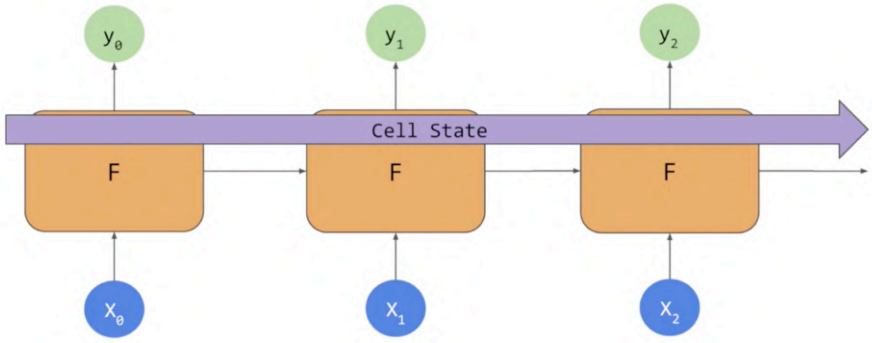
Today has a beautiful blue <...>

Today has a beautiful blue sky

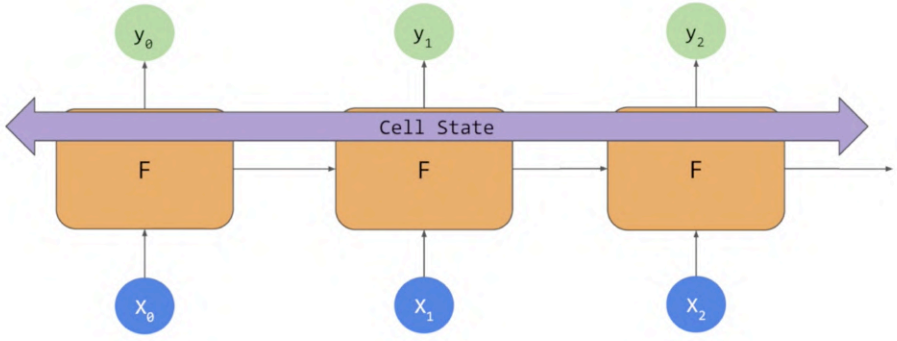
I lived in Ireland, so at school they made me learn how to speak <...>

I lived in Ireland, so at school they made me learn how to speak Gaelic

An update to RNNs is called LSTM, long short - term memory has been created. In addition to the context being PaaSed 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



Cell states can also be bidirectional. So later contexts can impact earlier ones as we'll see when we look at the code.



Here's my model where I've added the second layer as an LSTM. I use the tf.keras.layers.LSTM to do so. The parameter passed in is the number of outputs that I desire from that layer, in this case it's 64.

```
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')
])
```

If I wrap that with tf.keras.layers.Bidirectional, it will make my cell state go in both directions. You'll see this when you explore the model summary, which looks like this.

```
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')
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, None, 64)          523840
_________________________________________________________________
bidirectional_1 (Bidirection (None, 128)               66048
_________________________________________________________________
dense_4 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 65
=================================================================
Total params: 598,209
Trainable params: 598,209
Non-trainable params: 0
```

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.

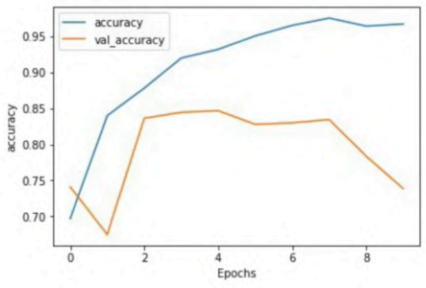
But when you feed an LSTM into another one, you do have to put the return sequences equal true parameter into the first one. This ensures that the outputs of the LSTM match the desired inputs of the next one.

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

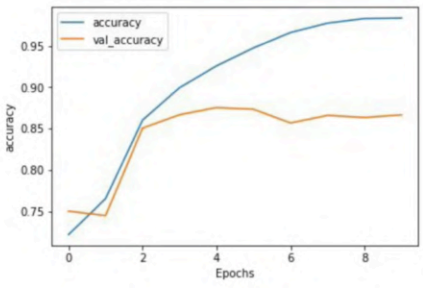
```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 64)          523840
_________________________________________________________________
bidirectional_2 (Bidirection (None, None, 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
```

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 : Accuracy Measurement



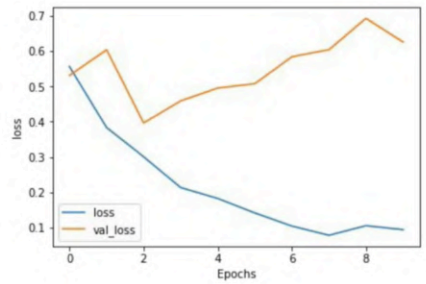
1 Layer LSTM



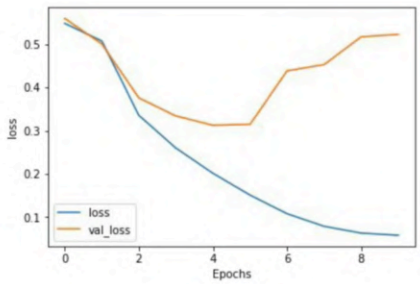
2 Layer LSTM

10 Epochs : Loss Measurement

If you look at loss, over the first 10 epochs, we can see similar results.



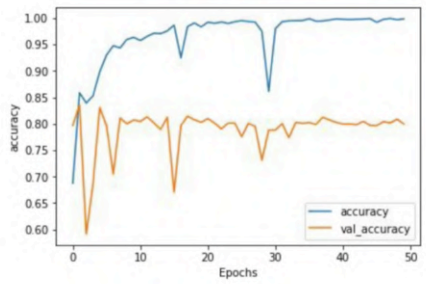
1 Layer LSTM



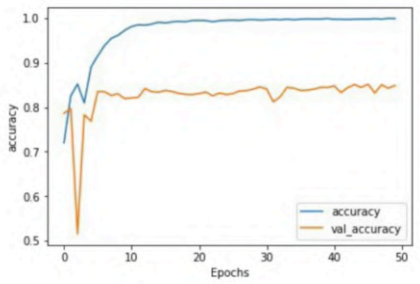
2 Layer LSTM

50 Epochs : Accuracy Measurement

Look what happens when we increase to 50 epochs training. Our one layer LSTM, while climbing in accuracy, is also prone to some pretty sharp dips. The final result might be good, but those dips makes me suspicious about the overall accuracy of the model. Our two layer one looks much smoother, and as such makes me much more confident in its results. Note also the validation accuracy. Considering it levels out at about 80 percent, it's not bad given that the training set and the test set were both 25,000 reviews. But we're using 8,000 sub-words taken only from the training set. So there would be many tokens in the test sets that would be out of vocabulary. Yet despite that, we are still at about 80 percent accuracy.



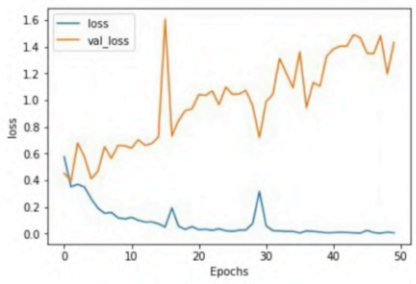
1 Layer LSTM



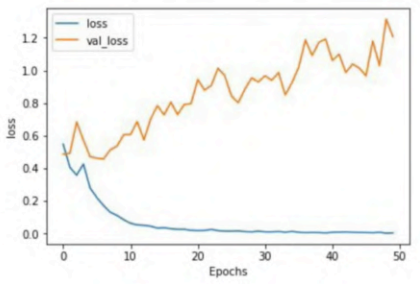
2 Layer LSTM

50 Epochs : Loss Measurement

Our loss results are similar with the two layer having a much smoother curve. The loss is increasing epoch by epoch. So that's worth monitoring to see if it flattens out in later epochs as would be desired.



1 Layer LSTM



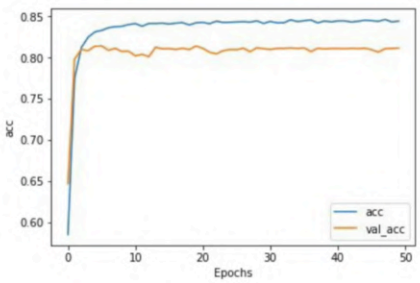
2 Layer LSTM

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              input_length=max_length),
    tf.keras.layers.Flatten(),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

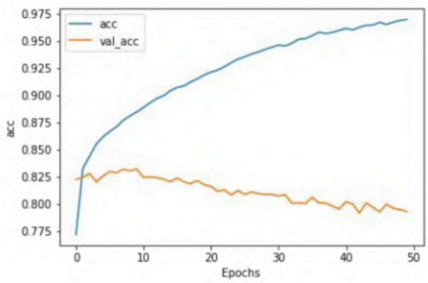
we can experiment with the layers that bridge the embedding and the dense by removing the flatten and puling from here, and replacing them with an LSTM like this

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              input_length=max_length),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

I quickly got close to 85 percent accuracy and then it flattened out there. The validation set was a little less accurate, but the curves we're quite in sync. On the other hand, when using LSTM, I reached 85 percent accuracy really quickly and continued climbing towards about 97.5 percent accuracy within 50 epochs. The validation set dropped slowly, but it was still close to the same value as the non- LSTM version. Still the drop indicates that there's some over fitting going on here. So a bit of tweaking to the LSTM should help fix that.

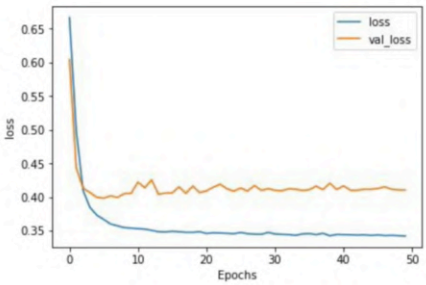


Without LSTM

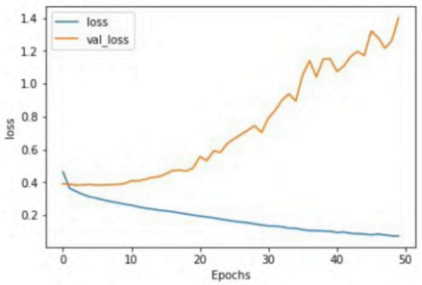


With LSTM

Similarly, the loss values from my non-LSTM one got to healthy state quite quickly and then flattened out. Whereas with the LSTM, the training loss drop nicely, but the validation one increased as I continue training. Again, this shows some over fitting in the LSTM network. While the accuracy of the prediction increased, the confidence in it decreased. So you should be careful to adjust your training parameters when you use different network types, it's not just a straight drop-in like I did here.



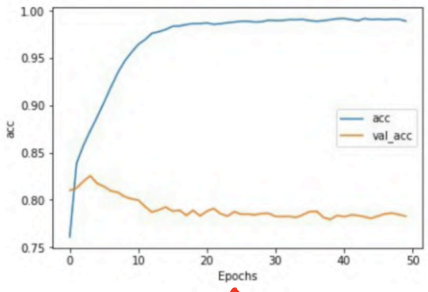
Without LSTM



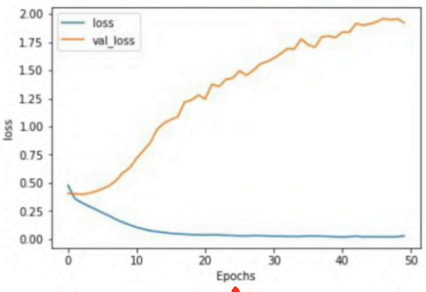
With LSTM

```
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')
])
```

The code to use a convolutional on network is here. It's very similar to what you had before. 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.



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



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

```
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')
])
```

```
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
```

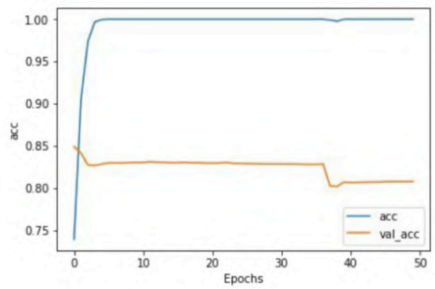
```
max_length = 120

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

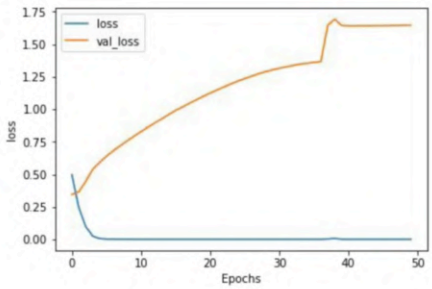
```
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 120, 16)           16000
_________________________________________________________________
conv1d (Conv1D)              (None, 116, 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
```

If we go back to the model and explore the parameters, we'll see that we have 128 filters each for 5 words. 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.

```
imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)



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')
])

model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 120, 16)           160000
_________________________________________________________________
flatten (Flatten)            (None, 1920)              0
_________________________________________________________________
dense (Dense)                (None, 6)                 11526
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 7
=================================================================
Total params: 171,533
Trainable params: 171,533
Non-trainable params: 0
```





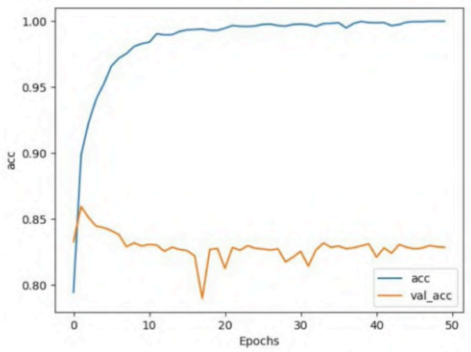
Nice accuracy, but clear overfitting but it only takes about five seconds per epoch to train.

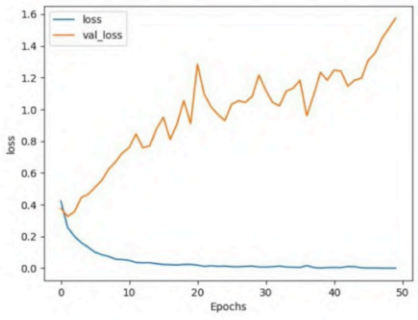
IMDB with Embedding-only : ~ 5s per epoch

```python
imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)



# Model Definition with LSTM
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_7 (Embedding)      (None, 120, 16)           16000
_________________________________________________________________
bidirectional_7 (Bidirection (None, 64)                12544
_________________________________________________________________
dense_14 (Dense)             (None, 24)                1560
_________________________________________________________________
dense_15 (Dense)             (None, 1)                 25
=================================================================
Total params: 30,129
Trainable params: 30,129
Non-trainable params: 0
```

less



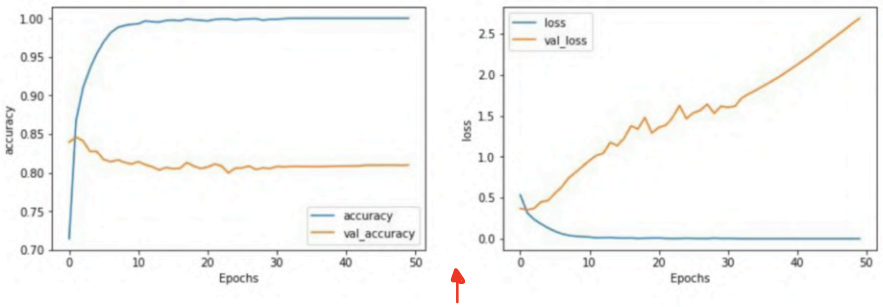


IMDB with LSTM ~43s per epoch

Accuracy is better, but there's still some overfitting

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(32)),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

model.summary()
```

Try a GRU layer instead, with a GRU being a different type of RNN, and I make it bidirectional



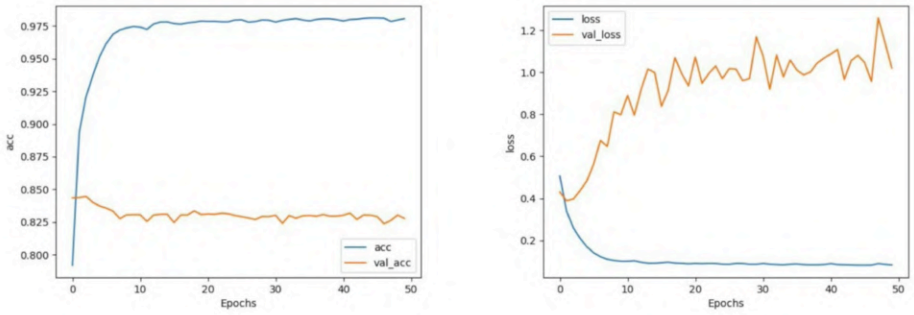
Accuracy is again very good on training, and not too bad on validation but again, showing some overfitting.

IMDB with GRU : ~ 20s per epoch

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 120, 16)           160000
_________________________________________________________________
conv1d (Conv1D)              (None, 116, 128)          10368
_________________________________________________________________
global_average_pooling1d (Gl (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 6)                 774
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 7
=================================================================
Total params: 171,149
Trainable params: 171,149
Non-trainable params: 0
_________________________________________________________________
```

Convolutional Layer

Close to 100 percent accuracy on training, and about 83 percent on validation, but again with overfitting.



IMDB with CNN : ~ 6s per epoch

Remember that with text, you'll probably get a bit more overfitting than you would have done with images. Not least because you'll almost always have out of vocabulary words in the validation data set. That is words in the validation dataset that weren't present in the training, naturally leading to overfitting. These words can't be classified and, of course, you're going to have these overfitting issues, but see what you can do to avoid them.