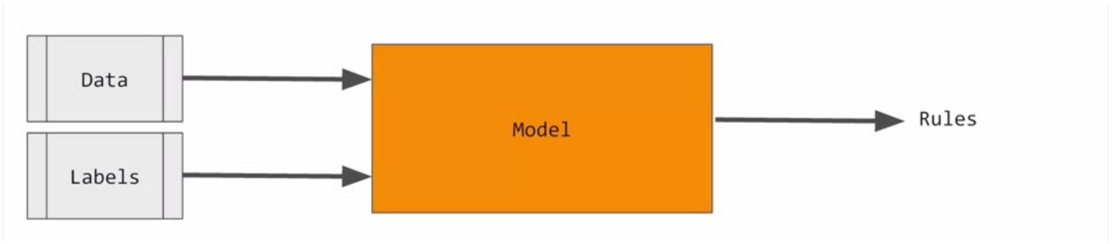
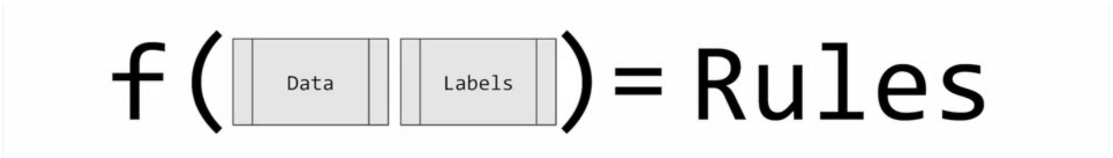
3.1 Introduction

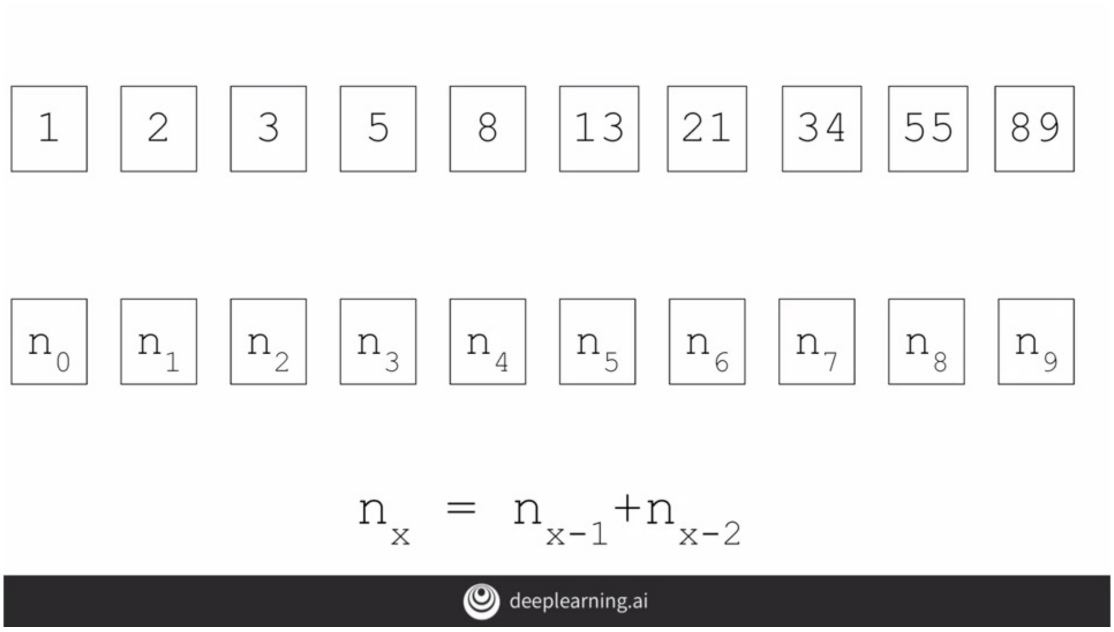
Last week, we looked at doing classification using texts and trying to train and understand positive and negative sentiment in movie reviews. We finished by looking at the effect of tokenizing words, and saw that our classifier failed to get any meaningful results. The main reason for this was that the context of words was 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.



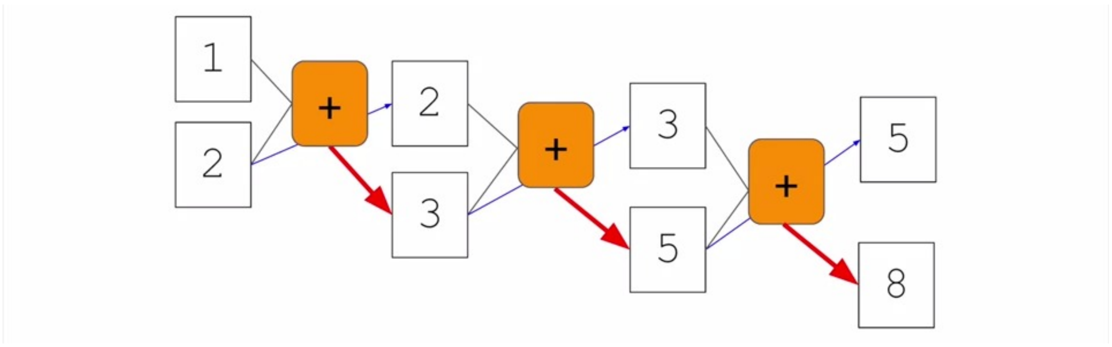
The neural network is like a function that when you feed it in data and labels, it infers the rules from these, and then you can use those rules.



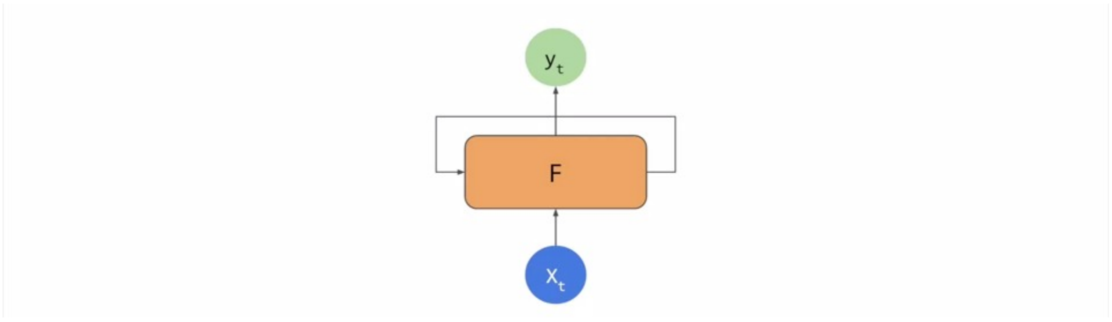
So it could be seen as a function a little bit like this, you take the data and you take the labels, and you get the rules. But this doesn't take any kind of sequence into account. To understand why sequences can be important, consider this set of numbers.


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
$n_x = n_{x-1} + n_{x-2}$
deeplearning.ai

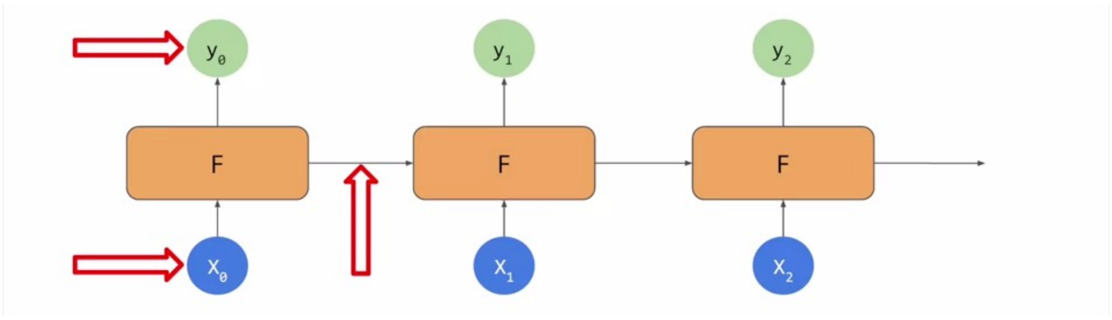
If you've never seen them before, they're called the Fibonacci sequence. So let's replace the actual values with variables such as n_0, n_1 and n_2, etc., to denote them. Then the sequence itself can be derived where a number is the sum of the two numbers before it. So 3 is 2 plus 1, 5 is 2 plus 3, 8 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.



Visualized, it might also look like this, one and two feed into the first function and three comes out. Two gets carried over to the next, where it's fed in along with the three to give us a five. The three is carried on to the next where it's fed into the function along with the five to get an eight and so on.



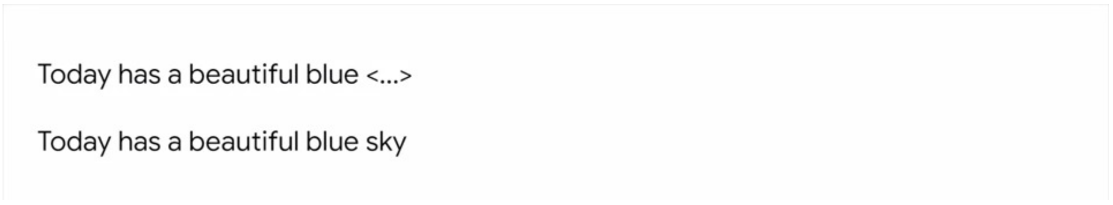
This is similar to the basic idea of a recurrent neural network or RNN, which is often drawn a little like this. You have your x as in input and your y as an output. But there's also an element that's fed into the function from a previous function.



That becomes a little more clear when you chain them together like this, 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. I'm not going to go into detail and how they work, but you can learn much more about them at this course from Andrew.

3.2 LSTMs

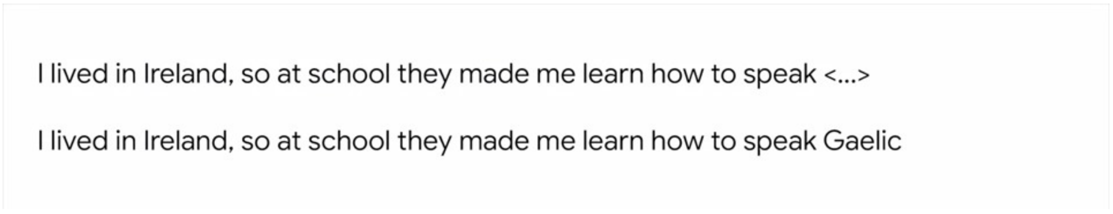
There can be a limitation when approaching text classification in this way. Consider the following. Here's a sentence.



Today has a beautiful blue <...>

Today has a beautiful blue sky

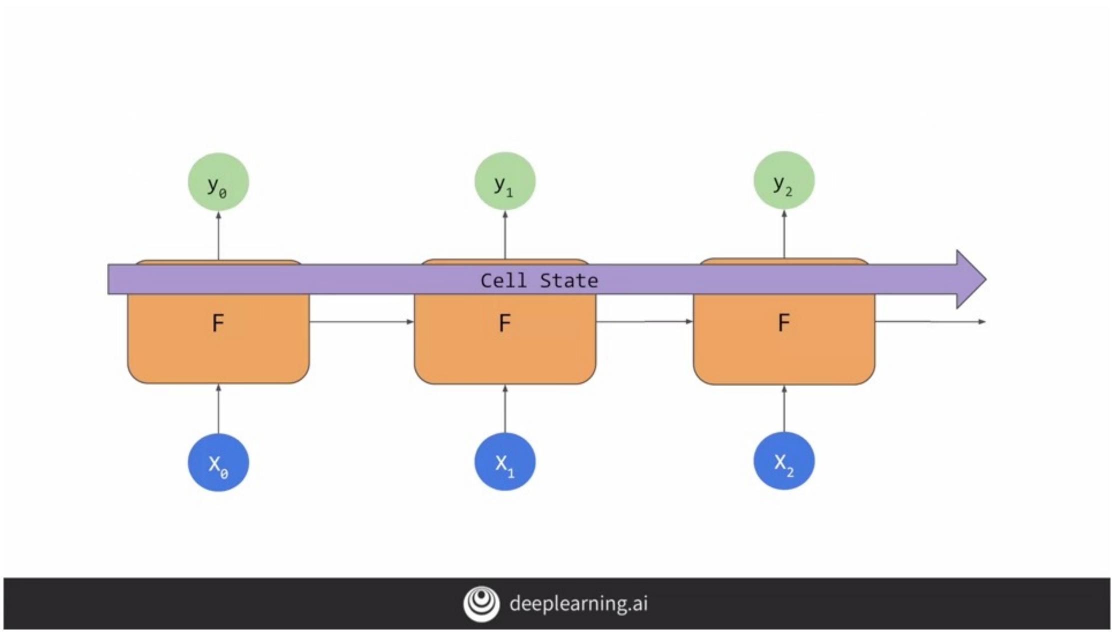
Today has a beautiful blue. What do you think would come next? Probably sky. Right? Today has a beautiful blue sky. Why would you say that? Well, there's a big clue in the word blue. In a context like this, it's quite likely that when we're talking about a beautiful blue something, we mean a beautiful blue sky. So, the context word that helps us understand the next word is very close to the word that we're interested in.



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

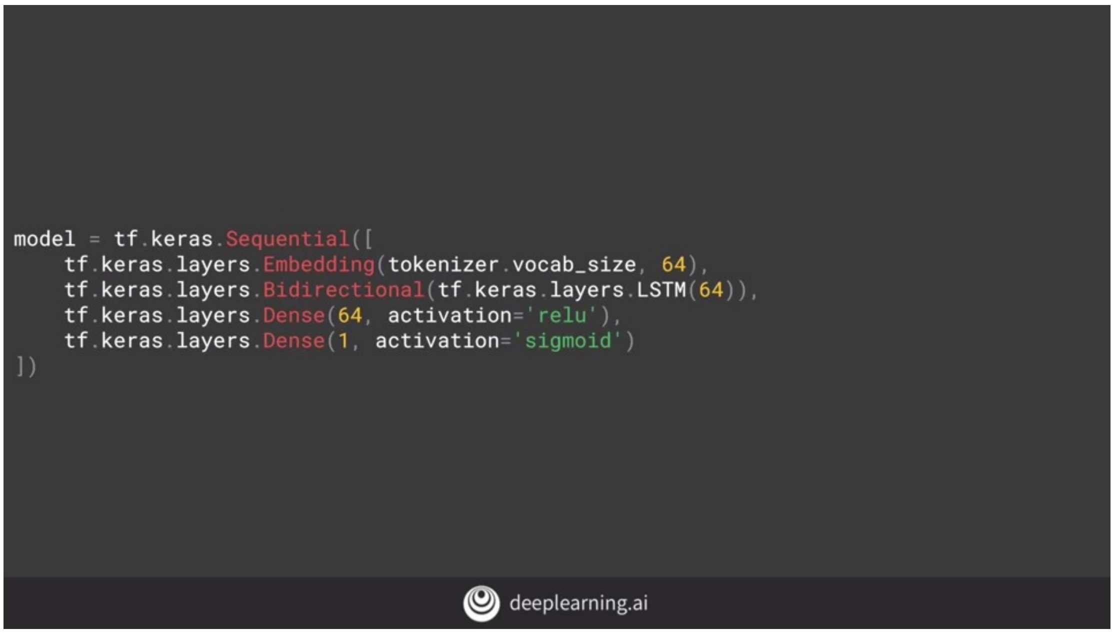
But, what about a sentence like this, I lived in Ireland so at school they made me learn how to speak something. How would you finish that sentence? Well, you might say Irish but you'd be much more accurate if you said, I lived in Ireland so at school they made me learn how to speak Gaelic. First of course, is the syntactic issue. Irish describes the people, Gaelic describes the language. **But more importantly 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. 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 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 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.

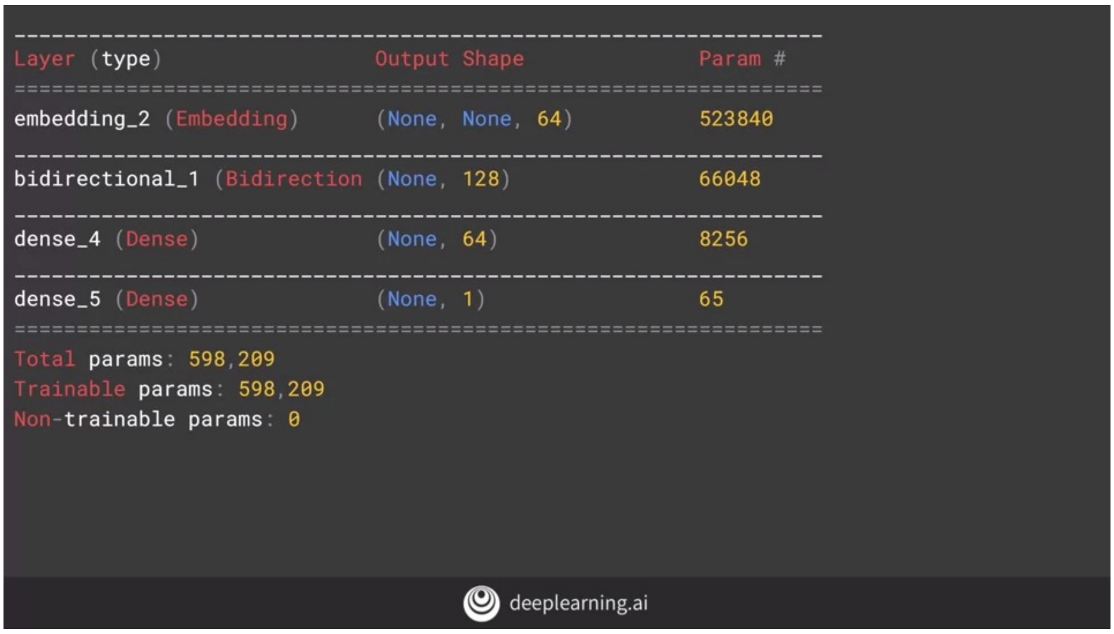
3.3 Implementing LSTMs in code

So let's now take a look at how to implement LSTMs in code.



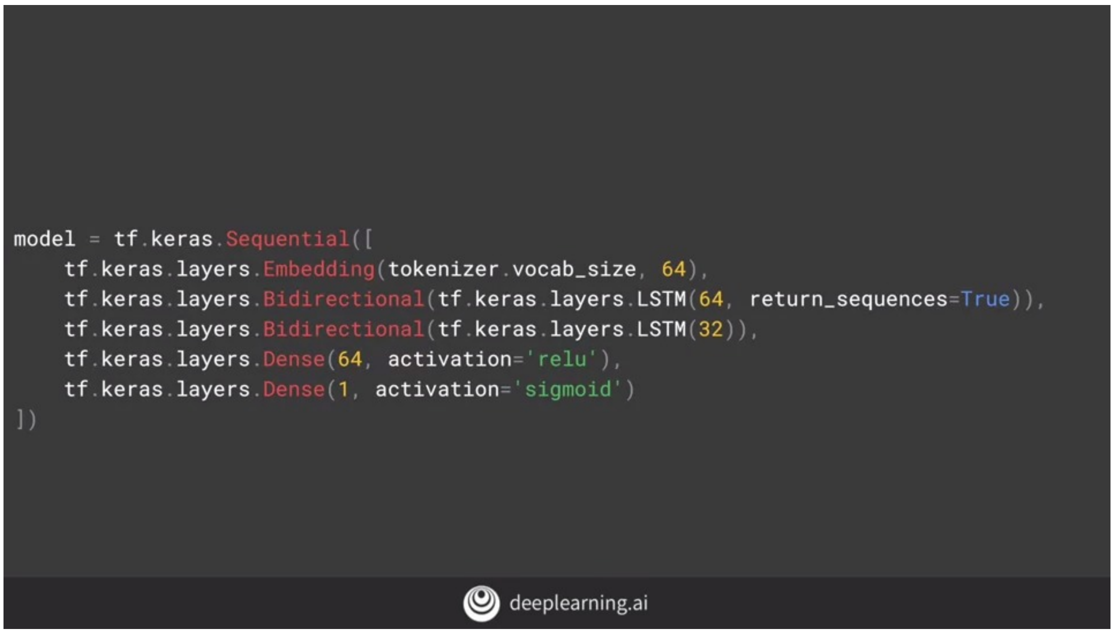
```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

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



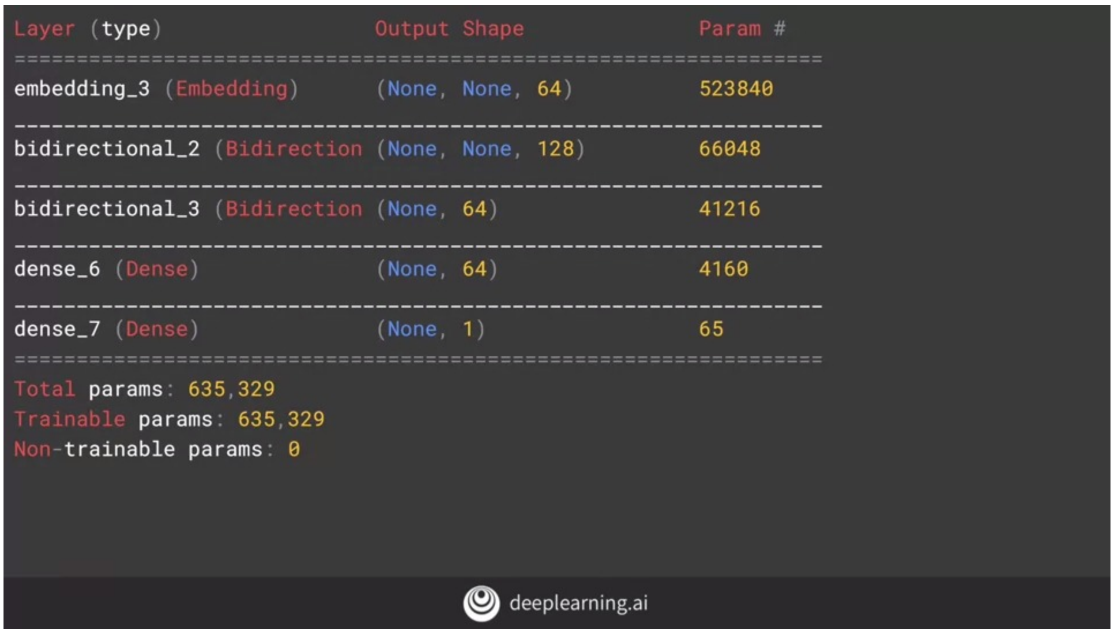
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, None, 64)          523840
_________________________________________________________________
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
```

You'll see this when you explore the model summary, which looks like this. 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. Y



```
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')
])
```

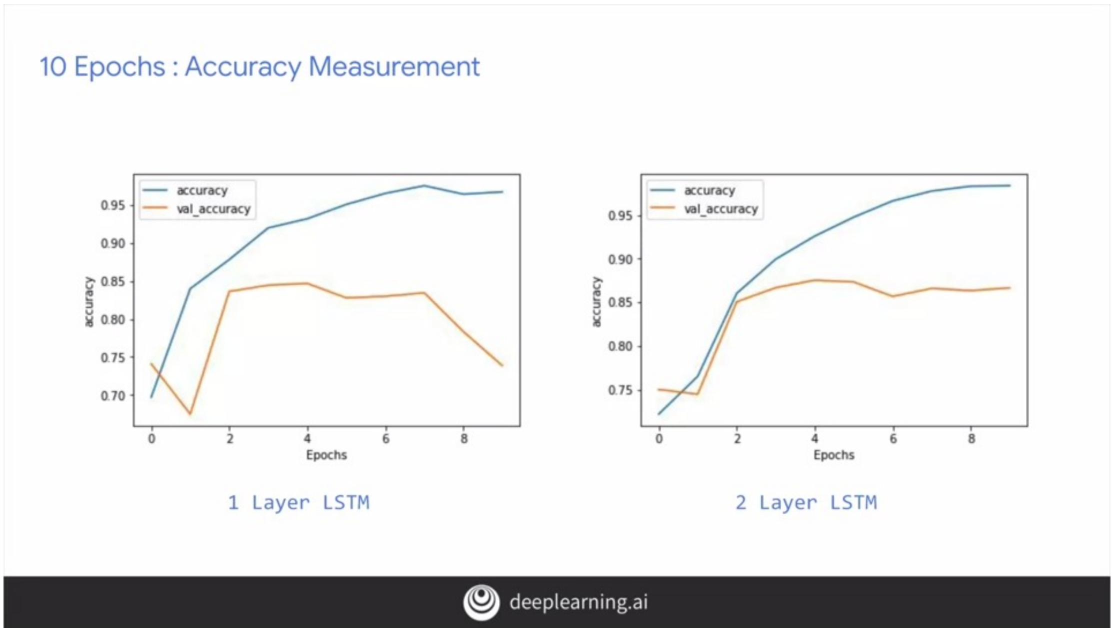
You can also stack LSTMs like any other keras layer by using code like this. But when you feed an LSTM into another one, you do have to put the return sequences equal true parameter into the first one. This ensures that the outputs of the LSTM match the desired inputs of the next one.



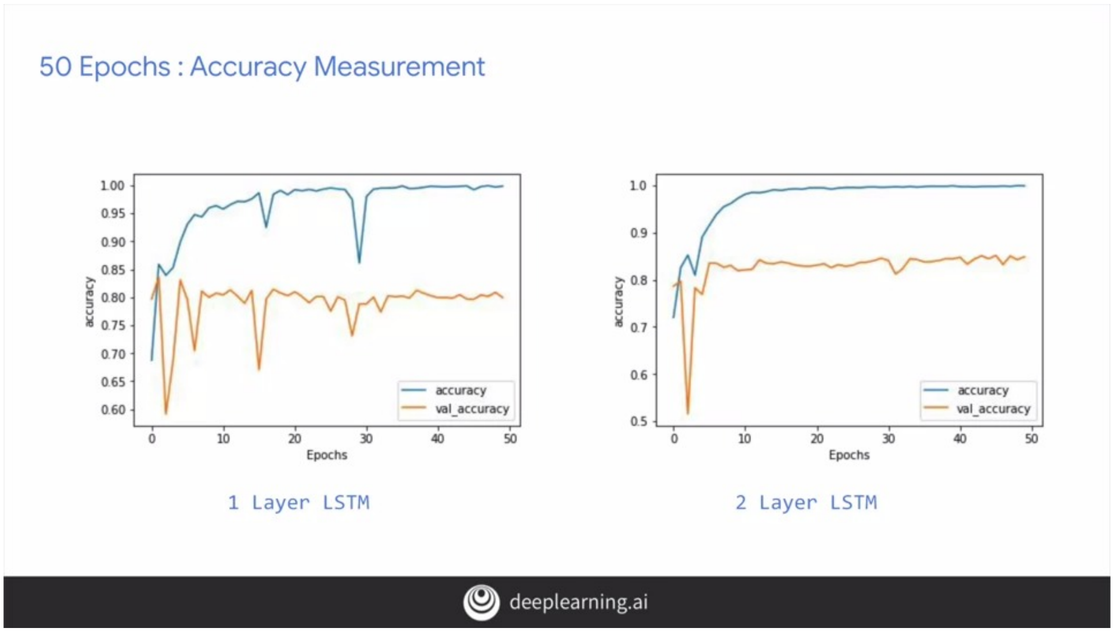
```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 64)          523840
_________________________________________________________________
bidirectional_2 (Bidirection (None, None,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

The summary of the model will look like this. 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.

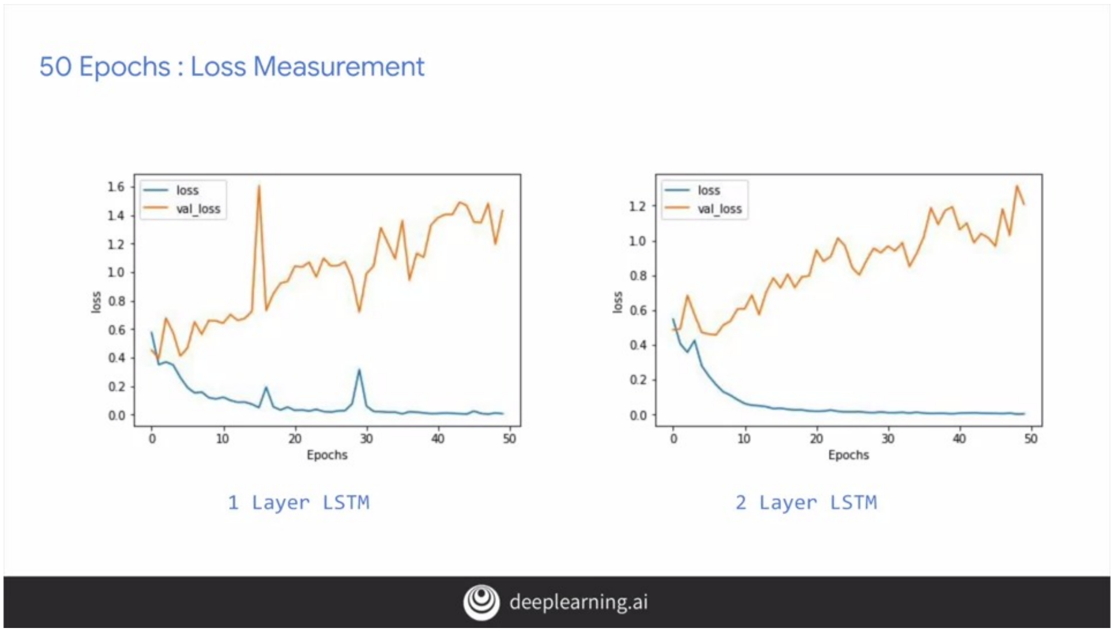
3.4 Accuracy and loss



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



But 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.



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. I hope this was a good introduction into how RNNs and LSTMs can help you with text classification. Their inherent sequencing is great for predicting unseen text if you want to generate some, and we'll see that next week. But first, I'd like to explore some other RNN types, and you'll see those in the next video.

3.5 A word from Laurence

In the last video we saw LSTMs and how they work with cell state to help keep context in a way that helps with understanding language. Well, words that aren't immediate neighbors can affect each other's context.
In this video, you'll see some other options of RNN including convolutions, **Gated Recurrent Units also called GRUs**, and more on how you can write the code for them. You'll investigate the impact that they have on training. I'm not going to go into depth on how they work, and that information is available in the deep learning specialization from Andrew. So do check it out there.

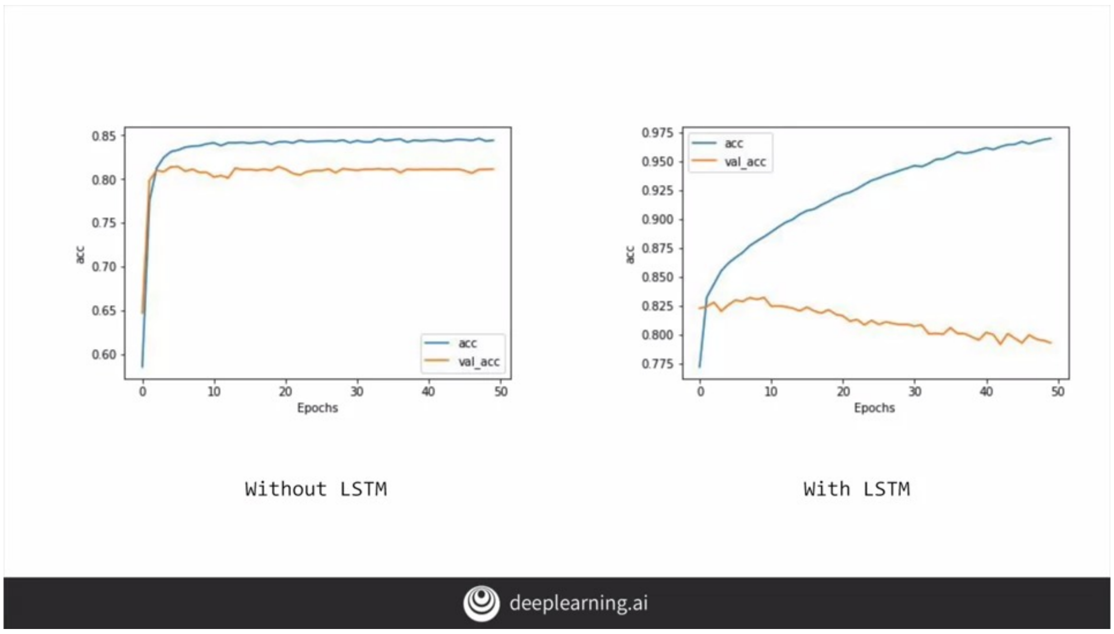
3.6 Looking into the code

```
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')
])
```

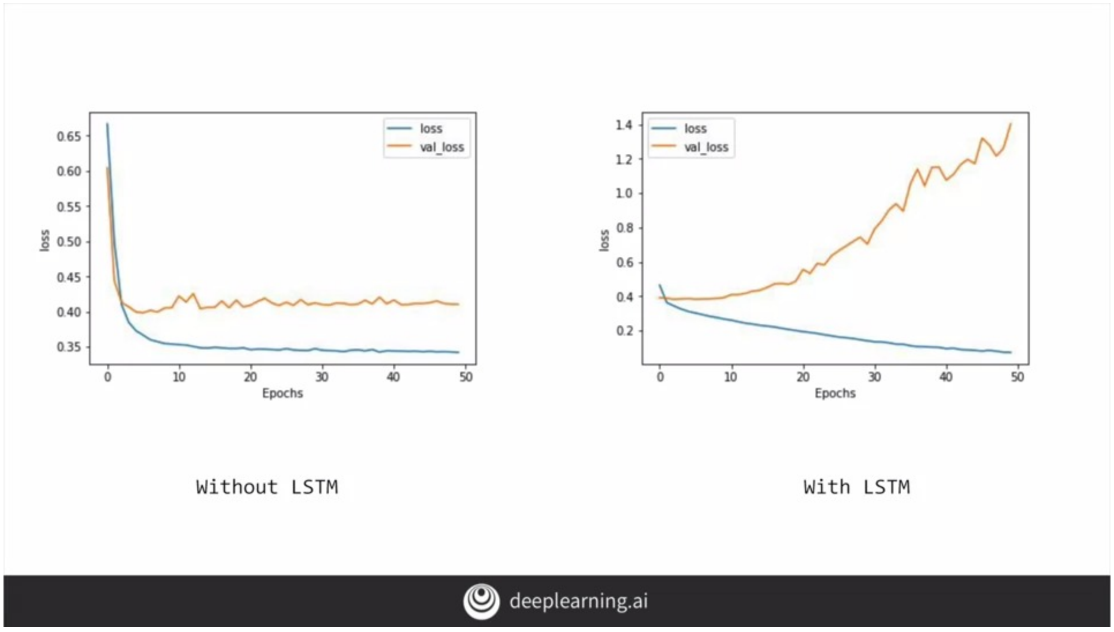
So let's start with this basic neural network. It has an embedding taking my vocab size, embedding dimensions, and input length as usual. The output from the embedding is flattened, averaged, and then fed into a dense neural network.

```
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')
])
```

But 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.



For a trainee using the sarcasm data-set with these, when just using the pooling and flattening, 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.

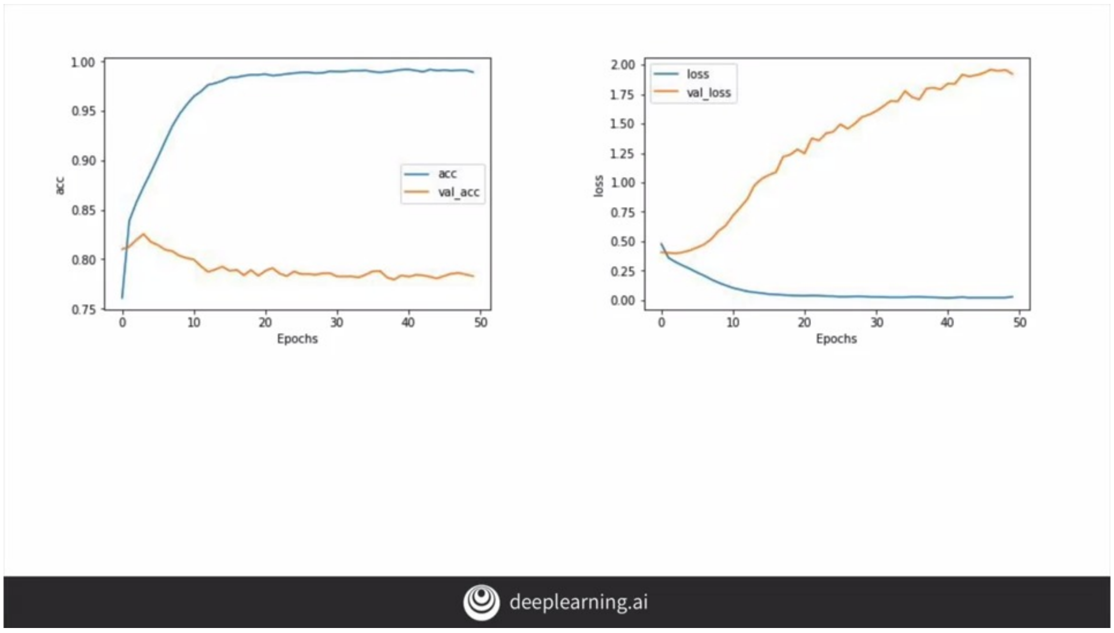


Similarly, the loss values from my non-LSTM one got to healthy state quite quickly and then flattened out. Whereas with the LSTM, the training loss drop nicely, but the validation one increased as I continue training. Again, this shows some over fitting in the LSTM network. While the accuracy of the 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.

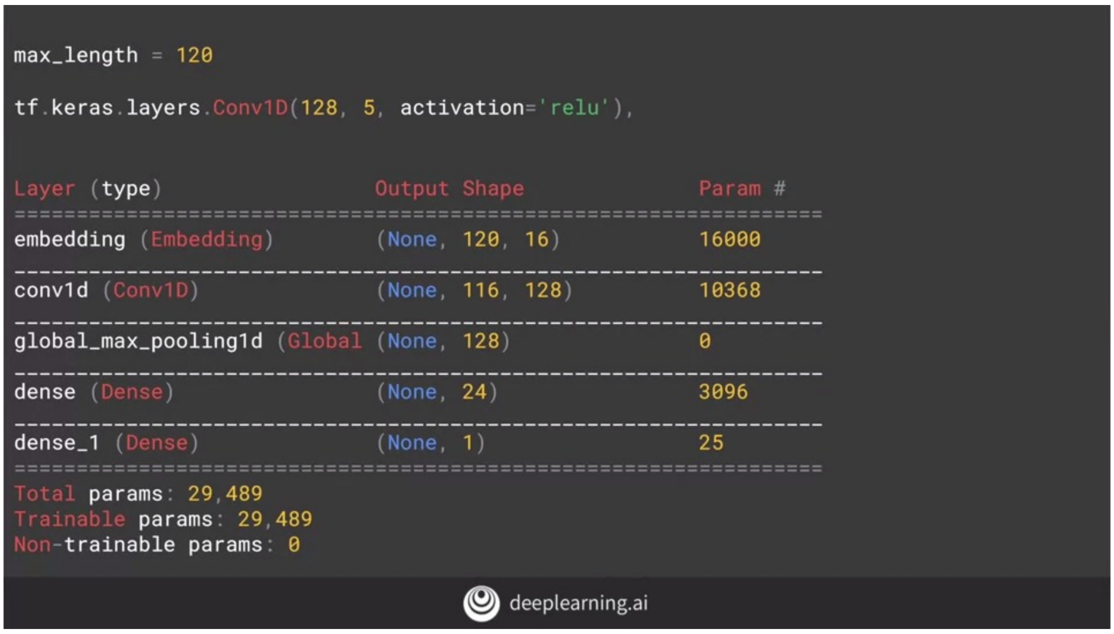
3.7 Using a convolutional network

```
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')
])
```

Another type of layer that you can use is a convolution, in a very similar way to what you did with images. 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. 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.

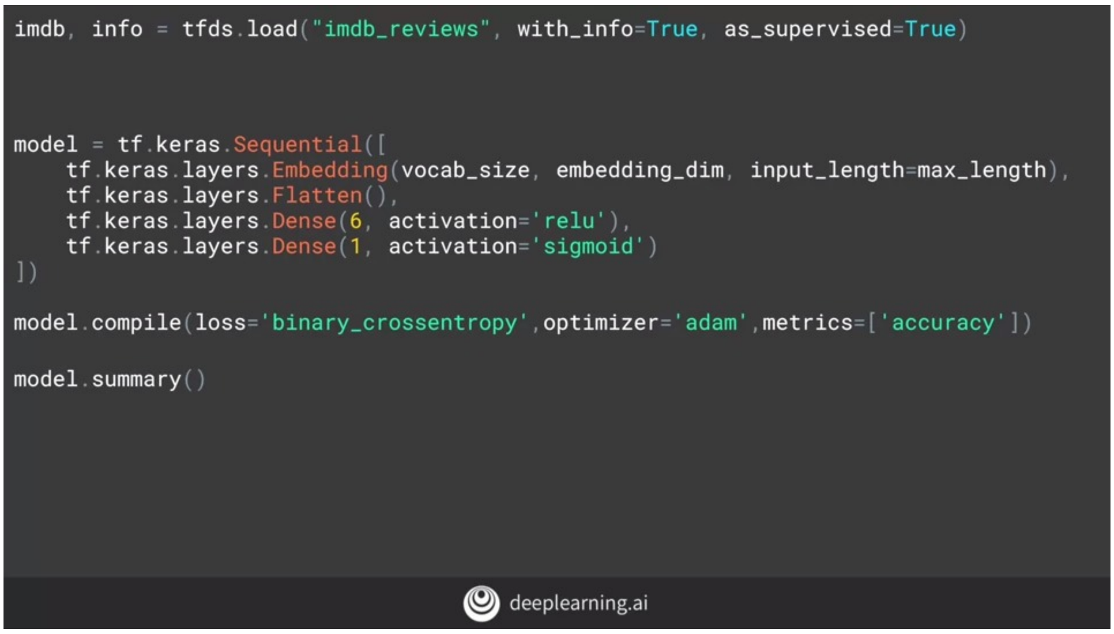


```
max_length = 120

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
```

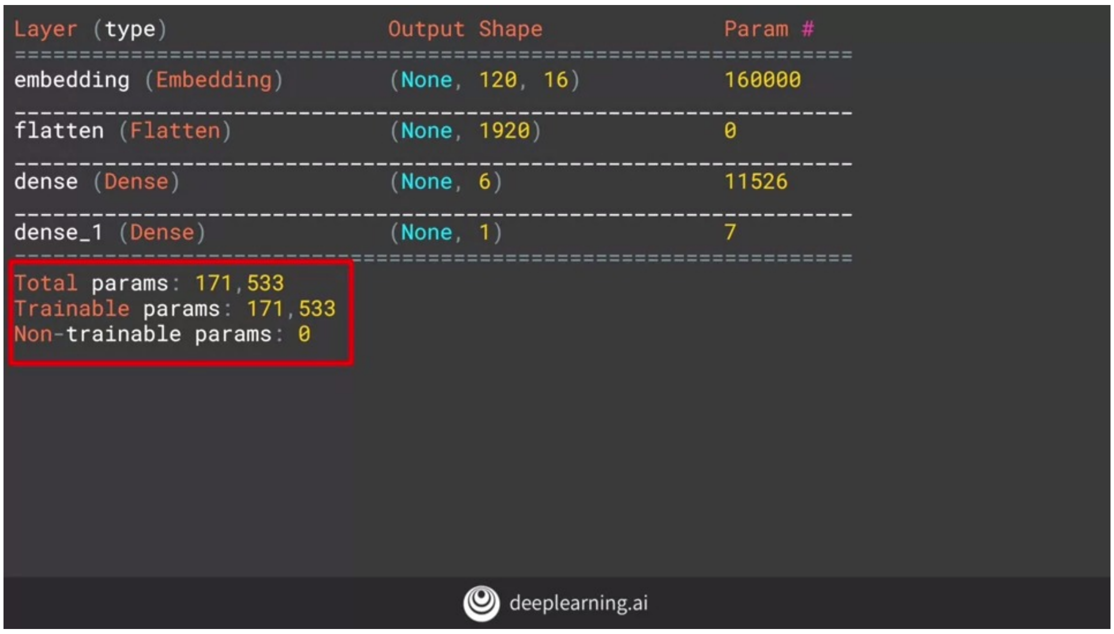
If we go back to the model and explore the parameters, we'll see that we have 128 filters each for 5 words. And an exploration of the model will show these dimensions. As the size of the input was 120 words, and a filter that is 5 words long will shave off 2 words from the front and back, leaving us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

3.8 Going back to the IMDB dataset

So now that we've seen these, let's go back to the IMDB dataset that we used earlier in this course.

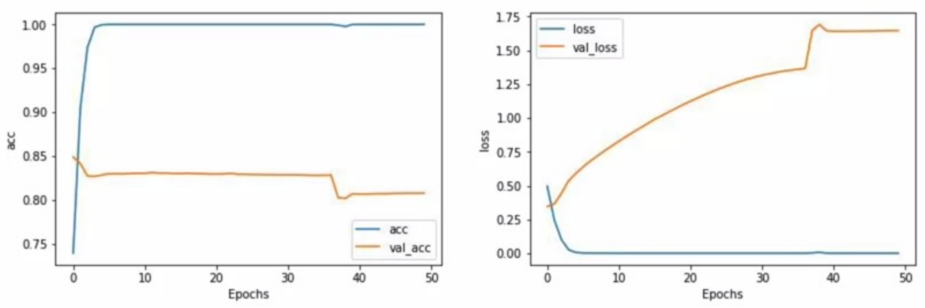


```
imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)


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
```

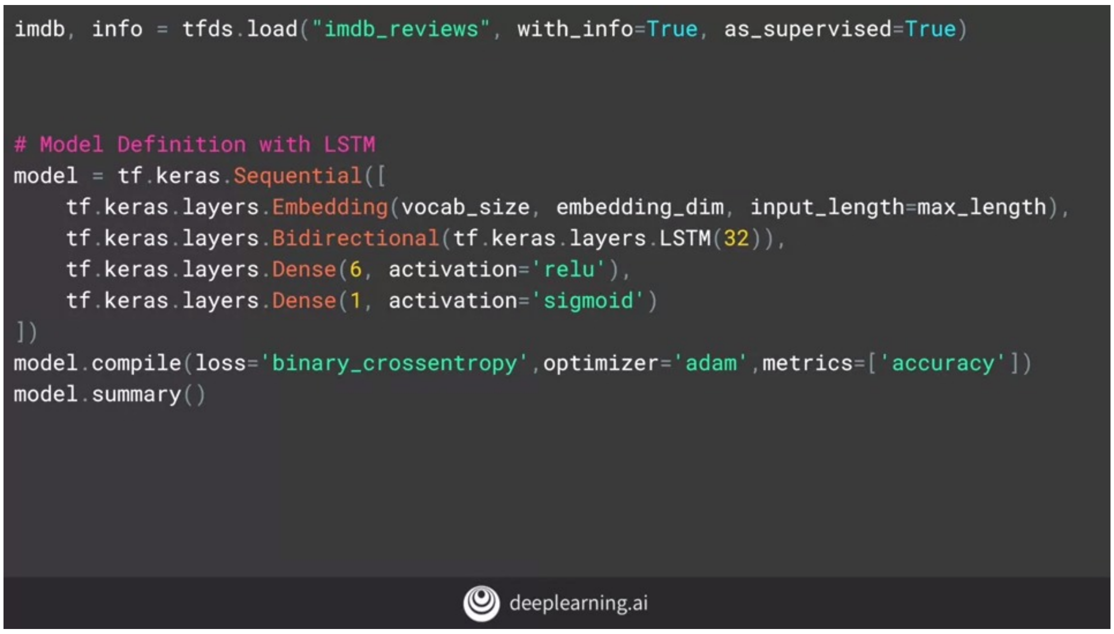
Here, I'll just use an embedding that I flattened before it goes into the dense. My model will look like this, with a 171,533 parameters, and the performance will be like this.



IMDB with Embedding-only : ~ 5s per epoch

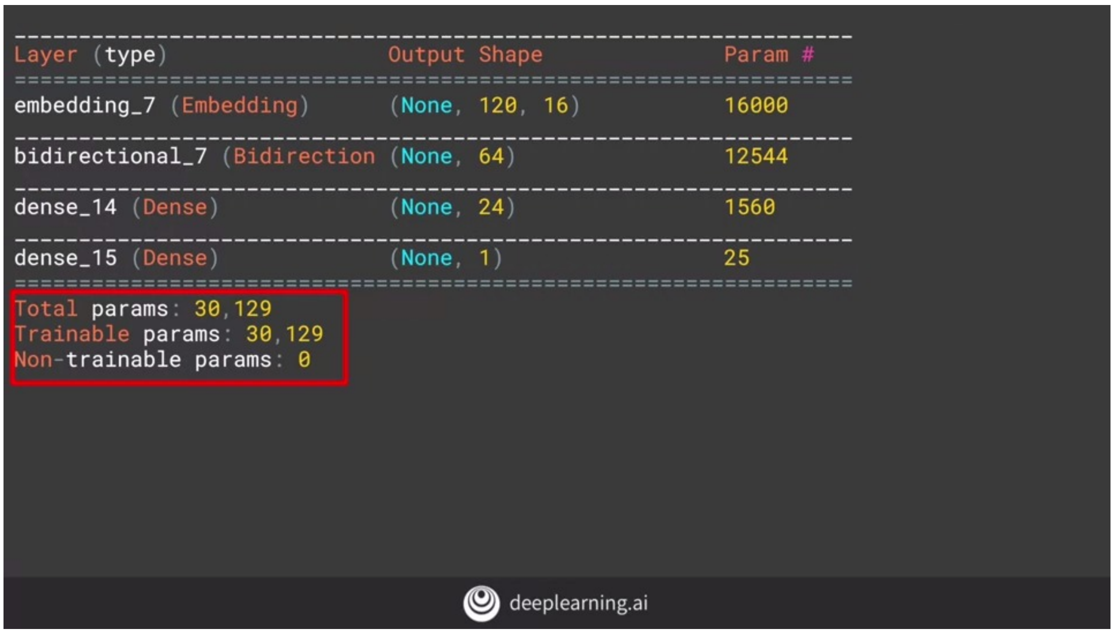
deeplearning.ai

It's nice accuracy, but clear overfitting but it only takes about five seconds per epoch to train.



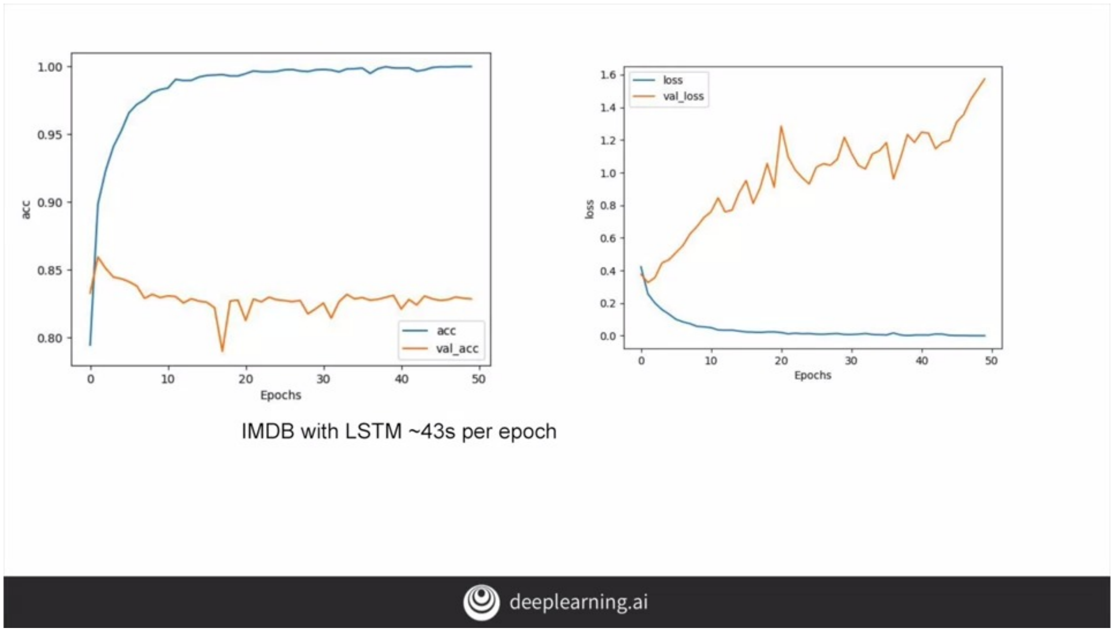
```
imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)


# 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()
```

deeplearning.ai

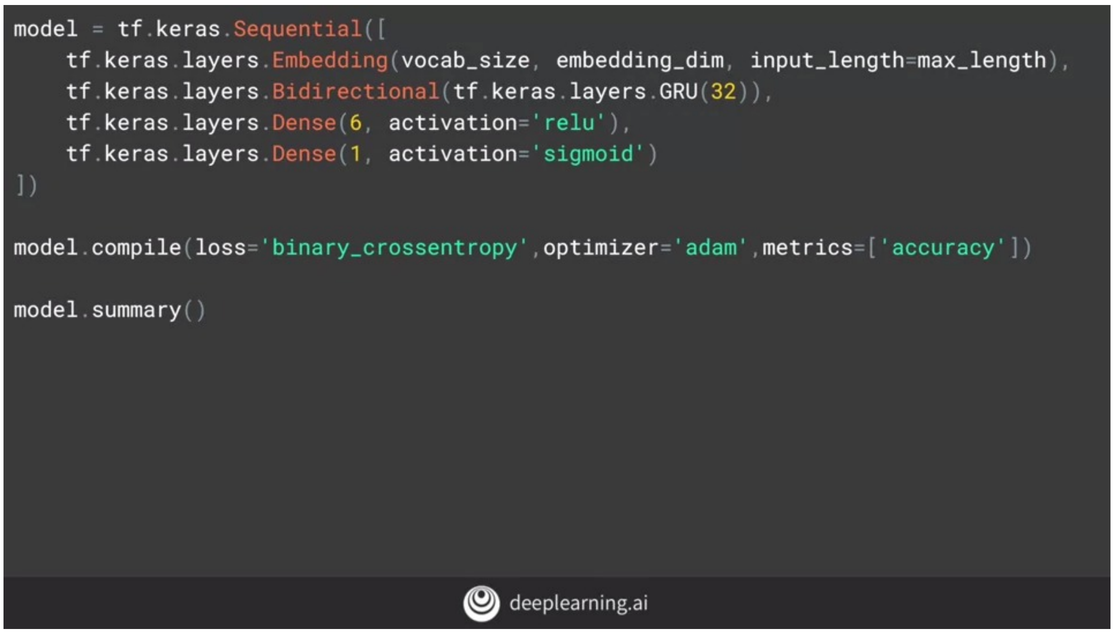


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_7 (Embedding)      (None, 120, 16)           16000
_________________________________________________________________
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
```

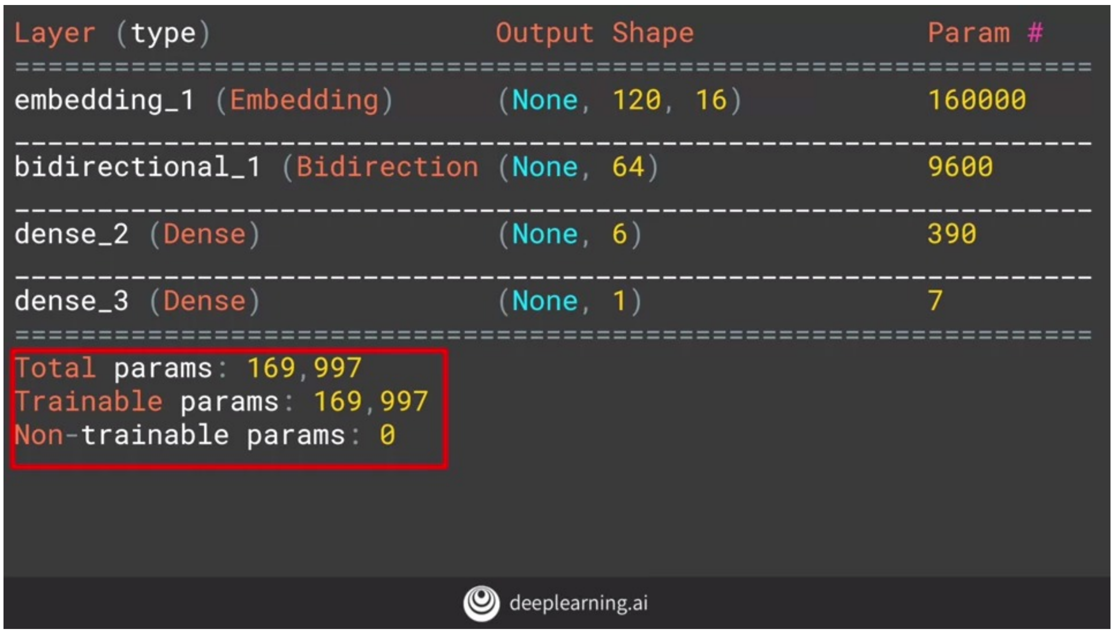
If I change this to use an LSTM, I'll now have only 30,129 parameters, but it will take about 43 seconds per epoch.



The accuracy is better, but there's still some overfitting.

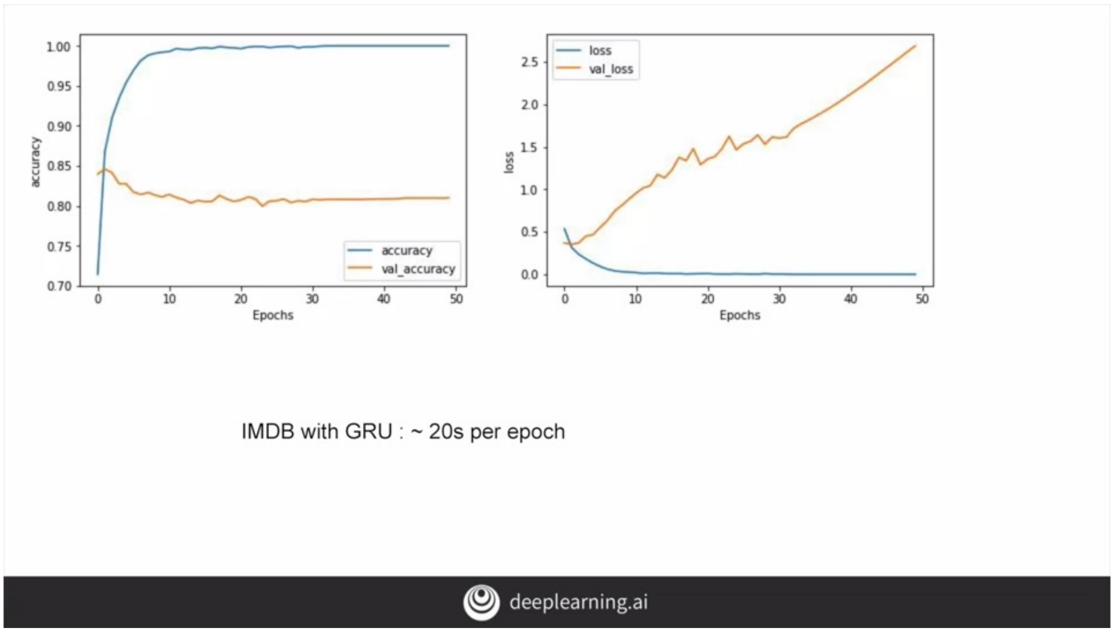


```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(32)),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

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

model.summary()
```



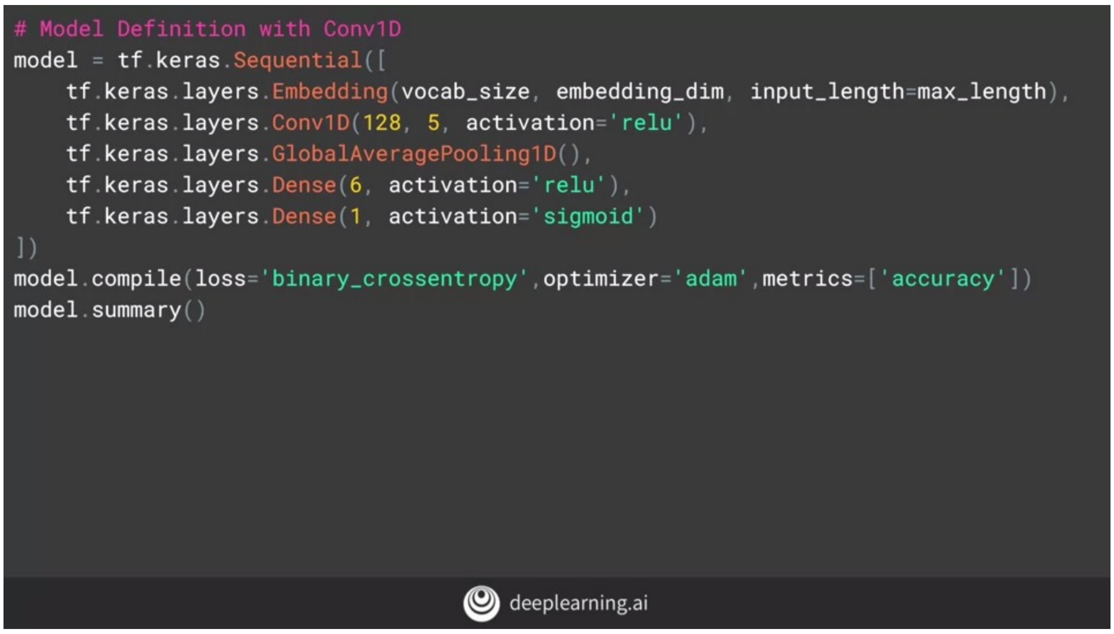
```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 120, 16)           160000
_________________________________________________________________
bidirectional_1 (Bidirection (None, 64)                9600
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 390
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 7
=================================================================
Total params: 169,997
Trainable params: 169,997
Non-trainable params: 0
```

If I try a GRU layer instead, with a GRU being a different type of RNN, and I make it bidirectional, my network will have a 169,997 parameters.

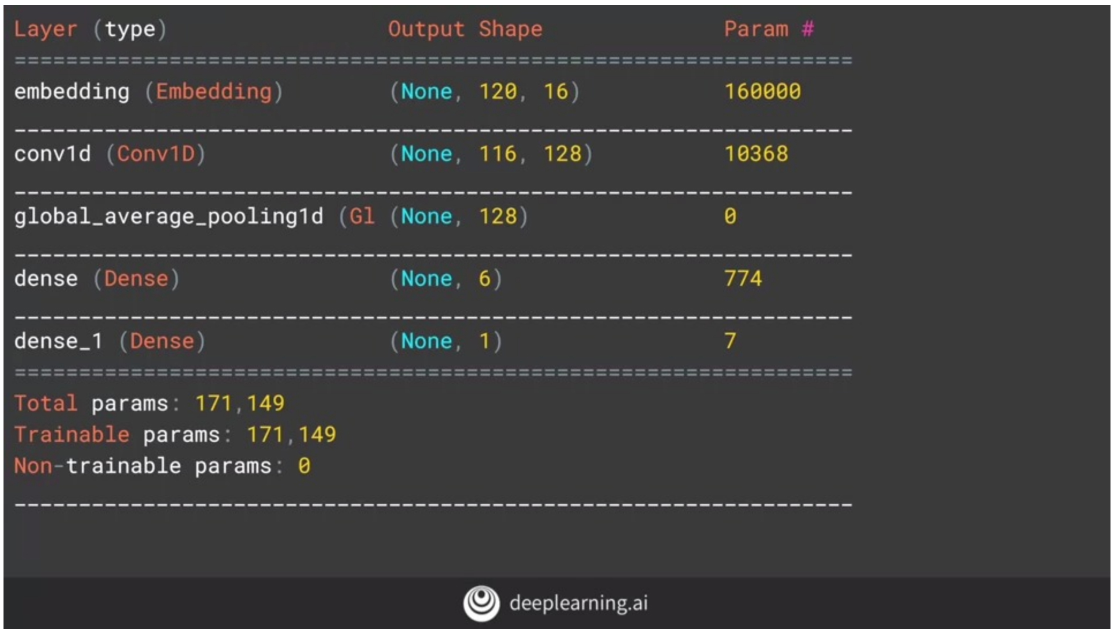


IMDB with GRU : ~ 20s per epoch

My training time will fall to 20 seconds per epoch, and my accuracy is again very good on training, and not too bad on validation but again, showing some overfitting.

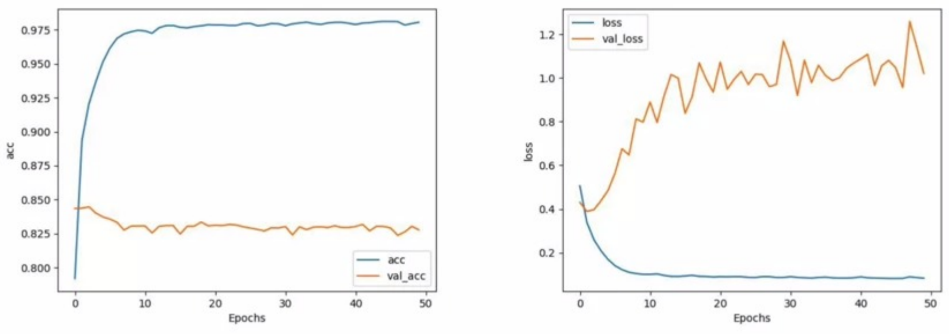


```
# Model Definition with Conv1D
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Conv1D(128, 5, activation='relu'),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(6, activation='relu'),
    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
_________________________________________________________________
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
_________________________________________________________________
```

deeplearning.ai



IMDB with CNN : ~ 6s per epoch

deeplearning.ai

With a convolutional network, I'll have a 171,149 parameters and it only takes about six seconds per epoch to get me close to 100 percent accuracy on training, and about 83 percent on validation, but again with overfitting.

3.9 Tips from Laurence

Before you go onto the next unit, I have created some CO-LABS with each of these options. Try them out for yourself, check on the time, check on the results, and see what techniques you can figure out to avoid some of the overfitting. 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.