COS30018 - Option B - Task 4: Machine Learning 1

Cody Cronin-Sporys

103610020

# Creating/ Explaining the function:

For the first part of this task we need to write a function that creates a prediction model based on various parameters such as number of layers, layer name and layer size. I’ve decided to use the create\_model() function from the p1 code as a base that I can explain and adapt to fit the current project.

After creating a new python file and setting up the includes and imports for both the stock\_prediction file and the new model\_creation file, we can start examining the code.

Firstly, looking at the function itself, it has many parameters for building the model and I’ve written a short description for each of them in the code:

A screen shot of a computer program

Description automatically generated

The main ones to focus on for this weeks task are

* Cell, this is the name of each layer, the current model uses LSTM but this can be changed through this parameter. It is used as the parameter for each model.add() function
* Units, which is the size of each layer and is used as an input for each model.add(cell()) function, with cell being the layer name
* N\_layers, this is the total number of layers for the model, it determines how many times the for loop is run.

Now for the actual code. The following for loop is the main driving force behind the model creation:

A screenshot of a computer code

Description automatically generated

The for loop itself uses the n\_layers parameter to determine the number of times the layer creation process is looped through. Inside the for loop are 3 different conditions based on the current layer number. The first if statement and containing code is used only for the first layer, the second if statement and containing code is used only for the last layer, and the final else stamen and containing code is used for all other layers (hidden layers) between the first and last layer.

Looking at the hidden layer block of code first, we can see that it uses the bidirectional bool parameter in an if statement, and will either use or not use the tensorflow layer Bidirectional as a input for model.add(). Aside from this one difference, both model.add() functions are the same. They use cell for the layer type/ name, then have the units parameter and return\_sequences=True as inputs.

Looking at the last layer, the only difference between it and the hidden layers is that the return\_sequence input for model.add() is set to false instead of true.

Finally, look at the first layer, we can see that it has some additional inputs for the model.add function. Specifically it sets up the batch input shape using the sequence\_length and n\_features parameters.

After a layer has been added from one of the methods described above, a dropout layer is added to the model using the dropout parameter and an input. This is repeated for every created layer.

Once all layers have been created and the for loop is complete, a final Dense layer is added to the model and then it is compiled using the defined loss and optimizer parameters. The model is then complete and is return from the function.

A black background with white text

Description automatically generated

Going back to the stock\_prediction file, we can see how this function has been implemented into the current codebase. The first step is to setup new variables that will act as the inputs for the model creation function:

A screen shot of a computer program

Description automatically generated

We can then call the function in the same place as the old model building code was, using the parameters defined above.

A screen shot of a computer

Description automatically generated

The code can then continue as normal, since all the implementation issues were fixed and documented at the start of the previous report.

# Testing Different Variables:

The first test I’m going to do is with all the settings as standard. So layer name is LSTM, layer size is 256 and layer number is 2. We also have the batch size as 32 and epochs as 25 for the model fitting. This model is going to act as the control for the experiments, so we have something to compare back to.

Looking at the whole dataset we can see the following:

A graph showing a graph of amzn price

Description automatically generated

This isn’t very useful since it’s too large to see the difference between predicted and actual so ill be zooming in on just a small section of the data, the section between 300 and 340:

A graph showing a price of amzn

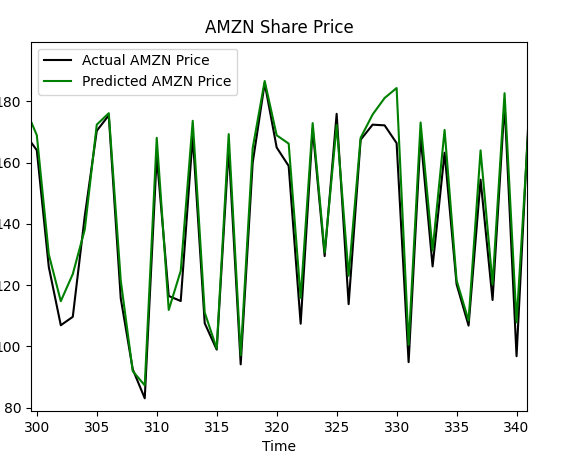
Description automatically generated

We can see in this from this dataset that the prediction is fairly accurate in most places with a few outlier sections where the prediction is way off (such as the dip near 300 or the spike near 330). Looking at the console log we can also see that each step (epoch) took around 10-11 seconds.

A screen shot of a computer

Description automatically generated

For the second model I’ll be changing the layer name (CELL) from LSTM to GRU.



A screen shot of a computer program

Description automatically generated

First of all, we can see from the console logs that the average time taken for each step has dropped down to around 8-9 seconds, which is a fairly significant improvement, saving almost a minute over the full 25 steps. As for the prediction itself, it’s very similar with small improvements in some areas but worse in others. For example, the spike just below 320 is more accurate with GRU but the dip between 320 and 325 is less accurate, and both very inaccurate for the small bump right on 320. Overall I would say in this case, GRU is the more accurate layer name on average.

For the next test I’ll switch the layer name back to LSTM and change the number of layers from 2 to 6.

A graph showing an average price

Description automatically generated

A black screen with white text

Description automatically generated

Despite taking far longer than previous models, with an average of ~50s for each step instead of ~10, the prediction are noticeably worse. This is likely due to overfitting of data, which is when random noise and fluctuations are picked up by the model due to the increased complexity. This would likely be improved by using GRU as it deals with overfitting better.

The next test is back to 2 layers, but the layer size will be changing from 256 to 64.

A graph showing a graph of an average price

Description automatically generated with medium confidence

A black screen with white text

Description automatically generated

This model is less accurate than the original mode, with a trend for overpredicting prices rather than underpredicting, which can be see from all the peaks overshooting the actual prices. The lower accuracy is likely due to a reduction in complexity and thus has come with the benefit of significantly reduced step times, with an average of ~2s.

The next test will be using different epoch size, Ill be reducing it down from 25 to 5.

A graph showing the price of amzn

Description automatically generated

A screen shot of a computer

Description automatically generated

This model has had the worst accuracy so far, with a trend to underpredict prices. The time per step is still the same but with 1/5 the number of steps the equivalent time per step goes from ~11s to ~2s

The final test will be with a batch\_size of 128 instead of 32.

A graph showing the price of amzn

Description automatically generated

A black screen with white text

Description automatically generated

This model is almost indistinguishable from the first model. It seems that changing just the batch size doesn’t have a noticeable effect on the predictions in this case. It does save a decent amount of time on each step though.