COS30018- Option B- Task 4: Machine Learning 1

Cody Cronin-Sporys

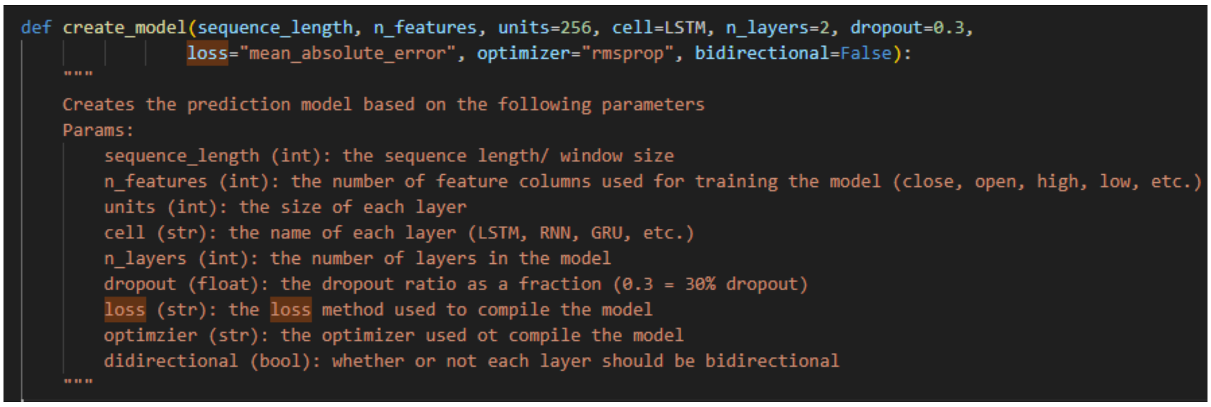
103610020

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:

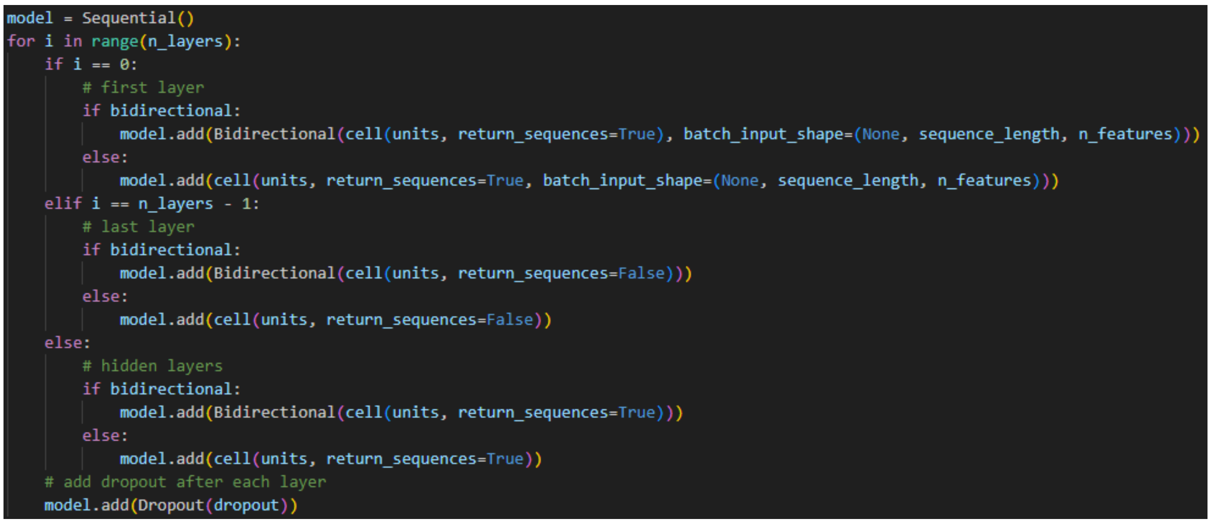


```
def create_model(sequence_length, n_features, units=256, cell=LSTM, n_layers=2, dropout=0.3,
                loss="mean_absolute_error", optimizer="rmsprop", bidirectional=False):
    """
    Creates the prediction model based on the following parameters
    Params:
        sequence_length (int): the sequence length/ window size
        n_features (int): the number of feature columns used for training the model (close, open, high, low, etc.)
        units (int): the size of each layer
        cell (str): the name of each layer (LSTM, RNN, GRU, etc.)
        n_layers (int): the number of layers in the model
        dropout (float): the dropout ratio as a fraction (0.3 = 30% dropout)
        loss (str): the loss method used to compile the model
        optimzier (str): the optimizer used ot compile the model
        didirectional (bool): whether or not each layer should be bidirectional
    """
```

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:



```python
model = Sequential()
for i in range(n_layers):
    if i == 0:
        # first layer
        if bidirectional:
            model.add(Bidirectional(cell(units, return_sequences=True), batch_input_shape=(None, sequence_length, n_features)))
        else:
            model.add(cell(units, return_sequences=True, batch_input_shape=(None, sequence_length, n_features)))
    elif i == n_layers - 1:
        # last layer
        if bidirectional:
            model.add(Bidirectional(cell(units, return_sequences=False)))
        else:
            model.add(cell(units, return_sequences=False))
    else:
        # hidden layers
        if bidirectional:
            model.add(Bidirectional(cell(units, return_sequences=True)))
        else:
            model.add(cell(units, return_sequences=True))
    # add dropout after each layer
    model.add(Dropout(dropout))
```

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.

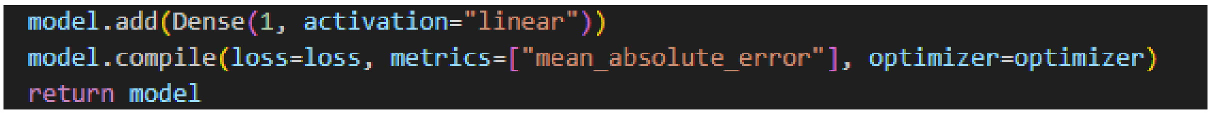
Looking at the hidden layer block of code first, we can see that it uses the bidirectional bool parameter in an if statement, and will either use or not use the tensorflow layer Bidirectional as a input for model.add(). Aside from this one difference, both model.add() functions are the same. They use 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.

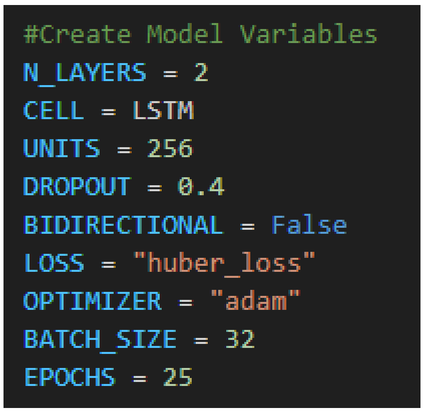
After a layer has been added from one of the methods described above, a dropout layer is added to the model using the dropout parameter and an input. This is repeated for every created layer.

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.



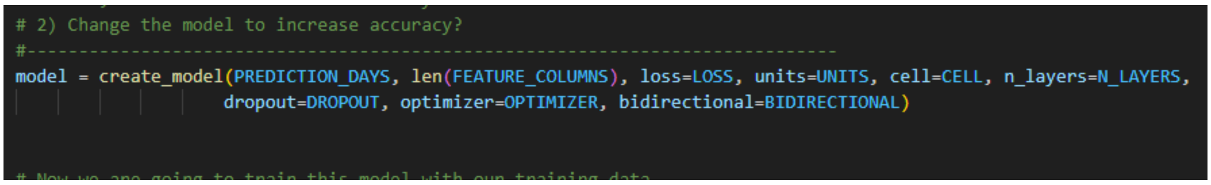
```python
model.add(Dense(1, activation="linear"))
model.compile(loss=loss, metrics=["mean_absolute_error"], optimizer=optimizer)
return model
```

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:



```
#Create Model Variables
N_LAYERS = 2
CELL = LSTM
UNITS = 256
DROPOUT = 0.4
BIDIRECTIONAL = False
LOSS = "huber_loss"
OPTIMIZER = "adam"
BATCH_SIZE = 32
EPOCHS = 25
```

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



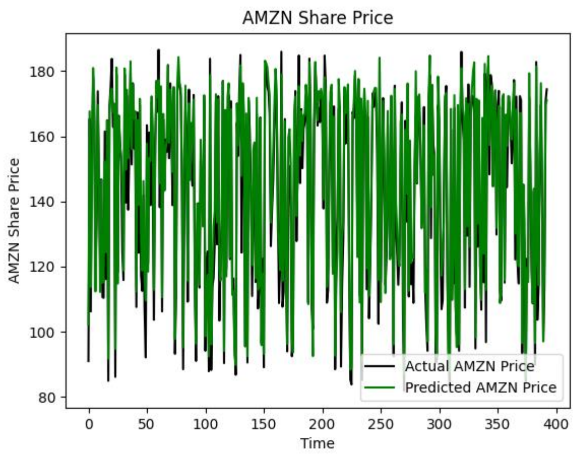
```
# 2) Change the model to increase accuracy?
#------------------------------------------------------------------------------
model = create_model(PREDICTION_DAYS, len(FEATURE_COLUMNS), loss=LOSS, units=UNITS, cell=CELL, n_layers=N_LAYERS,
                     dropout=DROPOUT, optimizer=OPTIMIZER, bidirectional=BIDIRECTIONAL)
```

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

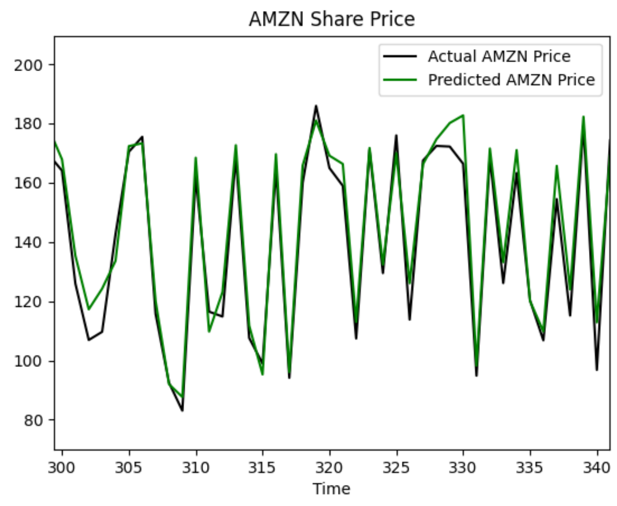
Testing Different Variables:

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.

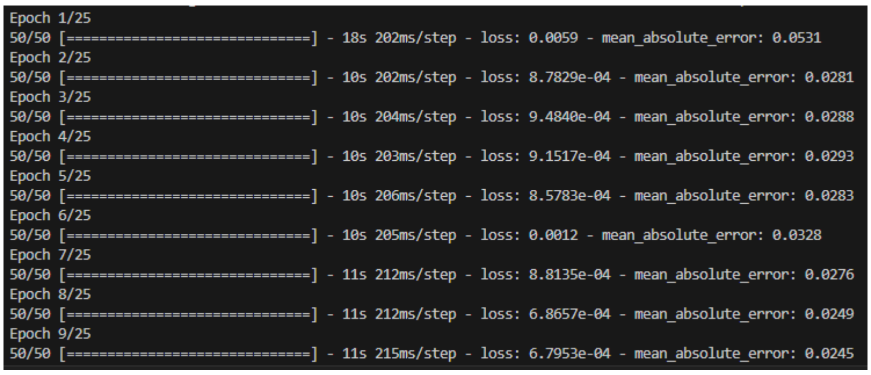
Looking at the whole dataset we can see the following:



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:

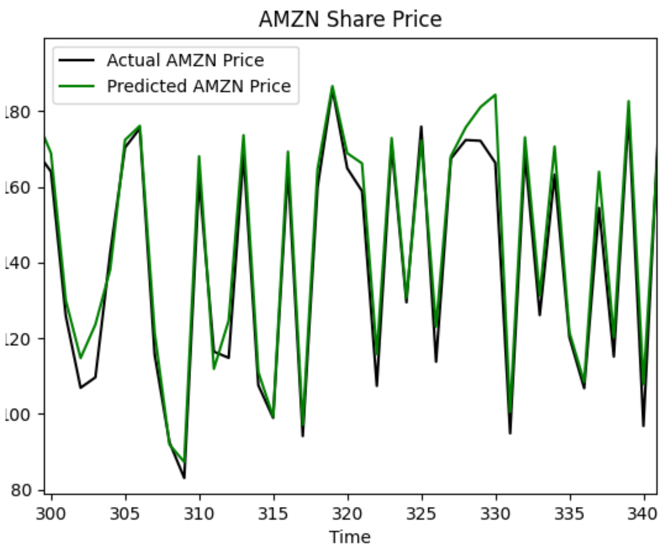


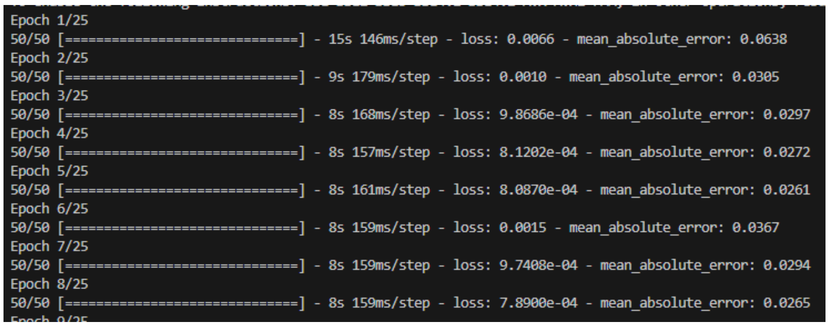
We can see in this from this dataset that the prediction is fairly accurate in most places with a few outlier sections where the prediction is way off (such as the dip near 300 or the spike near 330). Looking at the console log we can also see that each step (epoch) took around 10-11 seconds.



```
Epoch 1/25
50/50 [==============================] - 18s 202ms/step - loss: 0.0059 - mean_absolute_error: 0.0531
Epoch 2/25
50/50 [==============================] - 10s 202ms/step - loss: 8.7829e-04 - mean_absolute_error: 0.0281
Epoch 3/25
50/50 [==============================] - 10s 204ms/step - loss: 9.4840e-04 - mean_absolute_error: 0.0288
Epoch 4/25
50/50 [==============================] - 10s 203ms/step - loss: 9.1517e-04 - mean_absolute_error: 0.0293
Epoch 5/25
50/50 [==============================] - 10s 206ms/step - loss: 8.5783e-04 - mean_absolute_error: 0.0283
Epoch 6/25
50/50 [==============================] - 10s 205ms/step - loss: 0.0012 - mean_absolute_error: 0.0328
Epoch 7/25
50/50 [==============================] - 11s 212ms/step - loss: 8.8135e-04 - mean_absolute_error: 0.0276
Epoch 8/25
50/50 [==============================] - 11s 212ms/step - loss: 6.8657e-04 - mean_absolute_error: 0.0249
Epoch 9/25
50/50 [==============================] - 11s 215ms/step - loss: 6.7953e-04 - mean_absolute_error: 0.0245
```

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

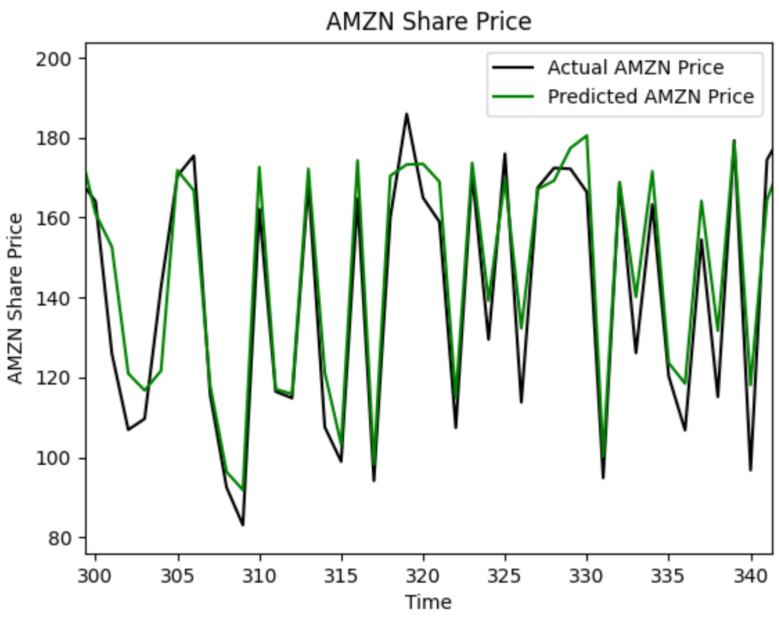


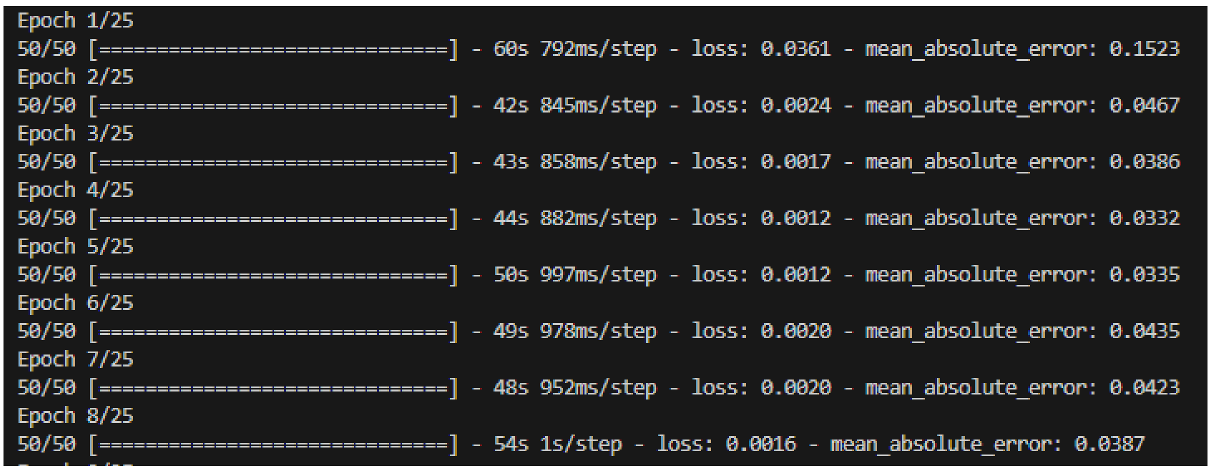


```
Epoch 1/25
50/50 [==============================] - 15s 146ms/step - loss: 0.0066 - mean_absolute_error: 0.0638
Epoch 2/25
50/50 [==============================] - 9s 179ms/step - loss: 0.0010 - mean_absolute_error: 0.0305
Epoch 3/25
50/50 [==============================] - 8s 168ms/step - loss: 9.8686e-04 - mean_absolute_error: 0.0297
Epoch 4/25
50/50 [==============================] - 8s 157ms/step - loss: 8.1202e-04 - mean_absolute_error: 0.0272
Epoch 5/25
50/50 [==============================] - 8s 161ms/step - loss: 8.0870e-04 - mean_absolute_error: 0.0261
Epoch 6/25
50/50 [==============================] - 8s 159ms/step - loss: 0.0015 - mean_absolute_error: 0.0367
Epoch 7/25
50/50 [==============================] - 8s 159ms/step - loss: 9.7408e-04 - mean_absolute_error: 0.0294
Epoch 8/25
50/50 [==============================] - 8s 159ms/step - loss: 7.8900e-04 - mean_absolute_error: 0.0265
```

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.

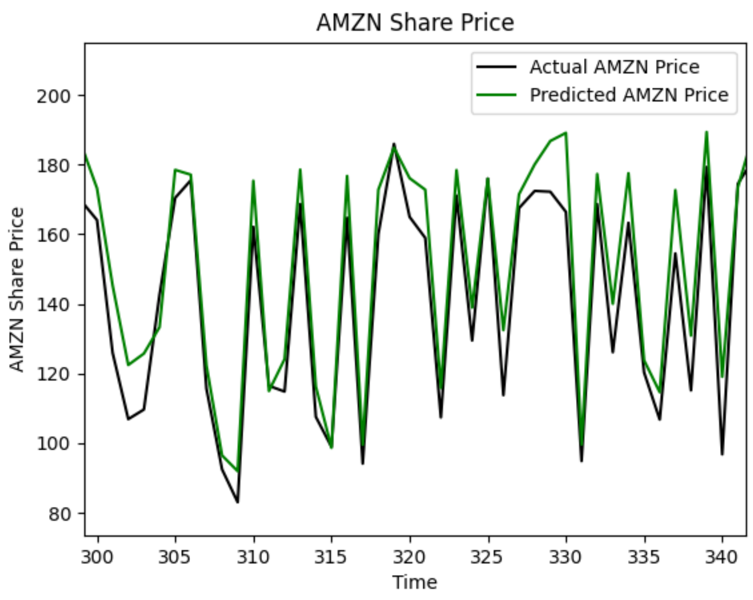


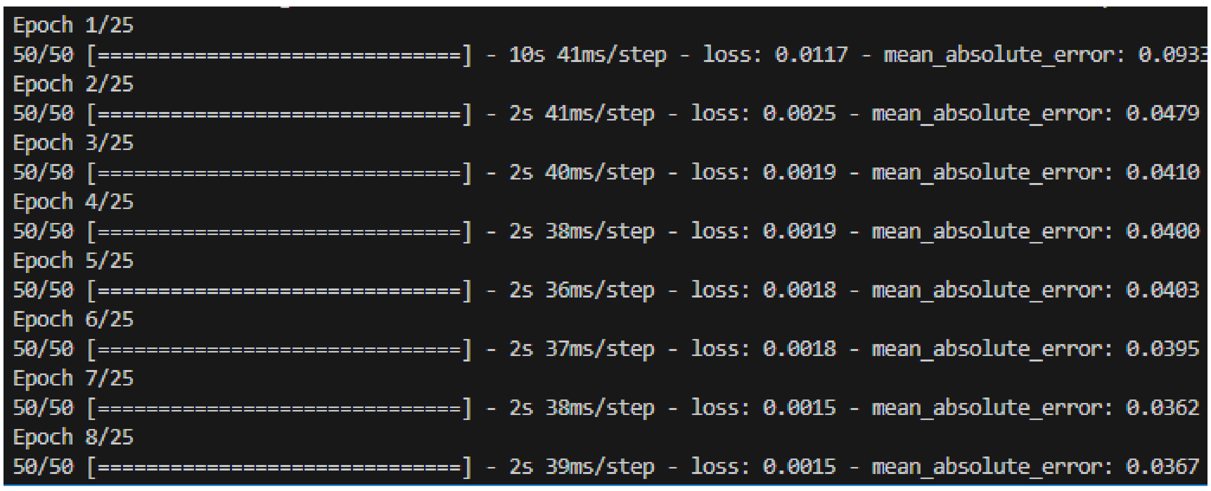


```
Epoch 1/25
50/50 [==============================] - 60s 792ms/step - loss: 0.0361 - mean_absolute_error: 0.1523
Epoch 2/25
50/50 [==============================] - 42s 845ms/step - loss: 0.0024 - mean_absolute_error: 0.0467
Epoch 3/25
50/50 [==============================] - 43s 858ms/step - loss: 0.0017 - mean_absolute_error: 0.0386
Epoch 4/25
50/50 [==============================] - 44s 882ms/step - loss: 0.0012 - mean_absolute_error: 0.0332
Epoch 5/25
50/50 [==============================] - 50s 997ms/step - loss: 0.0012 - mean_absolute_error: 0.0335
Epoch 6/25
50/50 [==============================] - 49s 978ms/step - loss: 0.0020 - mean_absolute_error: 0.0435
Epoch 7/25
50/50 [==============================] - 48s 952ms/step - loss: 0.0020 - mean_absolute_error: 0.0423
Epoch 8/25
50/50 [==============================] - 54s 1s/step - loss: 0.0016 - mean_absolute_error: 0.0387
```

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.

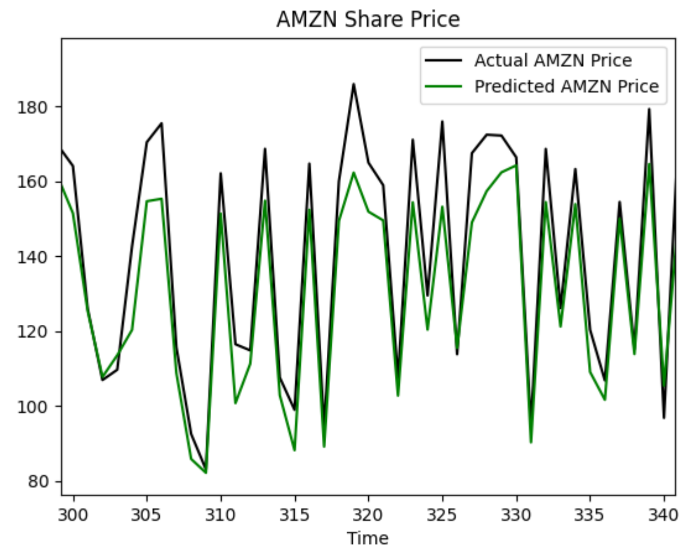


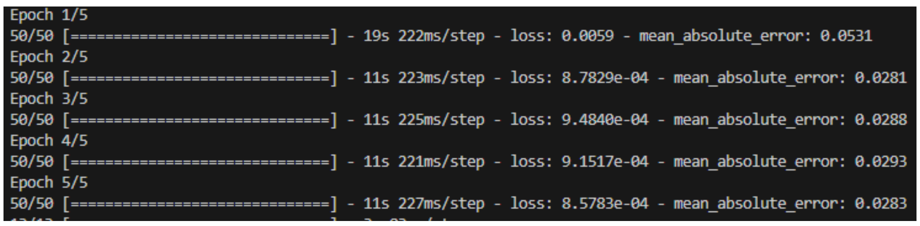


```
Epoch 1/25
50/50 [==============================] - 10s 41ms/step - loss: 0.0117 - mean_absolute_error: 0.093
Epoch 2/25
50/50 [==============================] - 2s 41ms/step - loss: 0.0025 - mean_absolute_error: 0.0479
Epoch 3/25
50/50 [==============================] - 2s 40ms/step - loss: 0.0019 - mean_absolute_error: 0.0410
Epoch 4/25
50/50 [==============================] - 2s 38ms/step - loss: 0.0019 - mean_absolute_error: 0.0400
Epoch 5/25
50/50 [==============================] - 2s 36ms/step - loss: 0.0018 - mean_absolute_error: 0.0403
Epoch 6/25
50/50 [==============================] - 2s 37ms/step - loss: 0.0018 - mean_absolute_error: 0.0395
Epoch 7/25
50/50 [==============================] - 2s 38ms/step - loss: 0.0015 - mean_absolute_error: 0.0362
Epoch 8/25
50/50 [==============================] - 2s 39ms/step - loss: 0.0015 - mean_absolute_error: 0.0367
```

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.

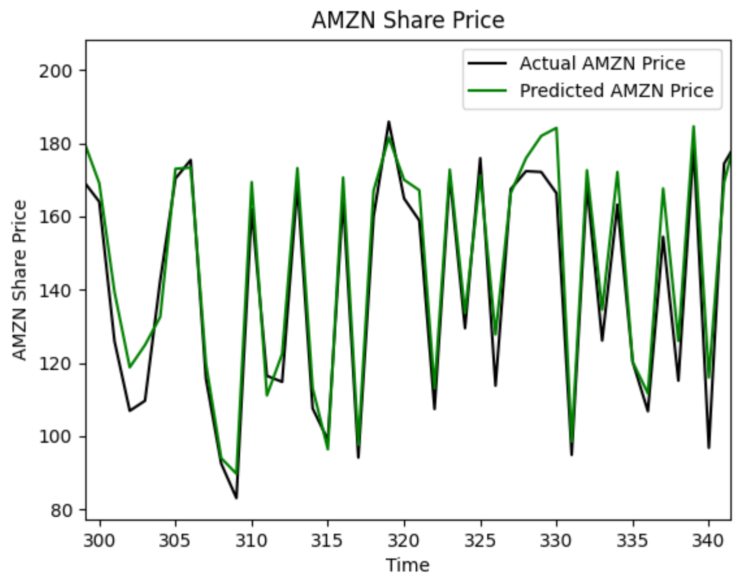
The next test will be using different epoch size, Ill be reducing it down from 25 to 5.

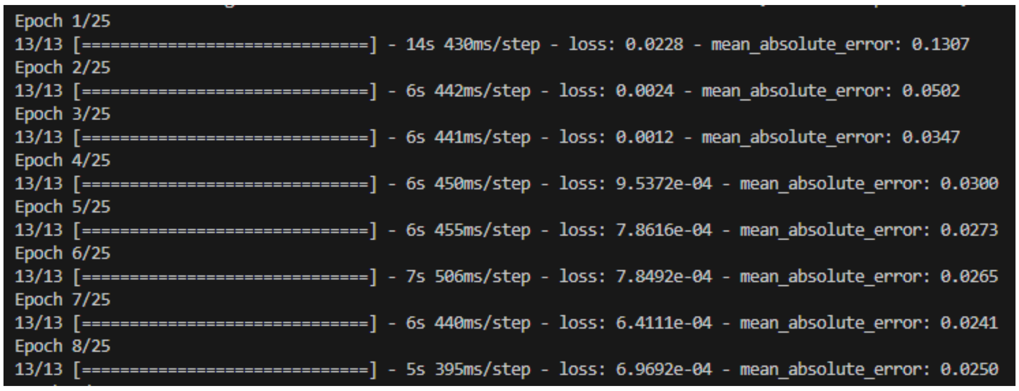



```
Epoch 1/5
50/50 [==============================] - 19s 222ms/step - loss: 0.0059 - mean_absolute_error: 0.0531
Epoch 2/5
50/50 [==============================] - 11s 223ms/step - loss: 8.7829e-04 - mean_absolute_error: 0.0281
Epoch 3/5
50/50 [==============================] - 11s 225ms/step - loss: 9.4840e-04 - mean_absolute_error: 0.0288
Epoch 4/5
50/50 [==============================] - 11s 221ms/step - loss: 9.1517e-04 - mean_absolute_error: 0.0293
Epoch 5/5
50/50 [==============================] - 11s 227ms/step - loss: 8.5783e-04 - mean_absolute_error: 0.0283
```

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.





```
Epoch 1/25
13/13 [==============================] - 14s 430ms/step - loss: 0.0228 - mean_absolute_error: 0.1307
Epoch 2/25
13/13 [==============================] - 6s 442ms/step - loss: 0.0024 - mean_absolute_error: 0.0502
Epoch 3/25
13/13 [==============================] - 6s 441ms/step - loss: 0.0012 - mean_absolute_error: 0.0347
Epoch 4/25
13/13 [==============================] - 6s 450ms/step - loss: 9.5372e-04 - mean_absolute_error: 0.0300
Epoch 5/25
13/13 [==============================] - 6s 455ms/step - loss: 7.8616e-04 - mean_absolute_error: 0.0273
Epoch 6/25
13/13 [==============================] - 7s 506ms/step - loss: 7.8492e-04 - mean_absolute_error: 0.0265
Epoch 7/25
13/13 [==============================] - 6s 440ms/step - loss: 6.4111e-04 - mean_absolute_error: 0.0241
Epoch 8/25
13/13 [==============================] - 5s 395ms/step - loss: 6.9692e-04 - mean_absolute_error: 0.0250
```

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.