Task 4 Report

Naca Hitchman - 103072170

# P1 Create Model Function

As the task specifications suggest doing, I have based my create model function primarily off the create model function found in the P1 base. Thus, I will explain how that original function works first.

Firstly, the original method is taking the input parameters sequence length and n features, which represent the shape of the test data. It also takes in the following:   


* units: The number of units in each layer of the model.
* cell: The type of recurrent layer to use (e.g., LSTM, GRU, or SimpleRNN).
* n\_layers: The number of layers in the model.
* dropout: The dropout rate to apply after each layer.
* loss: The loss function to use when compiling the model.
* optimizer: The optimizer to use when compiling the model.
* bidirectional: Whether to use a bidirectional model.

The function then creates an instance of the Sequential model class, which is a linear stack of layers that can be used to build a neural network. Next, the function enters a for loop that iterates over the number of layers specified by the n\_layers parameter. For each layer, the function performs the following steps: 

1. The function checks if the current layer is the first layer by comparing the loop index i to 0. If it is the first layer, the function adds a recurrent layer of the type specified by the cell parameter (e.g., LSTM, GRU, or SimpleRNN) with the number of units specified by the units parameter. The input shape of this layer is set to (None, sequence\_length, n\_features), where None is a placeholder for the batch size that will be determined when fitting the model. 
2. If the current layer is not the first layer, the function checks if it is the last layer by comparing the loop index i to n\_layers - 1. If it is the last layer, the function adds another recurrent layer of the type specified by the cell parameter with the number of units specified by the units parameter. However, this time, the return\_sequences argument is set to False so that only the last output of this layer is returned. 
3. If the current layer is neither the first nor the last layer, it must be a hidden layer. In this case, the function adds another recurrent layer of the type specified by the cell parameter with the number of units specified by the units parameter and sets its input shape to (None, sequence\_length, n\_features).
4. Before adding each recurrent layer, if bidirectional is set to True then it wraps the layer in a Bidirectional wrapper to create a bidirectional version of that layer.
5. After adding each recurrent or bidirectional layer, depending on whether bidirectional was set to True or False respectively, the function adds a Dropout layer with rate specified by dropout parameter after each recurrent or bidirectional layer.

After exiting from for loop and adding all layers to model, the function adds a final Dense output layer with 1 unit and linear activation. The function then compiles model using compile method with loss function and optimizer specified by loss and optimizer parameters respectively. Finally, the model is returned.

# Function Adjustments

To use within my existing project code, I’ve taken the P1 function, and modified it to fit the task specifications and include more functions. For the most part, the P1 function already achieves what is required by the spec, but there are a few useful adjustments made.



1. I’ve changed the units parameter from a single integer value to a list of integers. This allows for specifying different numbers of units for each layer in the model. In the original function, all layers had the same number of units, as specified by the units parameter. This also mean instead of using the value of the units parameter directly, which is a single integer value, it now uses indexing to access the appropriate element from the units list for each layer.
2. I’ve added a new cells parameter, which is a list of strings that specify the type of each layer in the model. This allows for using different types of recurrent layers (e.g., LSTM, GRU, or SimpleRNN) in the same model. In the original function, all layers were of the same type, as specified by the cell parameter. This also means instead of using the value of the cell parameter directly it now uses indexing to access the appropriate element from the cells list for each layer.
3. I’ve changed how the function gets a reference to the corresponding layer network object based on the cell name. Instead of directly using the value of the cell parameter, which is an object, it now uses the globals() function to get a reference to the corresponding layer network object based on the string value passed in for each layer. 
4. I’ve added a check in the for loop to make sure that the name of the cell (layer type) for each layer corresponds to a valid layer network type. If the cell name is not found in the global symbol table, the function raises a ValueError with an appropriate error message.

These changes and additions allow for more flexibility when creating models using this function. We can now specify different numbers of units and different types of recurrent layers for each layer in your model by passing in appropriate values for these parameters when calling this function.

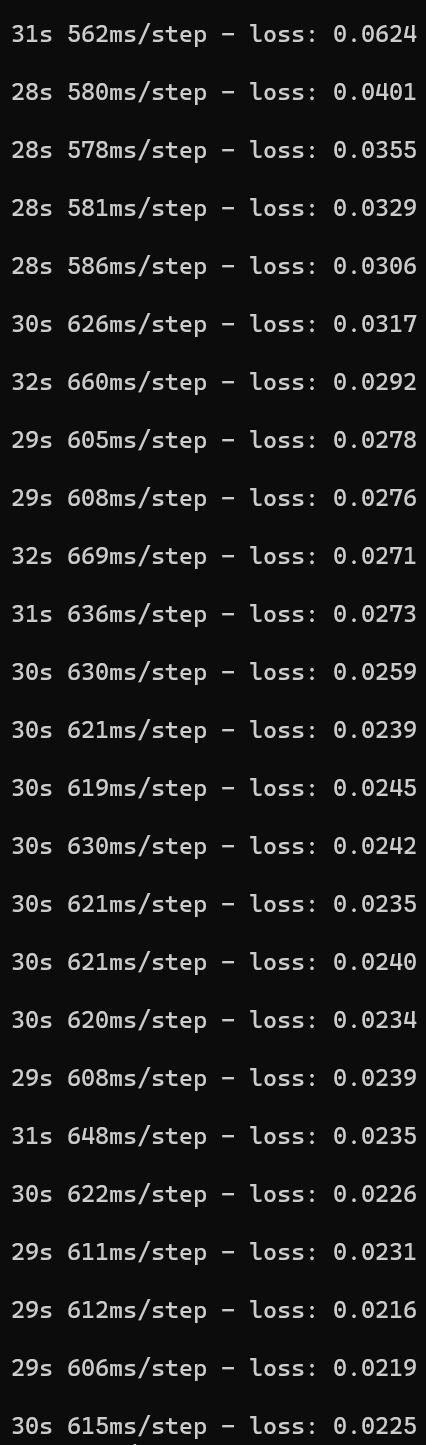
# Function Calling & Results

Due to now using a function to create the model, the original v0.1 code for creating the model can be replaced with this new method. It is as simple as setting the values for the parameters to be passed to the method, with sequence\_length and n\_features being derived from the training data shape, and the layer network types and unit sizes being specified in lists.

The create model function is then ran, before being fit with the training data with specified epochs and batch size.



This initial test uses two layers, an LTSM with 256 unit size, and a GRU with 128 unit size. It also keeps the standard 25 epochs and 32 batch size of the original v0.1 base. This predicts quite an accurate result to the test data, with on average 30 seconds per epoch, and loss between 0.0624 and 0.0225

The next test set uses the following parameters. This notably is using three layers of different types, with decreasing unit sizes. It is also testing a different optimizer method.



These results interestingly appear not as accurate, being very slightly offset and exaggerated. Further, the epoch time average is longer, but with a much lower average loss.

A graph showing the price of a stock market

Description automatically generatedA screenshot of a computer screen

Description automatically generated

The set uses two GRU layers, starting with a much larger 512 layer size. The dropout is also slightly higher, with yet another different optimizer. The epochs and batch size are also increased in this iteration.



As is extremely noticeable from these results, this particular combination of model and fit settings does not work at all for our purposes. Both the actual and predicted prices become completely incorrectly displayed, and the loss for each epoch are insanely high numbers. This could be the result of many of the factors, or only one or two, such as the optimiser or loss methods.

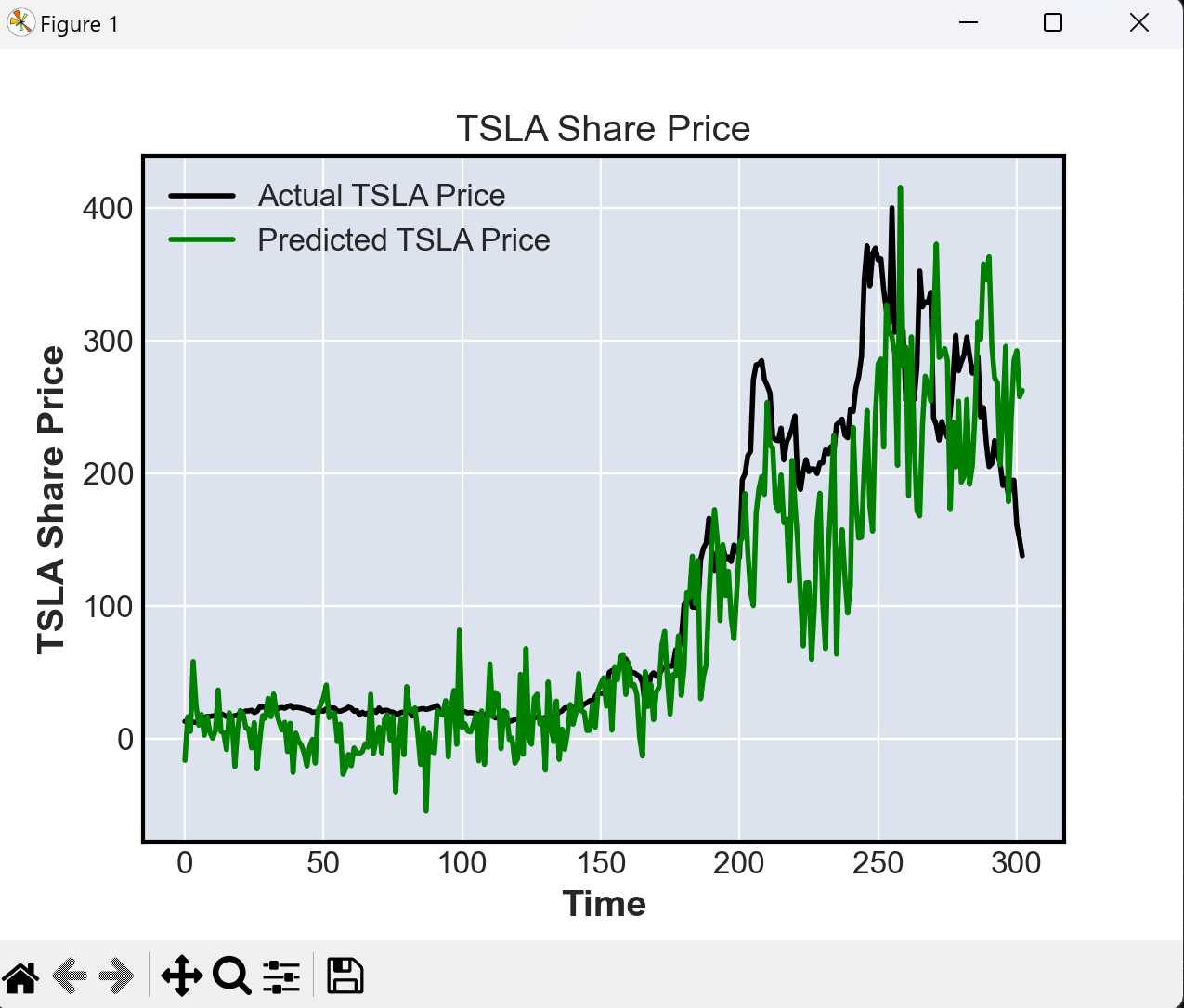
A graph with numbers and a green line

Description automatically generatedA black screen with white text

Description automatically generated

This last set is testing three SimpleRNN layers, with yet another combination of loss and optimizer methods. It also tests much smaller epoch and batch\_size values. The unit sizes for each layer are also reduced.

As expected with a much small epoch and batch size, while the overall arc shape is close, the predicted values are not close to the actual values. The values are super erratic with wild prices differences from day to day. This is likely due to a combination of the lower epoch and batch size, as well as the layer configurations and potentially the loss and optimiser methods used. We can see this also means the average epoch time, and loss, is very low.

A screen shot of a computer

Description automatically generated

Overall, these results suggest that different model creation and fitting configurations can have a wildly large impact on the performance and accuracy of the final model and prediction. It would be very important when using machine learning to understand the differences between each option in a configuration and how they could impact the final result to be able to select values that give the desired result in the desired amount of time.