# 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:

```
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):
```

- 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:

```
model = Sequential()
for i in range(n_layers):
```

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.
   ```
   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)))
   ```
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.

```
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))
```

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).

```
else:
        # hidden layers
        if bidirectional:
                model.add(Bidirectional(cell(units, return_sequences=True)))
        else:
                model.add(cell(units, return_sequences=True))
```

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.

```
model.add(Dropout(dropout))
```

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.

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

## 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.

```
def create_model(sequence_length, n_features, units=[256], cells=['LSTM'],
n_layers=2, dropout=0.3, loss="mean_absolute_error", optimizer="rmsprop",
bidirectional=False):
```

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.

```
# Get the number of units for this layer
unit = units[i]
```

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.

```
# Get the name of the cell (layer type) for this layer
cell_name = cells[i]
```

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.

```
# Get a reference to the corresponding layer network object
cell = globals()[cell_name]
```

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.

```
# Check if the cell name corresponds to a valid layer network type
if cell_name not in globals():
      raise ValueError(f"Invalid layer network type: {cell_name}")
```

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.

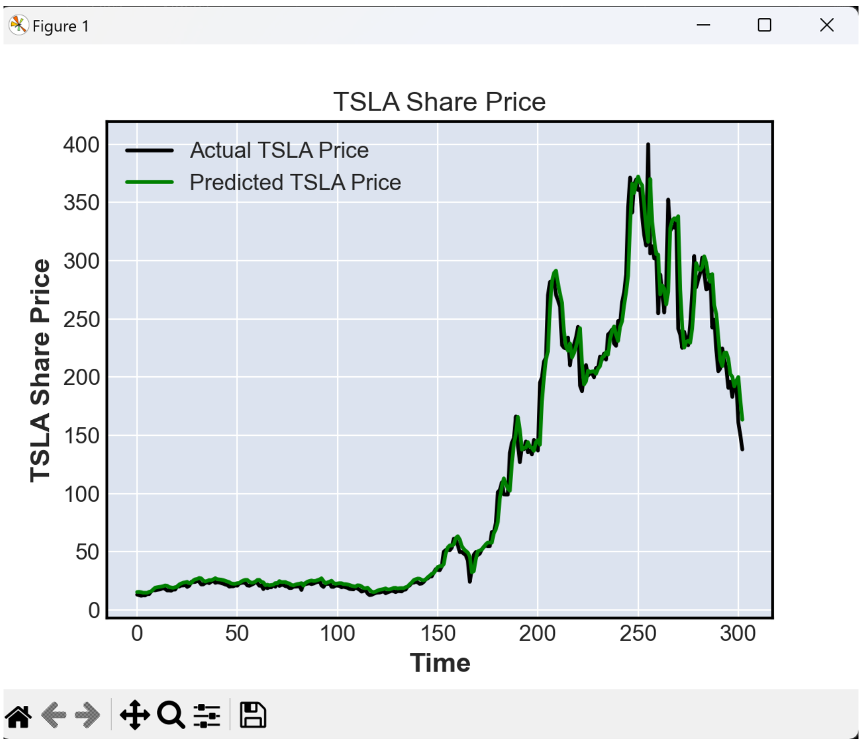
## Function Calling & Results

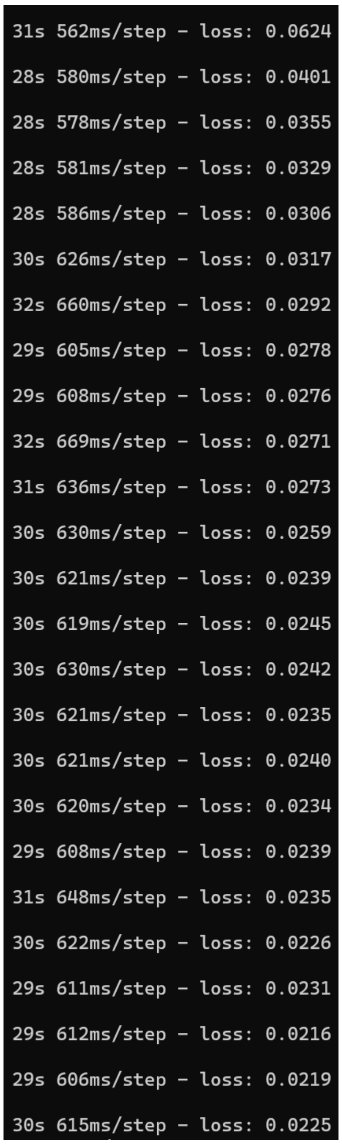
Due to now using a function to create the model, the original v0.1 code for creating the model can be replaced with this new method. It is as simple as setting the values for the parameters to be passed to the method, with sequence_length and n_features being derived from the training data shape, and 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.

```
#Task 4
sequence_length = data['X_train'].shape[1]
n_features = data['X_train'].shape[2]
units = [256, 128]
cells = ['LSTM', 'GRU']
n_layers = 2
dropout = 0.3
loss = "mean_absolute_error"
optimizer = "rmsprop"
bidirectional = True

# Create the model using the create_model function
model = create_model(sequence_length, n_features, units=units, cells=cells,
n_layers=n_layers, dropout=dropout, loss=loss, optimizer=optimizer,
bidirectional=bidirectional)

# Set the number of epochs and batch size
epochs = 25
batch_size = 32

# Train the model on the training data
model.fit(data['X_train'], data['y_train'], epochs=epochs,
batch_size=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



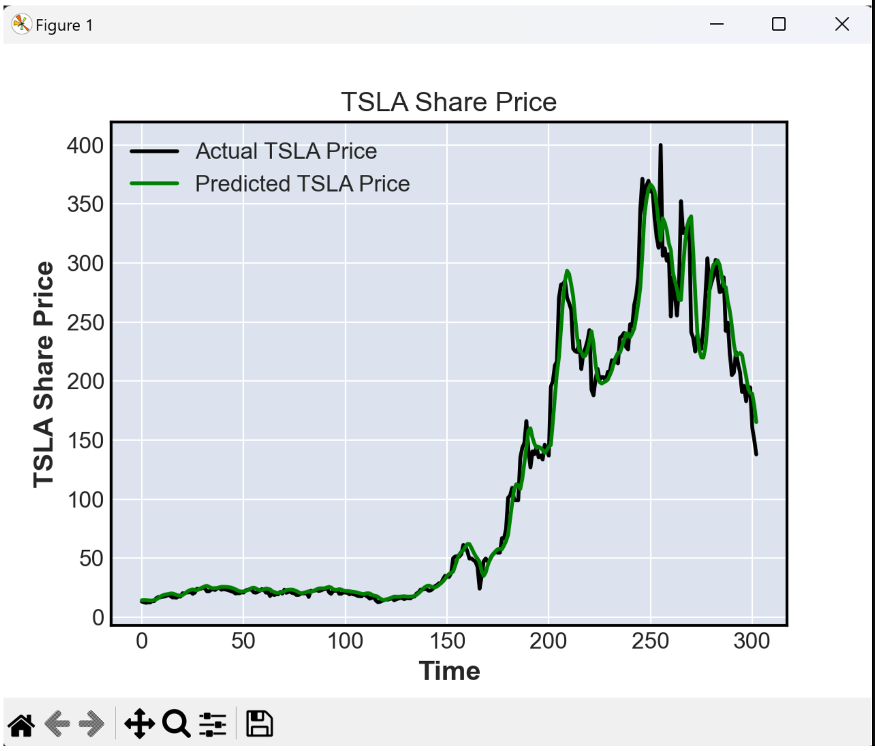


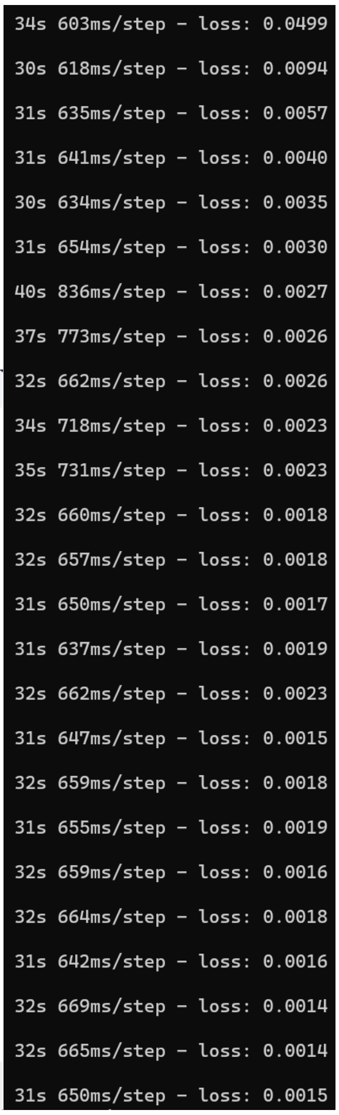
```
31s 562ms/step - loss: 0.0624
28s 580ms/step - loss: 0.0401
28s 578ms/step - loss: 0.0355
28s 581ms/step - loss: 0.0329
28s 586ms/step - loss: 0.0306
30s 626ms/step - loss: 0.0317
32s 660ms/step - loss: 0.0292
29s 605ms/step - loss: 0.0278
29s 608ms/step - loss: 0.0276
32s 669ms/step - loss: 0.0271
31s 636ms/step - loss: 0.0273
30s 630ms/step - loss: 0.0259
30s 621ms/step - loss: 0.0239
30s 619ms/step - loss: 0.0245
30s 630ms/step - loss: 0.0242
30s 621ms/step - loss: 0.0235
30s 621ms/step - loss: 0.0240
30s 620ms/step - loss: 0.0234
29s 608ms/step - loss: 0.0239
31s 648ms/step - loss: 0.0235
30s 622ms/step - loss: 0.0226
29s 611ms/step - loss: 0.0231
29s 612ms/step - loss: 0.0216
29s 606ms/step - loss: 0.0219
30s 615ms/step - loss: 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.

```
#set 2
# Set the model parameters
units = [256, 128, 64]
cells = ['LSTM', 'GRU', 'SimpleRNN']
n_layers = 3
dropout = 0.2
loss = "mean_squared_error"
optimizer = "adam"
bidirectional = True

# Set the training parameters
epochs = 25
batch_size = 32
```

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.



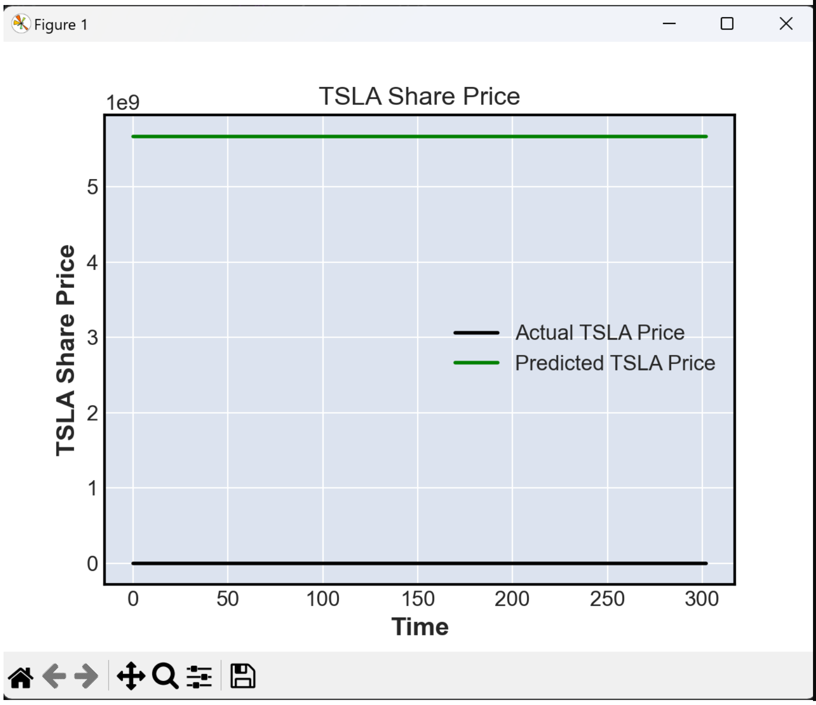


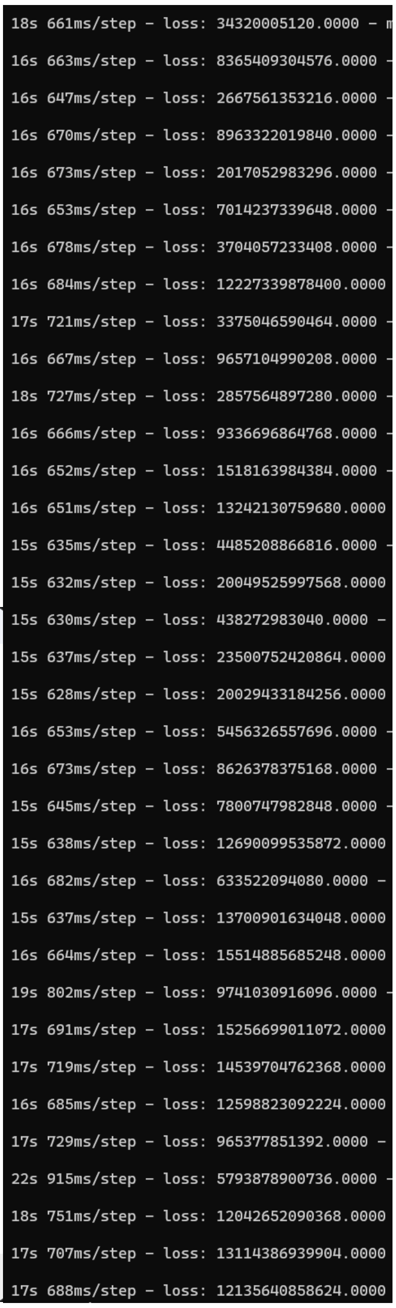
```
34s 603ms/step - loss: 0.0499
30s 618ms/step - loss: 0.0094
31s 635ms/step - loss: 0.0057
31s 641ms/step - loss: 0.0040
30s 634ms/step - loss: 0.0035
31s 654ms/step - loss: 0.0030
40s 836ms/step - loss: 0.0027
37s 773ms/step - loss: 0.0026
32s 662ms/step - loss: 0.0026
34s 718ms/step - loss: 0.0023
35s 731ms/step - loss: 0.0023
32s 660ms/step - loss: 0.0018
32s 657ms/step - loss: 0.0018
31s 650ms/step - loss: 0.0017
31s 637ms/step - loss: 0.0019
32s 662ms/step - loss: 0.0023
31s 647ms/step - loss: 0.0015
32s 659ms/step - loss: 0.0018
31s 655ms/step - loss: 0.0019
32s 659ms/step - loss: 0.0016
32s 664ms/step - loss: 0.0018
31s 642ms/step - loss: 0.0016
32s 669ms/step - loss: 0.0014
32s 665ms/step - loss: 0.0014
31s 650ms/step - loss: 0.0015
```

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.

```python
# Set the model parameters
units = [512, 256]
cells = ['GRU', 'GRU']
n_layers = 2
dropout = 0.4
loss = "mean_absolute_percentage_error"
optimizer = "sgd"
bidirectional = False

# Set the training parameters
epochs = 35
batch_size = 64
```

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.



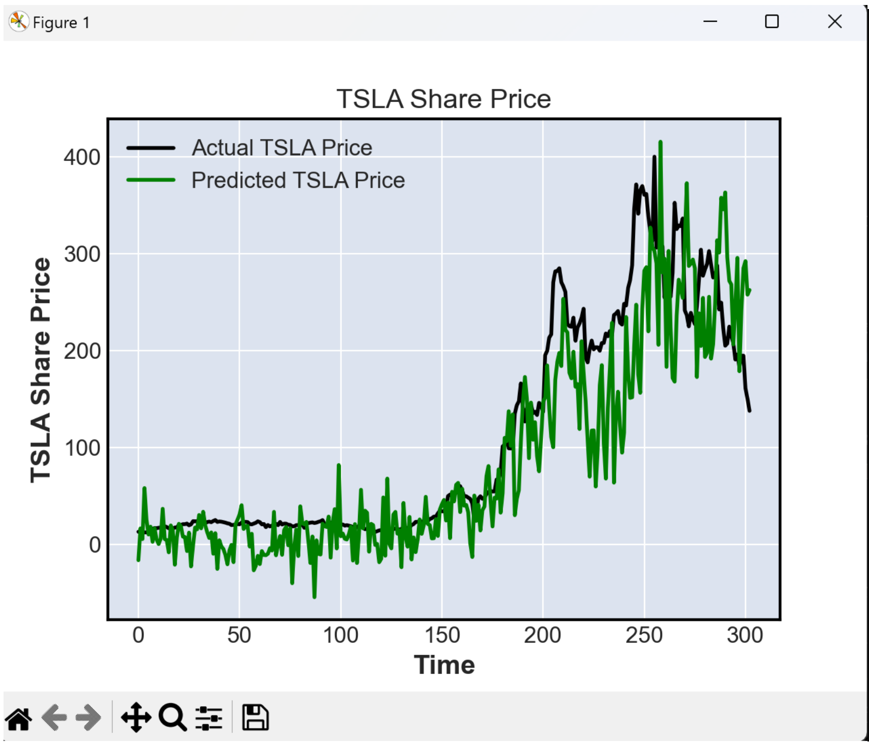


```
18s 661ms/step - loss: 34320005120.0000 - m
16s 663ms/step - loss: 8365409304576.0000 -
16s 647ms/step - loss: 2667561353216.0000 -
16s 670ms/step - loss: 8963322019840.0000 -
16s 673ms/step - loss: 2017052983296.0000 -
16s 653ms/step - loss: 7014237339648.0000 -
16s 678ms/step - loss: 3704057233408.0000 -
16s 684ms/step - loss: 12227339878400.0000
17s 721ms/step - loss: 3375046590464.0000 -
16s 667ms/step - loss: 9657104990208.0000 -
18s 727ms/step - loss: 2857564897280.0000 -
16s 666ms/step - loss: 9336696864768.0000 -
16s 652ms/step - loss: 1518163984384.0000 -
16s 651ms/step - loss: 13242130759680.0000
15s 635ms/step - loss: 4485208866816.0000 -
15s 632ms/step - loss: 20049525997568.0000
15s 630ms/step - loss: 438272983040.0000 -
15s 637ms/step - loss: 23500752420864.0000
15s 628ms/step - loss: 20029433184256.0000
16s 653ms/step - loss: 5456326557696.0000 -
16s 673ms/step - loss: 8626378375168.0000 -
15s 645ms/step - loss: 7800747982848.0000 -
15s 638ms/step - loss: 12690099535872.0000
16s 682ms/step - loss: 633522094080.0000 -
15s 637ms/step - loss: 13700901634048.0000
16s 664ms/step - loss: 15514885685248.0000
19s 802ms/step - loss: 9741030916096.0000 -
17s 691ms/step - loss: 15256699011072.0000
17s 719ms/step - loss: 14539704762368.0000
16s 685ms/step - loss: 12598823092224.0000
17s 729ms/step - loss: 965377851392.0000 -
22s 915ms/step - loss: 5793878900736.0000 -
18s 751ms/step - loss: 12042652090368.0000
17s 707ms/step - loss: 13114386939904.0000
17s 688ms/step - loss: 12135640858624.0000
```

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.

```
# Set the model parameters
units = [128, 64, 32]
cells = ['SimpleRNN', 'SimpleRNN', 'SimpleRNN']
n_layers = 3
dropout = 0.5
loss = "huber_loss"
optimizer = "adagrad"
bidirectional = True

# Set the training parameters
epochs = 15
batch_size = 16
```

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.



```
10s 75ms/step - loss: 0.2906
9s 97ms/step - loss: 0.2249
8s 85ms/step - loss: 0.1942
8s 87ms/step - loss: 0.1673
8s 81ms/step - loss: 0.1515
7s 79ms/step - loss: 0.1562
8s 81ms/step - loss: 0.1411
8s 83ms/step - loss: 0.1408
8s 85ms/step - loss: 0.1234
7s 78ms/step - loss: 0.1324
7s 76ms/step - loss: 0.1120
7s 77ms/step - loss: 0.1329
7s 77ms/step - loss: 0.1175
7s 79ms/step - loss: 0.1250
7s 77ms/step - loss: 0.1139
1s 28ms/step
```

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.