Report Week 6 – Task B.4

# Create Model Function

As indicated by the task requirements, I have developed my `create\_model` function predominantly based on the `create\_model` function from the P1 base. To provide clarity, I will first outline the functionality of the original method. Initially, the original function accepts two input parameters: `sequence\_length` and `n\_features`, which define the dimensions of the test data. Additionally, it incorporates the following elements:

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):

• 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 initiates by instantiating the Sequential model class, which serves as a linear arrangement of layers suitable for constructing a neural network. Subsequently, it proceeds into a for loop that cycles through the quantity of layers defined by the n\_layers parameter. Within this loop, the function executes a series of steps for each layer:

# Create a Sequential model

    model = Sequential()

    # Loop over the number of layers

    for i in range(n\_layers):

1. The function determines whether the current layer is the initial layer by assessing if the loop index `i` equals 0. If it is indeed the first layer, the function incorporates a recurrent layer of the type indicated by the `cell` parameter (such as LSTM, GRU, or SimpleRNN) and specifies the number of units using the `units` parameter. The input shape for this layer is defined as `(None, sequence\_length, n\_features)`, where `None` serves as a placeholder for the batch size, which will be established during the model fitting process.

# If this is the first layer...

        if i == 0:

            if bidirectional:

                model.add(Bidirectional(cell(unit, return\_sequences=True), batch\_input\_shape=(None, sequence\_length, n\_features)))

            else:

                model.add(cell(unit, return\_sequences=True, batch\_input\_shape=(None, sequence\_length, n\_features)))

1. When the current layer is not the first, the function determines whether it is the last layer by evaluating if the loop index `i` equals `n\_layers - 1`. If it is indeed the last layer, the function introduces an additional recurrent layer of the type defined by the `cell` parameter, utilizing the number of units indicated by the `units` parameter. In this instance, the `return\_sequences` argument is configured to `False`, ensuring that only the final output from this layer is provided.

# If this is the last layer...

        elif i == n\_layers - 1:

            if bidirectional:

                model.add(Bidirectional(cell(unit, return\_sequences=False)))

            else:

                model.add(cell(unit, return\_sequences=False))

1. When the existing layer is not the first or the last, it is categorized as a hidden layer. In this scenario, the function introduces an additional recurrent layer defined by the cell parameter, utilizing the quantity of units indicated by the units parameter. The input shape for this layer is configured as (None, sequence\_length, n\_features).

else:

            if bidirectional:

                model.add(Bidirectional(cell(unit, return\_sequences=True)))

            else:

                model.add(cell(unit, return\_sequences=True))

4. Prior to incorporating each recurrent layer, if the parameter `bidirectional` is set to `True`, the layer is encapsulated within a Bidirectional wrapper, thereby transforming it into a bidirectional layer.

5. Following the addition of each recurrent or bidirectional layer, the function evaluates the value of `bidirectional`. If it is `True`, a Dropout layer is appended with the rate defined by the `dropout` parameter; if `False`, the Dropout layer is still added accordingly. This is executed with the command:

# Add dropout after each layer

        model.add(Dropout(dropout))

Upon completion of the for loop and the integration of all layers into the model, a final Dense output layer with a single unit and linear activation is appended. Subsequently, the model is compiled using the `compile` method, with the loss function and optimizer determined by the `loss` and `optimizer` parameters. Ultimately, the model is returned.

model.add(Dense(1, activation="linear"))

    # Compile the model with the specified loss function and optimizer

    model.compile(loss=loss, metrics=["mean\_absolute\_error"], optimizer=optimizer)

    # Return the compiled model

    return model

# Function Adjustment

To integrate it into my current project code, I have adapted the P1 function to align with the task specifications and to incorporate additional functionalities. Generally, the P1 function meets the requirements outlined in the specifications; however, I have implemented several beneficial modifications.

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 have modified the `units` parameter from a singular integer to a list of integers. This change enables the specification of varying unit counts for each layer within the model. Previously, the original function enforced uniformity across all layers, as they relied on the same integer value defined by the `units` parameter. Consequently, instead of directly utilizing this singular integer, the updated approach employs indexing to retrieve the relevant element from the `units` list for each individual layer.

# Get the number of units for this layer

        unit = units[i]

1. I’ve added a new cells parameter, which is a list of strings that specify the type of each layer inthe 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 parameterdirectly it now uses indexing to access the appropriate element from the cells list for each layer.

cell\_name = cells[i]

1. I have modified the method by which the function obtains a reference to the relevant layer network object associated with the cell name. Rather than directly utilizing the cell parameter's value, which is an object, the function now employs the `globals()` function. This adjustment allows it to retrieve the corresponding layer network object based on the string value provided for each layer.

cell = globals()[cell\_name]

1. 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.

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

The previous approach to model creation, represented by the original v0.1 code, can now be substituted with a more efficient function. This updated method allows for straightforward parameter configuration, where `sequence\_length` and `n\_features` are extracted from the dimensions of the training dataset. Additionally, the types of layers in the network and their respective unit sizes are defined within lists. After setting these parameters, the create model function is executed, followed by fitting the model to the training data, specifying the number of epochs and batch size during the process.

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

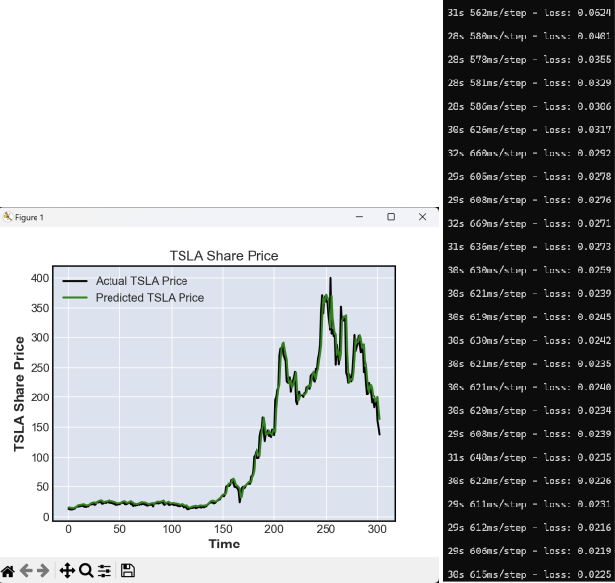
epochs = 25

batch\_size = 32

# Train the model on the training data

model.fit(data['X\_train'], data['y\_train'], epochs=epochs, batch\_size=batch\_size)

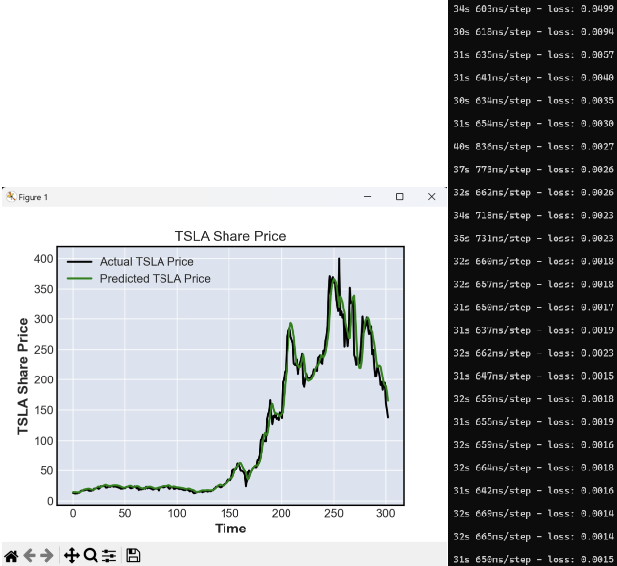
This initial test uses two layers, an LTSM with 256 unit size, and a GRU with 128 unit size. It also keeps the  
standard 25 epochs and 32 batch size of the original v0.1 base. This predicts quite an accurate result to  
the test data, with on average 30 seconds per epoch, and loss between 0.0624 and 0.022



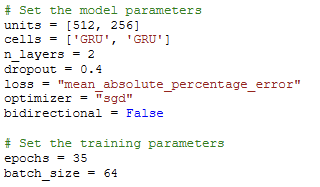
The next test set uses the following parameters. This notably is using three layers of different types, with  
decreasing unit sizes. It is also testing a different optimizer method.

#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.



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 screenshot of a computer screen

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.

A screenshot of a computer program

Description automatically generated

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 graph showing a price

Description automatically generated with medium confidence

Overall, these findings indicate that various configurations for model development and fitting can significantly influence the performance and precision of the final model and its predictions. It is crucial for practitioners in machine learning to comprehend the distinctions among the available configuration options and their potential effects on outcomes. This understanding is essential for selecting parameters that yield the desired results efficiently.