COS30018 Intelligent Systems – Task B.4

Ryan Chessum

This week I needed to create a function that can make an Deep learning model. The function has many parameters so we can make a model that fits the specifications that we need.

Here is the function:

```
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):

    #Variables:
    ###################
    # sequence length: no of days for lookback
    # n_features:no of feature columns
    # units:number of nodes in each layer
    # cell: cell type
    # n_layers: number of layers
    # dropout: drop out frequency
    # loss: loss metric
    # optimizer: what optimizer we want to use
    # bidirectional: bidirectional y/n
    ####################################

    #create a sequential model
    model = Sequential()

    #Create amount of layers equal to n_layers
    for i in range(n_layers):
        if i == 0:
            # first layer
            if bidirectional:
                #set the input layer if first layer
                model.add(Bidirectional(cell(units, return_sequences=True), batch_input_shape=(None, sequence_length, n_features)))
            else:
                #if not bidirectional
                model.add(cell(units, return_sequences=True, batch_input_shape=(None, sequence_length, n_features)))
        elif i == n_layers - 1:
            # last layer
            if bidirectional:
                #no return sequences on last layer
                model.add(Bidirectional(cell(units, return_sequences=False)))
            else:
                model.add(cell(units, return_sequences=False))
        else:
            if bidirectional: #if bidirectional
                model.add(Bidirectional(cell(units, return_sequences=True)))
            else:#if not bidirectional
                model.add(cell(units, return_sequences=True))
        # add dropout after each layer
        model.add(Dropout(dropout))
    #final layer, specify linear activation function
    model.add(Dense(1, activation="linear"))
    #compire the model
    model.compile(loss=loss, metrics=["mean_absolute_error"], optimizer=optimizer)
    return model
```

The first two parameters (sequence length and n_features), are to specify the shape of the input layer. The model must know the exact kind of data it must be fed otherwise an error will occur. While trying to set up a model, I had an error where the data I wasing giving the model didn't match the value I had specified and so the program could not run. Sequence length is the amount of days to look back to make a prediction and n_features is the amount of feature columns in the data.

Next units is the amount of nodes in each hidden layer, cell specifies the cell type for each layer, n_layers specifies the number of layers for the model, dropout specifies the dropout frequency, loss

specifies the loss metric, optimizer is the optimizer we want to use and bidirectional lets us specify weather or not we want a bidirectional model.

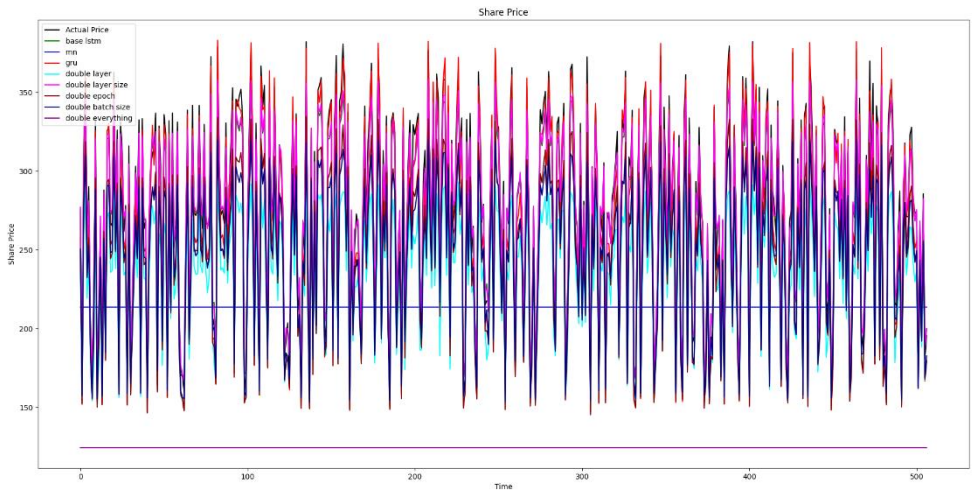
When the function is called, it starts a loop. In each cycle of the loop it creates each layer of the neural network. The number of layers it creates is equal to the number of layers specified in the parameters.

If it's the first layer of the network, the model will create the input layer. If it is the last layer it will also set return sequences to false for that layer. Every other layer is made normally. At the end of the loop, a dropout layer is added so that there is a dropout layer after each main layer. Finally a dense layer is added at the end which has a linear activation function. At the end the model is compiled and returned.

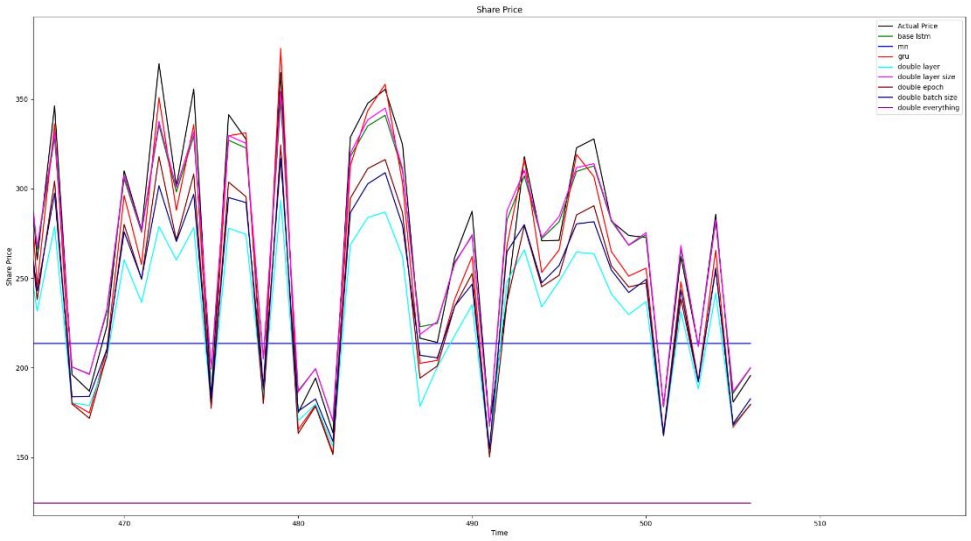
Because deep learning algorithms act like a black box, it can be hard to determine if a model is effective unless we try out different methods and compare the results. Thankfully this function lets us do that very easily. Using the function, I created 8 different models using the same data and plotted the results so we can compare them.

There is a base lstm model, a simple rnn model, a gru model, an lstm model with double the amount of layers, a lstm model with double layer sizes, a layer that is trained with double the amount of epochs, a model with double batch size and an lstm model with everything doubled.

Here are the results:



It is a bit hard to see clearly but it seems as though double batch size and by extension double everything did not work.



Taking a closer look we can see that most models predict a lot lower than the actual price but overall the gru model actually performs the best in this test.