Task 4 - Machine Learning 1

Kelvin Dang - 104776732 - Swinburne University of Technology

Overview

In this week's task, I was challenged with writing a function returning a Deep Learning model, with given several parameters as inputs, including the number of layers, the size of each, and the layer name, as well as experimenting with different DL networks (e.g., LSTM, GRU, RNN, etc.) and with different hyperparameters setups.

Model Creation

I was able to implement this function thanks to TensorFlow alongside Keras, one of the most popularly used library for developing and training DL models. With this inclusion of a new library, I customized my `requirements.txt` as follows, before creating a new virtual environment in place of the previous one, and kicking off my implementation process.

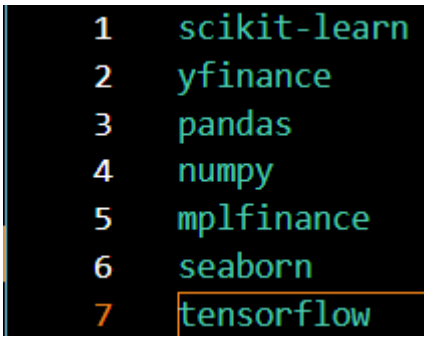

Figure 1: The modified `requirements.txt` file, with the newest inclusion of TensorFlow.

Similar to the design of P1, I chose stacked LSTM (aka. Long Short-Term Memory) as the starting point of the model design function, with the choice of other DL layers such as SimpleRNN (simple recurrent neural network), GRU (gated recurrent units) still open. The reason for this is because LSTM has many advantages over other types of recurrent neural networks, for example, the ability of avoiding the vanishing gradient problem, and its insensitivity to

gap length (Hochreiter & Schmidhuber, 1997). The following figure shows the structure of a LSTM cell.

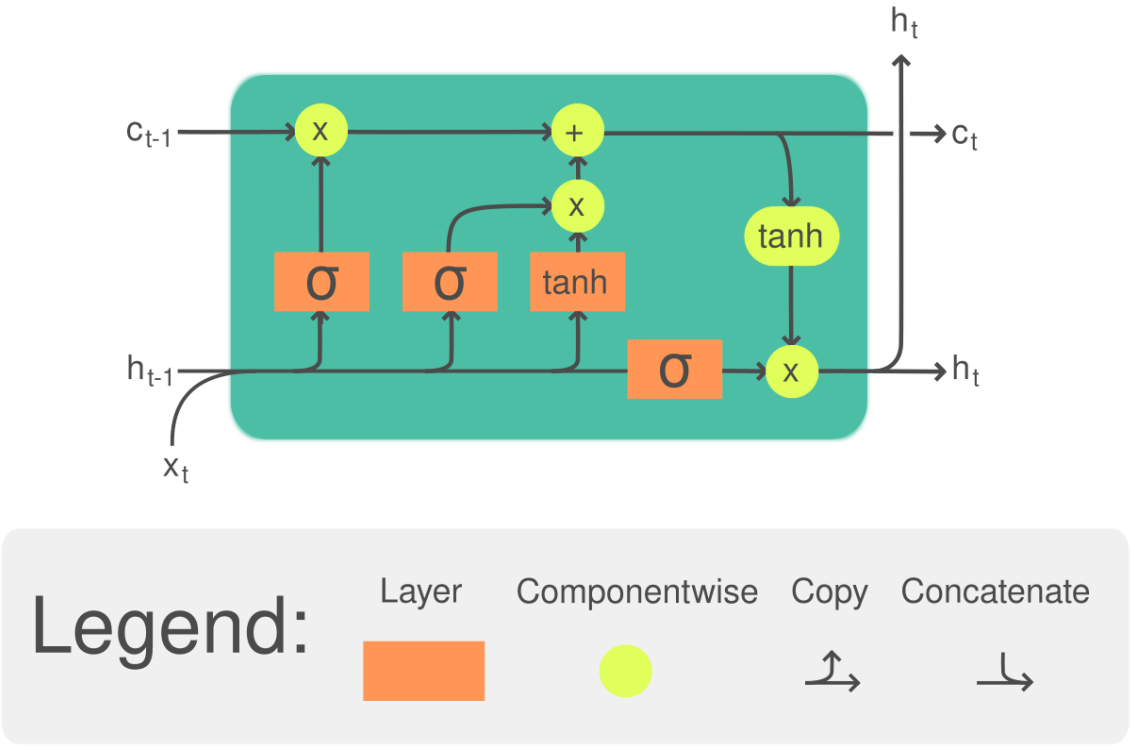

Figure 2: The structure of a single LSTM cell.

Here is the complete implementation of the function dedicated to model creation, with reference to the one from P1 plus some modifications:

```python
def create_model(sequence_length, n_features, units=256, layer=LSTM, n_layers=2, dropout=0.3,
                 loss="mean_absolute_error", optimizer="rmsprop", bidirectional=False):
    # Initialize the model.
    model = Sequential()
    for i in range(n_layers):
        if i == 0:
            # Create the input layer.
            if bidirectional:
                model.add(Bidirectional(layer(units, return_sequences=True),
                                        input_shape=(sequence_length, n_features)))
            else:
                model.add(layer(units, return_sequences=True,
                                input_shape=(sequence_length, n_features)))
        elif i == n_layers - 1:
            # Create the last layer.
            if bidirectional:
                model.add(Bidirectional(layer(units, return_sequences=False)))
            else:
                model.add(layer(units, return_sequences=False))
        else:
            # Create hidden layers.
            if bidirectional:
                model.add(Bidirectional(layer(units, return_sequences=True)))
            else:
                model.add(layer(units, return_sequences=True))
        # Add dropout after each layer.
        model.add(Dropout(dropout))
    # Create the output layer.
    model.add(Dense(1, activation="linear"))
    # Compile the model.
    model.compile(loss=loss, metrics=["mae"], optimizer=optimizer)
    return model
```

Figure 3: The function for model creation.

The function takes in the sequence length (i.e., the number of previous days' features used to predict the following ones) and the number of features (i.e., the price values) as inputs, as well as the type of DL layer, number of layers stacked together, the choice to use bidirectional RNN or otherwise, along with several others.

Note that every recurrent layer has its `return_sequences` parameter set to True, except the last one, which directly leads to the output dense layer. The reason for this is that it makes the respective layer to return the full sequence of hidden state for each timestep in the input, rather than just the final output, as each recurrent layer requires a complete sequence as input. Because I chose to stack multiple LSTM layers together, before adding the last dense output layer, this configuration is necessary for all but the last LSTM layer.

I also applied dropout to every LSTM layer. According to the official documentation of TensorFlow, dropout, which is one of the most effective and

popularly used method to mitigate the risk of overfitting, is the process of randomly having a number of output features of the respective layer set to 0 during training. This helps the network to ignore spurious patterns of the training data, hence substantially reducing the risk of overfitting.

In the process of implementing this function, I encountered a problem. The input shape of the LSTM layers in P1's function is set to `(None, sequence_length, n_features)`, resulted in a `ValueError` shown below.

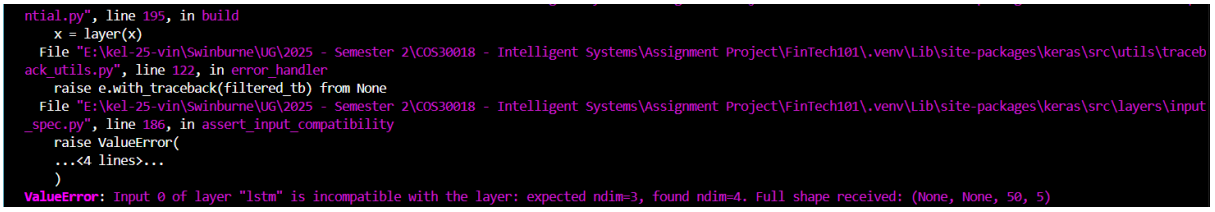

Figure 4: The `ValueError` when setting the incorrect input dimension.

I think this problem is caused by the fact that during the process of working in the previous task, I also modified the `yf.download()` function call in order to eliminate the multilevel index of the downloaded DataFrame, which has the synonymous shape to the input training data. Having set it to `(sequence_length, n_features)`, this issue was effectively resolved, and the model creation function run successfully.

Before going to the upcoming process of model training, I added new parameters, which is the input parameters of the model creation function, to the `parameters.py` module as follows.

```
# Model options.
LOSS_FUNCTION = "huber"  # Options: "mse", "mae", "huber"
UNITS = 256
LAYER = LSTM  # Options: "LSTM", "GRU", "SimpleRNN"
N_LAYERS = 2
DROPOUT = 0.3
OPTIMIZER = "adam"  # Options: "adam", "rmsprop", "sgd"
BIDIRECTIONAL = False
```

Figure 5: Model parameters.

Model Training

Having implemented the model creation function, I added a single call to the newly implemented function to the `train.py` module. Before training the model, I would have to include some more parameters regarding the training configurations such as batch size and number of epochs to the `parameters.py` module.

```
# Training options.
BATCH_SIZE = 64
EPOCHS = 500
```

Figure 6: Training parameters, including batch size and number of epochs.

With that done, I was ready to train the model.

```
# Construct the model.
model = create_model(HISTORY_DAYS, len(FEATURE_COLUMNS),
                     loss=LOSS_FUNCTION, units=UNITS, layer=LAYER, n_layers=N_LAYERS,
                     dropout=DROPOUT, optimizer=OPTIMIZER, bidirectional=BIDIRECTIONAL)

# Train the model.
history = model.fit(data["X_train"], data["y_train"],
                    batch_size=BATCH_SIZE, epochs=EPOCHS,
                    validation_data=(data["X_test"], data["y_test"]),
                    verbose=1)
```

Figure 7: Model creation and training.

```
ter 2/COS30018 - Intelligent Systems/Assignment Project/FinTech101/.venv/Scripts/python.exe" "e:/kel-25-vin/Swi
t Systems/Assignment Project/FinTech101/train.py"
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 105ms/step - loss: 3.6165e-04 - mae: 0.0210 - val_loss: 0.0022 - val_mae: 0.0593
Epoch 4/500
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 101ms/step - loss: 2.8282e-04 - mae: 0.0184 - val_loss: 0.0015 - val_mae: 0.0464
Epoch 5/500
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 103ms/step - loss: 2.8016e-04 - mae: 0.0183 - val_loss: 0.0014 - val_mae: 0.0459
Epoch 6/500
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 104ms/step - loss: 2.2763e-04 - mae: 0.0165 - val_loss: 0.0014 - val_mae: 0.0466
Epoch 7/500
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 111ms/step - loss: 2.6595e-04 - mae: 0.0175 - val_loss: 0.0015 - val_mae: 0.0477
Epoch 8/500
15/16 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - loss: 2.7696e-04 - mae: 0.0181
```

Figure 8: Successful output of the training process.

This successful output concludes the implementation of model creation and training processes.

Experiments of Hyperparameters Configurations

Number of epochs	Training loss (Huber)	Validation loss (Huber)	Training loss (MAE)	Validation loss (MAE)
50	1.8601e-04	0.0010	0.0149	0.0392
100	1.1828e-04	6.7845e-04	0.0118	0.0321
200	8.3286e-05	1.6584e-04	0.0099	0.0144
500	5.4835e-05	1.7128e-04	0.0080	0.0141

Types of DL layers (set at 500 epochs)	Training loss (Huber)	Validation loss (Huber)
SimpleRNN	4.8012e-05	8.4872e-04
GRU	5.5874e-05	7.4896e-04

References

- Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation, 9(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735
- The official documentation of TensorFlow, on https://www.tensorflow.org.