Task B4: Machine Learning 1

Dr Ru Jia
Friday, 14:30 – 16:30 class

Name: Min Khant Aung
ID: 103833225

Task 4 Report: Deep Learning Model Implementation and Experimentation

1. Introduction

In the context of stock market prediction, constructing effective Deep Learning (DL) models is crucial for capturing complex temporal dependencies. This report outlines the implementation and experimentation process involved in creating DL models with varying architectures to improve stock prediction accuracy.

2. Implementation of the Function

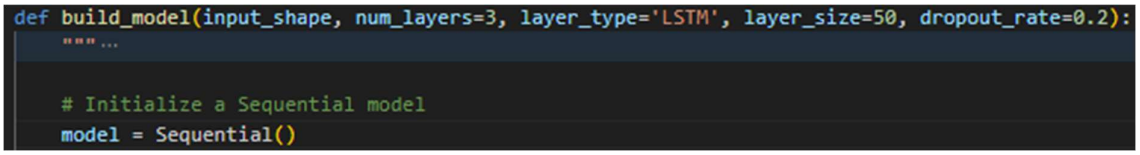
2.1 Overview

The build_model function was implemented to dynamically create DL models based on user-specified parameters. The function allows for flexibility in defining the network architecture, including the number of layers, size of each layer, and type of layers (e.g., LSTM, GRU, RNN, BiLSTM, BiGRU).

2.2 Code Explanation

Changes in model_operations.py

Function Definition



- **input_shape**: Shape of the input data.
- **num_layers**: Number of layers in the network.
- **layer_type**: Type of RNN layer to use (LSTM, GRU, RNN, BiLSTM, BiGRU).
- **layer_size**: Number of units in each layer.
- **dropout_rate**: Dropout rate to prevent overfitting.

Adding Layers

```
if layer_type == 'GRU':
    model.add(GRU(units=layer_size, return_sequences=(num_layers > 1), input_shape=input_shape))
elif layer_type == 'BiLSTM':
    model.add(Bidirectional(LSTM(units=layer_size, return_sequences=(num_layers > 1)), input_shape=input_shape))
elif layer_type == 'RNN':
    model.add(SimpleRNN(units=layer_size, return_sequences=(num_layers > 1), input_shape=input_shape))
elif layer_type == 'LSTM':
    model.add(LSTM(units=layer_size, return_sequences=(num_layers > 1), input_shape=input_shape))
elif layer_type == 'BiGRU':
    model.add(Bidirectional(GRU(units=layer_size, return_sequences=(num_layers > 1)), input_shape=input_shape))
else:
    # Raise an error if an unsupported layer type is provided
    raise ValueError(f"Unsupported layer_type: {layer_type}")
```

- **Layer Addition**: The first layer is added based on the specified type and configuration. return_sequences is set to True if more than one layer is present.

Adding Dropout and Additional Layers

```
# Apply dropout for regularization
model.add(Dropout(dropout_rate))

# Add the remaining RNN layers (if num_layers > 1)
# Each subsequent layer should consider whether to return sequences to the next layer
for _ in range(1, num_layers):
    if layer_type == 'GRU':
        model.add(GRU(units=layer_size, return_sequences=(_ < num_layers - 1)))
    elif layer_type == 'BiLSTM':
        model.add(Bidirectional(LSTM(units=layer_size, return_sequences=(_ < num_layers - 1))))
    elif layer_type == 'RNN':
        model.add(SimpleRNN(units=layer_size, return_sequences=(_ < num_layers - 1)))
    elif layer_type == 'LSTM':
        model.add(LSTM(units=layer_size, return_sequences=(_ < num_layers - 1)))
    elif layer_type == 'BiGRU':
        model.add(Bidirectional(GRU(units=layer_size, return_sequences=(_ < num_layers - 1))))

    # Apply dropout after each RNN layer for regularization
    model.add(Dropout(dropout_rate))
```

- **Dropout**: Added after each layer to prevent overfitting.
- **Additional Layers**: Layers are added in a loop, with return_sequences set based on the layer's position.

Output Layer and Compilation

```
# Add the final output layer, which produces a single value as the prediction
model.add(Dense(units=1))

# Compile the model using the Adam optimizer and mean squared error loss
model.compile(optimizer='adam', loss='mean_squared_error')

return model
```

Output Layer: A dense layer with one unit for regression tasks.

- **Compilation**: The model is compiled with the Adam optimizer and mean squared error loss function.

Changes in main.py

```
model = build_model((x_train.shape[1], len(FEATURE_COLUMNS)), num_layers=5,
layer_type='BiLSTM', layer_size=100, dropout_rate=0.3)
```

In the main script,(currently) I modified the model initialization by using the build_model function with a *BiLSTM architecture*. Specifically, I set the model to have **5 layers**, with each layer containing **100 units**. Additionally, I increased the **dropout rate to 0.3** to help reduce overfitting. The input shape was defined as (x_train.shape[1], len(FEATURE_COLUMNS)) to ensure compatibility with the feature dimensions in the dataset.

Experiments.py for testing with different configurations

In the model_experiments.py script, the following steps are carried out:

- **Model Configurations:**

```
model_types = ['LSTM', 'GRU', 'BiLSTM']  # Types of RNNs to experiment with
layers_config = [2, 3, 4]  # Different number of layers to experiment with
units_config = [50, 100, 150]  # Different numbers of units per layer to
experiment with
epochs_config = [25, 50]  # Different numbers of training epochs to
experiment with
batch_size_config = [32, 64]  # Different batch sizes for training
```

These configurations define the range of experiments to be conducted. The script will loop through each combination of model type, number of layers, units per layer, epochs, and batch size.

- **Loading and Preparing Data:**

```
data = load_data(COMPANY, TRAIN_START, TRAIN_END, nan_handling=NAN_METHOD,
fill_value=FILL_VALUE,
                 cache_dir=CACHE_DIR, use_cache=USE_CACHE)
x_train, y_train, x_test, y_test, scalers = prepare_data(data,
FEATURE_COLUMNS, PREDICTION_DAYS,
                                                         split_method=SPLIT_METHOD,
split_ratio=SPLIT_RATIO,
                                                              random_split=RANDOM_S
PLIT)
```

This section loads and prepares the stock market data for training and testing, using functions from data_processing.py. The dataset is split into training and testing sets, and features are scaled.

- **Building the Model:**

```
• model = build_model(input_shape=input_shape, num_layers=num_layers,
  layer_type=model_type, layer_size=layer_size)
```

For each configuration, the build_model function (from model_operations.py) is called to dynamically create the model based on the specified type (LSTM, GRU, BiLSTM), number of layers, and units per layer.

- **Training and Timing the Model:**

```
 # Measure the time it takes to train the model.
                    start_time = time.time()
                    trained_model = train_model(model, x_train, y_train,
epochs=epochs, batch_size=batch_size)
                    training_time = time.time() - start_time  #Calculate
the training duration.
```

The train_model function is used to train the model on the training data. The training time is recorded to evaluate the efficiency of different models and configurations.

- **Evaluating the Model:**

```
# Evaluate the model's performance on both the training and validation (test)
data.
train_loss = trained_model.evaluate(x_train, y_train, verbose=0)  # MSE on
training data
val_loss = trained_model.evaluate(x_test, y_test, verbose=0)  # MSE on
validation (test) data
```

After training, the model's performance is measured using the Mean Squared Error (MSE) on both the training and validation sets.

- **Storing the Results:**

```
results.append({
'Model Type': model_type,  # Type of the model (LSTM, GRU, or BiLSTM)
'Layers': num_layers,  # Number of layers in the model
'Units per 1 layer': layer_size,  # Number of units in each layer
'Epochs': epochs,  # Number of epochs used for training
'Batch Size': batch_size,  # Batch size used for training
'Training Loss': train_loss,  # MSE on training data
```

```
'Validation Loss': val_loss,  # MSE on test data
'Time Taken (s)': training_time  # Time taken for training (in seconds)
                    })
```

The results of each experiment, including the model type, layer configurations, losses, and training time, are stored in a list. Finally, the results are saved to a CSV file for further analysis:

```
# Convert the results list to a DataFrame for easy analysis.
data = pd.DataFrame(results)

# Save the results to a CSV file for future reference and analysis.
data.to_csv("experiment_results.csv", index=False)
```

This experimental setup allows for testing multiple DL models and hyperparameters efficiently. Each configuration's results are saved in a CSV file, making it easy to compare models based on different criteria such as accuracy and training time.

3. Experimentation and Results

In this section, we examine the performance of different deep learning models trained to predict time series data, using LSTM, GRU, and BiLSTM architectures. The experiment compares various configurations by adjusting parameters such as the number of layers, units per layer, epochs, and batch sizes. Key metrics like training loss, validation loss, and the time taken are evaluated to assess model performance.

3.1 Experimental Setup

To evaluate the performance of various Deep Learning (DL) models for stock prediction on **experiments.py**, I conducted experiments using different configurations of LSTM, GRU, and BiLSTM models. The experiments varied in the following parameters:

- **Model Types:** LSTM, GRU, BiLSTM
- **Number of Layers:** 2, 3, 4
- **Units per Layer:** 50, 100, 150
- **Epochs:** 25, 50
- **Batch Size:** 32, 64

The dataset was split into training and validation sets, and each model was trained using the train_model function from model_operations.py. The primary performance

metrics were **Training Loss**, **Validation Loss**, and **Training Time**. The performance was evaluated using the Mean Squared Error (MSE) on both the training and validation data.

*** MSE** is a commonly used metric for regression tasks, where a lower value indicates better model accuracy. *

3.2 Results

LSTM:

- LSTM models generally performed well, with validation losses often remaining lower than training losses, indicating good generalization.
- Increasing the number of layers or units typically improved performance but also led to an increase in training time.
- For example, the 4-layer LSTM with 150 units and a batch size of 64 achieved the lowest validation loss of **0.00264567** in **308.29 seconds**. However, models with fewer layers, such as a 2-layer LSTM with 50 units, also showed good performance with a low validation loss of **0.002957692** in **38.77 seconds**.

1	Model Type	Layers	Units per 1	Epochs	Batch Size	Training Lo	Validation	Time Taken
2	LSTM	2	50	25	32	0.0050662	0.0034022	16.652139
3	LSTM	2	50	25	64	0.0056293	0.0050814	12.519036
4	LSTM	2	50	50	32	0.0028242	0.0029577	38.772944
5	LSTM	2	50	50	64	0.0047486	0.0043378	30.300042
6	LSTM	2	100	25	32	0.0043601	0.0047602	38.620874
7	LSTM	2	100	25	64	0.0048148	0.0033334	48.017765
8	LSTM	2	100	50	32	0.0027395	0.0031987	74.114137
9	LSTM	2	100	50	64	0.0049432	0.0031662	63.705364
10	LSTM	2	150	25	32	0.0051827	0.0036196	68.130288
11	LSTM	2	150	25	64	0.0046849	0.0031942	49.742477
12	LSTM	2	150	50	32	0.0036295	0.0065838	145.71643
13	LSTM	2	150	50	64	0.0045808	0.0033167	93.919813
14	LSTM	3	50	25	32	0.0050136	0.0050894	31.010823
15	LSTM	3	50	25	64	0.0050263	0.0033606	25.06718
16	LSTM	3	50	50	32	0.0029962	0.0046403	56.62285
17	LSTM	3	50	50	64	0.0048704	0.0049372	47.264396
18	LSTM	3	100	25	32	0.0044337	0.0031711	61.47998
19	LSTM	3	100	25	64	0.0057446	0.0062139	55.619054
20	LSTM	3	100	50	32	0.0030158	0.004997	118.72192
21	LSTM	3	100	50	64	0.0037666	0.0076694	107.62675
22	LSTM	3	150	25	32	0.0047491	0.0032746	315.14852
23	LSTM	3	150	25	64	0.0050697	0.0038949	83.22845
24	LSTM	3	150	50	32	0.002414	0.0031411	256.94857
25	LSTM	3	150	50	64	0.0034442	0.0070274	167.14187
26	LSTM	4	50	25	32	0.0053082	0.0036536	40.976237
27	LSTM	4	50	25	64	0.0053617	0.0036594	35.403605
28	LSTM	4	50	50	32	0.0032923	0.0030315	76.927442
29	LSTM	4	50	50	64	0.0043925	0.0032079	63.480354
30	LSTM	4	100	25	32	0.0042004	0.0036374	82.857993
31	LSTM	4	100	25	64	0.0048487	0.0034971	76.563163
32	LSTM	4	100	50	32	0.0036289	0.0056006	174.91538
33	LSTM	4	100	50	64	0.0036081	0.0034155	154.85546
34	LSTM	4	150	25	32	0.0053872	0.0057254	197.72541
35	LSTM	4	150	25	64	0.0050152	0.0042472	149.86526
36	LSTM	4	150	50	32	0.002715	0.0054168	383.53927
37	LSTM	4	150	50	64	0.0030951	0.0026457	308.29112

GRU:

- GRU models were faster than LSTM models on average, with comparable validation losses.
- A GRU with 4 layers, 100 units, and a batch size of 32 achieved the lowest validation loss of **0.00139834** in **144.58 seconds**, demonstrating strong performance in a relatively short time.
- However, higher batch sizes or epochs sometimes resulted in higher validation losses, likely due to overfitting, as seen in the 2-layer GRU with a batch size of 64, which had a validation loss of **0.014877738**.

38	GRU	2	50	25	32	0.0050191	0.0026215	22.396394
39	GRU	2	50	25	64	0.029919	0.0148777	14.644508
40	GRU	2	50	50	32	0.0060066	0.0064188	40.61634
41	GRU	2	50	50	64	0.0049709	0.0032881	25.767283
42	GRU	2	100	25	32	0.0036657	0.0033662	35.35908
43	GRU	2	100	25	64	0.005527	0.0038841	31.686033
44	GRU	2	100	50	32	0.0040963	0.0049679	70.59804
45	GRU	2	100	50	64	0.00344	0.0035172	58.61298
46	GRU	2	150	25	32	0.0090092	0.0103611	79.84501
47	GRU	2	150	25	64	0.0048439	0.0045321	48.415932
48	GRU	2	150	50	32	0.0044128	0.0051265	140.57738
49	GRU	2	150	50	64	0.002941	0.0034628	86.371465
50	GRU	3	50	25	32	0.0046728	0.0037188	32.355433
51	GRU	3	50	25	64	0.0288569	0.0138599	21.24226
52	GRU	3	50	50	32	0.0028296	0.0025428	60.2453
53	GRU	3	50	50	64	0.0040557	0.0027244	38.33153
54	GRU	3	100	25	32	0.0033058	0.0029022	56.35246
55	GRU	3	100	25	64	0.0050353	0.0030032	53.760123
56	GRU	3	100	50	32	0.0023831	0.0021726	104.83208
57	GRU	3	100	50	64	0.0029406	0.0025555	102.00537
58	GRU	3	150	25	32	0.003327	0.0024158	143.97243
59	GRU	3	150	25	64	0.0051418	0.0023905	76.443947
60	GRU	3	150	50	32	0.0029912	0.0041281	275.52085
61	GRU	3	150	50	64	0.003433	0.0033361	153.27882
62	GRU	4	50	25	32	0.0043571	0.0028997	47.816313
63	GRU	4	50	25	64	0.0284558	0.0078766	29.295199
64	GRU	4	50	50	32	0.0054451	0.0071932	83.547574
65	GRU	4	50	50	64	0.0043374	0.0040529	53.383323
66	GRU	4	100	25	32	0.0075962	0.0080135	75.283421
67	GRU	4	100	25	64	0.0058831	0.0032368	76.115868
68	GRU	4	100	50	32	0.0015761	0.0013983	144.58434
69	GRU	4	100	50	64	0.0030103	0.0030462	146.5243
70	GRU	4	150	25	32	0.0043969	0.00527	226.34778
71	GRU	4	150	25	64	0.0045284	0.0036731	123.6233
72	GRU	4	150	50	32	0.0022748	0.002335	443.99731
73	GRU	4	150	50	64	0.005446	0.0099647	265.3852

BiLSTM:

- BiLSTM models generally required longer training times, especially as the number of layers and units increased. For instance, a 4-layer BiLSTM with 150 units took over **3572.98 seconds** but did not significantly outperform simpler models.
- Despite the longer training times, BiLSTMs offered competitive performance, with a 3-layer BiLSTM (150 units, batch size 64) achieving a validation loss of **0.002521303** in **711.82 seconds**.
- Smaller models, such as a 2-layer BiLSTM with 50 units, offered a good balance between training time and performance, with validation losses as low as **0.003583482**.

74	BiLSTM	2	50	25	32	0.0062127	0.0084066	34.421674
75	BiLSTM	2	50	25	64	0.0047736	0.0047033	29.610417
76	BiLSTM	2	50	50	32	0.0036466	0.0035835	58.457826
77	BiLSTM	2	50	50	64	0.0031275	0.0056581	52.494458
78	BiLSTM	2	100	25	32	0.0032537	0.0032421	71.129849
79	BiLSTM	2	100	25	64	0.0033637	0.0028715	79.41703
80	BiLSTM	2	100	50	32	0.0024474	0.0027736	148.74011
81	BiLSTM	2	100	50	64	0.0026812	0.0034127	156.61709
82	BiLSTM	2	150	25	32	0.0027217	0.0027221	557.61771
83	BiLSTM	2	150	25	64	0.0042608	0.0038305	218.88155
84	BiLSTM	2	150	50	32	0.0028927	0.0030508	1061.4196
85	BiLSTM	2	150	50	64	0.0023224	0.0030248	447.06288
86	BiLSTM	3	50	25	32	0.004583	0.0038508	49.261417
87	BiLSTM	3	50	25	64	0.0060284	0.0064514	46.026208
88	BiLSTM	3	50	50	32	0.003569	0.0036462	90.590174
89	BiLSTM	3	50	50	64	0.00381	0.0030388	92.952708
90	BiLSTM	3	100	25	32	0.003316	0.0030609	127.66948
91	BiLSTM	3	100	25	64	0.004983	0.003816	166.176
92	BiLSTM	3	100	50	32	0.0047341	0.0050033	254.87537
93	BiLSTM	3	100	50	64	0.0030324	0.0042352	331.28497
94	BiLSTM	3	150	25	32	0.0041819	0.0053291	991.35329
95	BiLSTM	3	150	25	64	0.0045138	0.0034172	364.06905
96	BiLSTM	3	150	50	32	0.0027375	0.0038537	2048.0552
97	BiLSTM	3	150	50	64	0.0024565	0.0025213	711.81808
98	BiLSTM	4	50	25	32	0.0040693	0.0041456	144.44895
99	BiLSTM	4	50	25	64	0.0040866	0.0033045	102.15121
100	BiLSTM	4	50	50	32	0.002867	0.0032911	235.90679
101	BiLSTM	4	50	50	64	0.0024544	0.0027479	122.49983
102	BiLSTM	4	100	25	32	0.0029092	0.0030064	172.59446
103	BiLSTM	4	100	25	64	0.0044542	0.0034291	226.74597
104	BiLSTM	4	100	50	32	0.0025785	0.0031914	334.29476
105	BiLSTM	4	100	50	64	0.0026893	0.003486	954.73877
106	BiLSTM	4	150	25	32	0.0048745	0.0057513	1795.5204
107	BiLSTM	4	150	25	64	0.0032063	0.0024045	753.98807
108	BiLSTM	4	150	50	32	0.002941	0.0030249	3572.9864
109	BiLSTM	4	150	50	64	0.0046611	0.0065254	1445.9367

Insights:

LSTM Models:

- Generally performed well, with validation losses lower than training losses, indicating good generalization.
- Increasing the number of layers or units led to better performance but also significantly increased training time.

GRU Models:

- Faster than LSTM, with comparable performance. For example, a 4-layer GRU with 100 units achieved a validation loss of 0.0014 in 144.58 seconds, making it efficient and effective.
- GRU models sometimes exhibited overfitting with higher batch sizes or epochs.

BiLSTM Models:

- Required significantly longer training times, but the performance gain did not always justify the extra time.
- Smaller BiLSTM models struck a good balance between training time and performance.

4. Conclusion

In conclusion, the optimal model configuration depends on the balance between the accuracy required and the available computational resources. For tasks where time is a constraint, **GRU** models with a smaller number of units provide good performance in significantly less time. **LSTM** models offer robust performance, particularly for longer training epochs, while **BiLSTM** models may be more suitable for applications where higher model complexity is necessary, though at the cost of training time.

The implementation of the build_model function and subsequent experiments have provided valuable insights into constructing and tuning DL models for stock prediction. I outlined the basic key aspects of the implementation and results, providing a comprehensive overview of the process in this report.

5. Research and References

[Bidirectional LSTM]
(https://www.tensorflow.org/api_docs/python/tf/keras/layers/Bidirectional)

[Dropout]

(https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout)

[LSTM]

(https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM)

[GRU]

(https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRU)

[Model Compilation]

(https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile)

Deep Learning Book by Ian Goodfellow, Yoshua Bengio, and Aaron Courville: [Regularization for Deep Learning]

(https://www.deeplearningbook.org/contents/regularization.html)

TensorFlow Documentation: Model.evaluate to explain model evaluation.