Predicting Stock Market Prices with Long-Short Term Memory

By Franklin Liou, Kara Conrad, Kenneth J. Martinez, Aidan Pierre-Louis

University of Wisconsin-Madison

STAT 453

May 6th, 2024

1) Abstract

The stock market is noteworthy for being complex and near impossible to forecast. Deep learning practices have been used increasingly recently with new developments in RNNs to complete tasks with time series data. In the case of stock prediction, Long Short-Term Memory (LSTM)'s have been promising because of their 'long memory' feature. Our objective in this project was to accurately predict the daily closing prices of various Dow Jones companies' stocks. We examined ways in which to manipulate the input data, design the neural network architecture, and tweak hyperparameters such that they minimize loss and overfitting on the data throughout successive models. In this report we outline the design choices behind each model and the observed effect on their outputs as it concerns their ability to generalize and accurately predict the patterns of stock prices. While we found limited success in creating a single model that can generalize well across all companies, we designed a model which can generalize decently well with relatively small loss for a single company throughout our experimentation.

2) Introduction

Since the rise of artificial intelligence and deep learning, computer programs have been increasingly used to predict future events such as price movement in the stock market. Deep learning programs are able to recognize patterns in data to make estimations about the future which can be interpreted as general predictions. More specifically, RNNs or recurrent neural networks, are especially good at recognizing patterns in sequential data like stock market data. Modeling stock prices is a many-to-one RNN task, since the output data is one price based on the sequence of input data. However, since modeling stock price data requires recognizing long-term dependencies, using a RNN to model stock price data would result in a vanishing gradient issue, where the model's feature weights near zero. Therefore, it is necessary to use a LSTM model which uses a memory cell to be able to recognize long-term dependencies without running into

the vanishing gradient problem. The goal was not necessarily to outperform other related models, but to be able to successfully model stock price data using a LSTM neural network and to analyze how hyperparameter changes make the model more accurate. In the following sections we discuss some related experiments, outline the LSTM network we used for our two models, and analyze how successful each model was and why that was the case.

3) Related Work

Unsurprisingly, we are not the first to try to model stock market data using a LSTM network and we were inspired by multiple existing experiments and studies. Our first inspiration was mentioned in Lecture 14 where Bao et al. [1] establish that LSTM networks are more effective than RNNs for predicting times series with time steps of any size and any amount of steps. However, this study examines a model beyond a simple LSTM network, using wavelet transforms and stacked autoencoders in combination with a LSTM network with the goal of improving model performance over a simple LSTM network.

Another study that was more similar to ours was Yadav et al.'s [5] analysis of two different hyperparameters on the performance of a LSTM network on stock market data: whether the LSTM network maintains its state in between batches or not, and the optimal number of neural layers for an LSTM network. This study determined that whether or not the LSTM network maintains its state in between batches produced statistically insignificant differences, but noted that an LSTM network that does not maintain its state in between batches, also known as a stateless LSTM network, is more stable for longer time series prediction problems. Furthermore, the study determined that using just one neural layer was superior to using multiple neural layers in regards to minimizing loss based on root mean squared error, or RMSE. However, the study notes that using multiple neural layers may be more stable than using just one layer for complex problems since they observed a decrease in standard deviation with an increase in the number of neural layers.

Finally, our last inspiration for this project came from Youtuber Greg Hogg's video "Amazon Stock Forecasting in PyTorch with LSTM Neural Network (Time Series Forecasting)" [2]. This video was a simple demonstrative example of how one is able to use a LSTM neural network to model stock price data. It gave us insights into how to prepare our data for our own analysis of LSTM neural network performance on stock market data. Particularly, this video

introduced us to the lookback method of shifting data back a few days so our LSTM could use the data from the past couple days to predict a singular day. Overall, these related works helped us brainstorm ways on how we could approach modeling stock market data with an LSTM network, and helped us figure out which hyperparameters to tweak and which hyperparameters to leave alone.

4) Proposed Method

We decided on trying two different approaches to accurately predict stock market prices, but both approaches involved using a LSTM network. LSTM networks are able to revisit information but do so in a selective manner. It acts like a memory cell with three information controlling gates: the input gate, forget gate, and output gate. Here is a visual diagram of what one memory cell looks like in a LSTM network:

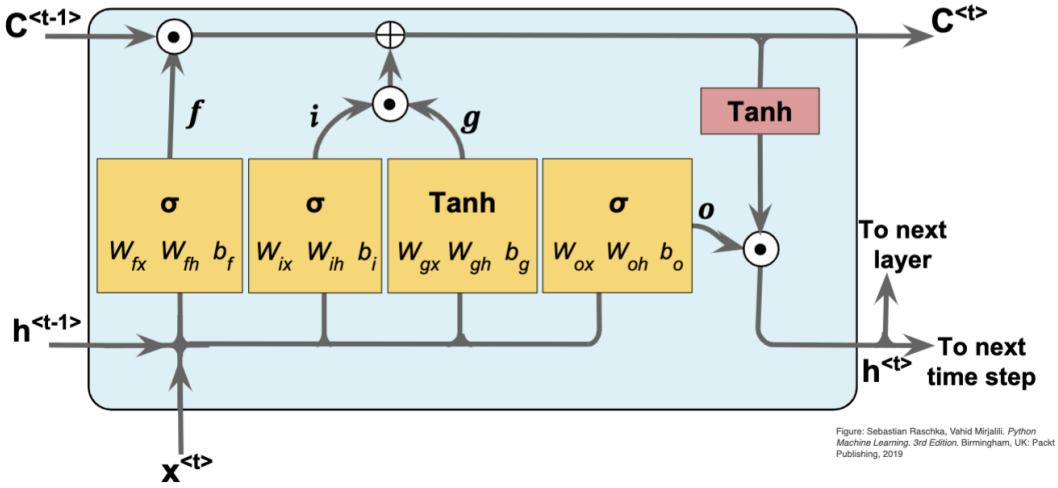


Figure: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Birmingham, UK: Packt Publishing, 2019

The input gate determines which new information should be stored in the memory cell. The forget gate determines which information from the previous cell state should be discarded or forgotten. The output gate determines the next hidden state based on the updated cell state, and determines what information to output to the rest of the network. The diagram and the following equations come from Lecture 14.

The first yellow box in the diagram represents the forget gate which is calculated with the equation:

$$f_t = \sigma\left(\mathbf{W}_{fx}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{fh}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_f\right)$$

The second yellow box represents the input gate which is calculated by:

$$\mathbf{i}_t = \sigma\left(\mathbf{W}_{ix}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{ih}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_i\right)$$

The input gate is multiplied by the input node which is represented by the third yellow box. The input node is calculated by:

$$\mathbf{g}_t = \tanh\left(\mathbf{W}_{gx}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{gh}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_g\right)$$

The multiplication of the input gate and input node updates the cell state. The fourth yellow box represents the output gate and is determined by the equation:

$$\mathbf{o}_t = \sigma\left(\mathbf{W}_{ox}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{oh}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_o\right)$$

Finally, the output gate, which is not exactly what is outputted to the next cell state. Instead, the hidden state that is passed on to the next cell state is calculated using the equation:

$$\mathbf{h}^{\langle t\rangle} = \mathbf{o}_t \odot \tanh\left(\mathbf{C}^{\langle t\rangle}\right)$$

Overall, the memory cell of a LSTM network determines how the model will learn, retain, and forget information efficiently, without running into the vanishing gradient problem that RNNs run into working with large, sequential datasets.

The practicality of LSTM in our sequenced data's backpropagation lies in its capacity to forget less informative inputs in each cell. While all the price data across the entire time time series informs our model of broader price fluctuation patterns, a priori knowledge suggests a stock's price is affected by that of more recent days and weeks more so than of the prices months and years before. Thus we desire some sort of *look-back* functionality in our model or input data features that further reinforces the importance of more recent features in the data. For programming the model using the PyTorch library without explicitly redefining much of the LSTM class[6] for our use, the more convenient option to implement look-back is through lagging price features in the input. Refer to Jimeng Shi et al. [3] for more details on how using various look-back methods improves forecasting accuracy.

5) Model 1

For our first model, we aimed to use as many features and as many companies as possible. In order to do this we would have to train our model on multiple companies at a time, including all of the features that were available to us in the yfinance python library. We used 43 big companies from across many industries, with data from April 5th, 2021 to April 5th, 2024

(756 open days). The features the library had were companies' daily Open, High, Low, Volume, Adj Close, Close values. We aimed to use Open, High, Low, Volume and Adj Close to predict companies' Closing prices.

5.1 Data Collection and Preparation

The data originally collected from yfinance was in the form of a 2 dimensional dataframe with dimensions 756 rows by 258 columns. The rows represented the days and the columns were grouped by features and then companies. We transformed this into a 3D pytorch tensor that was grouped by day, then features, then companies which is what is inputted into the model. Because each industry's stocks perform differently and likely range greatly, we decided to scale the features between [-1, 1]. This cleans up the data while still obtaining the effects any outliers would have [4]. We did this transformation to all models going forward. The dataset was split into a training set (90% of the data) and a testing set (10% of the data), ensuring that the model was evaluated on previously unseen data to test its generalization ability.

5.2 Model Training and Evaluation

For model training we used batch sizes of 10, 16 and 30. We also test 3 different epoch sizes, 100, 300 and 500. We measured the performance of each batch using MSE Loss and Stochastic Gradient Descent (SGD) as our optimizer with a learning rate of 0.001, 0.01 and 0.0001 during back propagation—set up shown in *Appendix B*. After training the model while alternating the hyperparameter, the combination that yielded the best average loss value were a batch size of 16, epoch size of 300 and a learning rate of 0.01. With these hyperparameters the training loss was 1062.11 units. This was alarming to see since this is an indicator that something was likely done wrong with our model. When validating our model using the testing data, the validation loss was 18,168.98 units, as seen in *Appendix B*.

5.3 Results and Discussion

To troubleshoot what may have gone wrong, we lowered the amount of companies that were imported to just those in the Dow Jones 30 list. When doing this and while alternating the same hyperparameter variations as previously listed, the training and testing error rates lowered to 917.12 and 15299.58 units respectively. We continued to eliminate companies and the fewer companies we included, the lower our error rates came out as. This highly suggests that our issue lied in the way our model was distinguishing data between different companies and was treating all the companies as one category versus different ones. We suspected this could have been

caused by the way we formatted the data, leading us to reformat the tensor to be ordered by day, company, then parameters. This configuration didn't make sense because the model's input size needs to be the number of companies we wanted to predict closing values for and when loading this data for training, the size of the third dimensions (parameters) is what the input size for our model needs to match with. With much confusion on what to do next, we pivoted to building a model easier to understand and use.

6) Model 2

For our second model, we used closing price as our only feature and adjusted the data with a lookback technique of seven days, giving each day's price more context by allowing the model to see the closing prices of the previous seven days. The implementation of Model 2 using Pytorch is in *Appendix C*.

6.1 Data Collection and Preparation

For the next model, it was decided to experiment with one company at a time instead of multiple companies as it makes it easier to interpret the results. In addition, only the close price feature for each day was analyzed. The dataset comprised closing stock prices from a subset of Dow Jones Industrial Average components, including "AAPL", "MSFT", "AMZN", and "GOOGL". The historical price data was obtained from Yahoo Finance using the yfinance library. The data spanned from April 5, 2018, to April 5, 2024. Each stock's data was processed to create a time series with lagged features, scaling the features to a range of [-1, 1] again to standardize the inputs to the network. The dataset was also split into a training set (90% of the data) and a testing set (10% of the data) as previously seen.

6.2 Training Model

Training involved feeding the prepared sequences into the model in batches, allowing for more efficient computation and improved gradient descent dynamics. Throughout the epochs, the model's parameters were optimized using the Adam optimizer, a variant of stochastic gradient descent that adapts learning rates based on the average of recent gradients for each parameter. The Root Mean Squared Error (RMSE) loss function was employed to quantify the discrepancy between the model's predictions and actual stock prices, driving the optimization process by backpropagation. Two distinct architectural variants of the LSTM model were employed—one without dropout regularization and another incorporating dropout layers—to assess their

influence on the model's ability to generalize and prevent overfitting. We adjusted the learning rate to values of 0.1, 0.01, and 0.001. This was carried out to determine the learning rate that best converges to a minimal loss without overshooting the minimum or converging too slowly, which are common problems with optimization algorithms. In our experiments, we adjusted the LSTM's hidden units to either 25 or 50. This was to understand how well the model can capture the intricacies of stock price movements—complex patterns might require more hidden units to learn effectively. With batch sizes, we looked at 16 and 32. The batch size affects learning dynamics; a smaller batch might help the model learn finer details but can be slow and noisy. A larger batch can lead to faster learning but might miss subtle patterns. We wanted to find a balance where the model can learn efficiently without sacrificing accuracy.

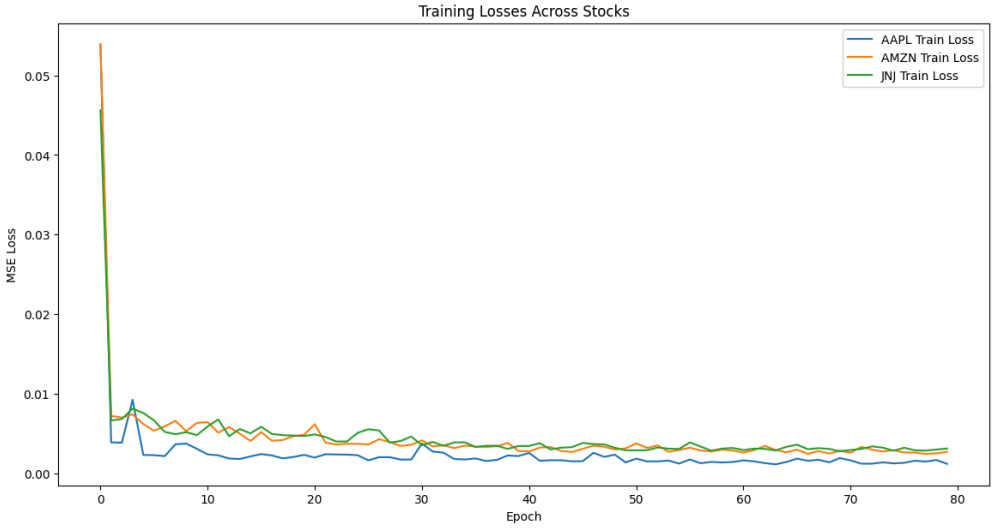
6.3 Model Evaluation

In order to ensure the reliability of our predictions, an evaluation step was implemented on unseen data that the trained model has not been exposed to. The criterion function that was used during evaluation was the RMSE, where the difference between the predicted and actual values was computed to produce a test loss. The test loss, along with the actual and predicted price graphs, helped determine the model's effectiveness in capturing the underlying trends and fluctuations in the stock prices. The graphs resulting from these tests are shown in *Appendix B*.

6.4 Results and Discussion

After training was implemented, the RMSE loss was plotted against the number of epochs, as shown in *Figure 1*. From the graph, we observe a sharp decline in MSE loss for all stocks at the onset of training, indicative of significant learning progress in the early stages and likely due to a small number of features, in this case, the close price only.

Figure 1. Training Losses Across Epochs for AAPL, AMZN, and JNJ Stocks.



In evaluating the performance of our LSTM models, *Table 1* provides a consolidated view of the twelve experiments conducted and their corresponding RMSE on test data for AAPL, AMZN, and JNJ stocks. The table highlights how variations in network depth, learning rates, and regularization impact the model's predictive accuracy.

Table 1. Summary of Experiments (Exp.) and Test RMSE Results

Exp.	**hidden size**	**batch size**	**learning rate**	**dropout rate**	**AAPL RMSE**	**AMZN RMSE**	**JNJ RMSE**
1	50	32	0.001	0.0	0.0558	0.0702	0.0476
2	50	16	0.001	0.0	0.0348	0.045	0.0409
3	25	16	0.001	0.0	0.0424	0.0483	0.0427
4	50	16	0.01	0.0	0.0481	0.0542	0.0499
5	50	16	0.1	0.0	0.9169	0.0799	0.1589
6	50	16	0.01	0.2	0.037	0.0465	0.0367
7	50	16	0.001	0.2	0.0395	0.0474	0.0378
8	50	16	0.001	0.3	0.0473	0.0463	0.0383
9	50	16	0.01	0.3	0.0449	0.0578	0.0392
10	50	32	0.001	0.3	0.0442	0.0464	0.0417
11	50	32	0.001	0.2	0.0353	0.0622	0.0408
12	25	16	0.001	0.2	0.0437	0.048	0.0393

6.5 Hidden Layer Size

The data obtained from our experimental iterations suggests a positive trend relating the size of the hidden layers to the model's test loss. A more pronounced decrease in test losses was observed for models with a higher number of hidden units. Specifically:

- *Experiment 7 vs. Experiment 12:* With all other hyperparameters held constant, the model configuration with 50 hidden units (Experiment 7) outperformed the configuration with 25 hidden units (Experiment 12) in terms of lower RMSE values for AAPL, AMZN, and JNJ.

- *Experiment 2 vs. Experiment 3:* A similar improvement was seen when comparing Experiment 2 (50 hidden units) to Experiment 3 (25 hidden units), with the former demonstrating superior performance in reducing test losses for all three companies.

These results imply that an increased number of hidden units can enhance the LSTM network's ability to discern and learn from the complex temporal patterns present in stock price data.

6.6 Batch Size

Upon analyzing the experimental data, a consistent trend emerges suggesting that a batch size of 16 yields lower test losses across the three companies examined compared to a batch size of 32, indicating improved model performance. This observation holds true when directly contrasting experiments where only the batch size variable was altered:

- *Experiment 2 vs. Experiment 1:* With all variables held constant except for the batch size, Experiment 2 (batch size 16) outperformed Experiment 1 (batch size 32) in terms of lower RMSE values for all companies.
- *Experiment 7 vs. Experiment 11:* Similarly, Experiment 7 (batch size 16) exhibited superior performance compared to Experiment 11 (batch size 32) for AMZN and JNJ, while AAPL showed a slight increase in RMSE.
- *Experiment 8 vs. Experiment 10:* Experiment 8 (batch size 16) demonstrated better outcomes than Experiment 10 (batch size 32) for AMZN and JNJ, paralleling the pattern observed in previous comparisons.

These results suggest that smaller batch sizes tend to enhance the learning capability of the model. This may be attributed to the fact that smaller batches offer a more frequent update of weights, potentially allowing the model to learn finer nuances in the data.

6.7 Learning Rate

The influence of learning rate was examined but it resulted in mixed outcomes. Without the implementation of dropout, a learning rate of 0.001 in Experiment 2 exhibited the lowest test loss. However, inconsistent outcomes across companies resulted when regularization was implemented. For a dropout of 20% of the nodes, a learning rate of 0.01 was the most adequate but for a 0.3 dropout rate, test losses varied across companies. This inconclusive outcome could be due to each companies' stock data that may make the algorithm oscillate around optimal solutions without converging. Thus, a careful tuning of the learning rate is essential for optimizing the model performance.

6.8 Dropout Rate

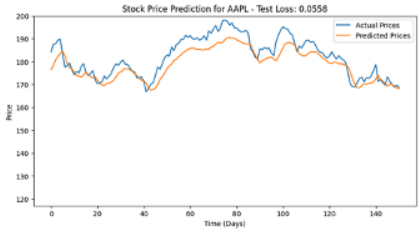
Across the experiments, the effectiveness of dropout as a regularization technique is evident. Specifically, experiments 1, 11, and 10—which varied dropout rates from 0, 0.2, to 0.3 respectively—demonstrate a clear trend where a dropout rate of 0.2 yielded the most favorable results, followed by 0.3, with the no-dropout setup performing the least effectively. This pattern was consistently observed in another set of experiments (4, 6, and 9) under different conditions but with the same variations in dropout rates. The recurrent observation that a moderate dropout rate of 0.2 optimizes performance suggests that it strikes a beneficial balance between sufficient regularization to prevent overfitting and maintaining enough capacity to learn important features in the data.

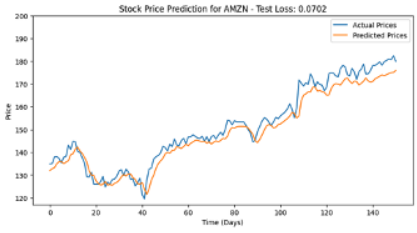
7) Conclusion

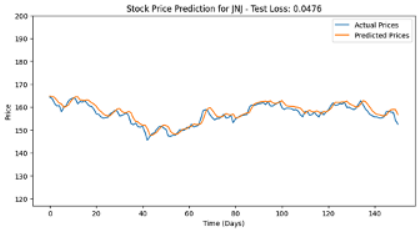
Within the scope of our original goal to create a single model which can predict stock closing prices for any company in the Dow Jones, we designed our first model and formatted our input data to do exactly this. Though the initial model was able to run and produce results, the scale of the loss produced was less than desirable. Improvements upon this model proved to be challenging due to confusion surrounding how the model interpreted the data owing to the number of features and dimensionality of the input. By adjusting our design to train models using only closing price as a price feature, we saw drastic decreases in loss. Furthermore, we saw even greater improvements by training unique models by each individual company. The implementation of the look-back technique and tweaking of hyperparameters as previously elaborated even further aided in this reduction of loss. By using dropout regularization, the generalization capabilities of our later models displayed impressive results in reducing overfitting so as to reduce their tendency to learn the noise patterns in a company's training data. In adjusting the scope of our goal as mentioned, we created a means of training models with satisfactory small loss and high generalization for predicting stock prices.

Appendix A: Plots for each experiment of Model 2 for each of the three companies analyzed.

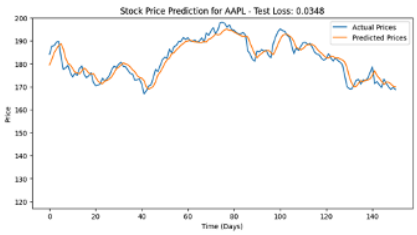
Experiment 1

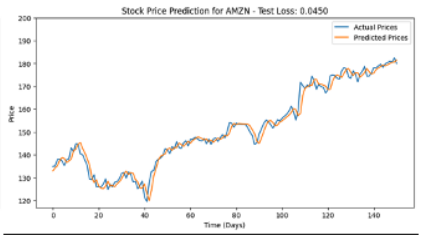


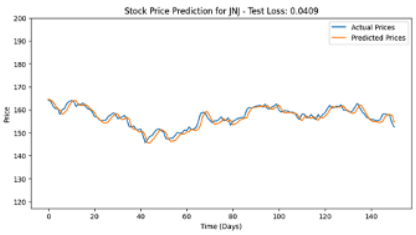




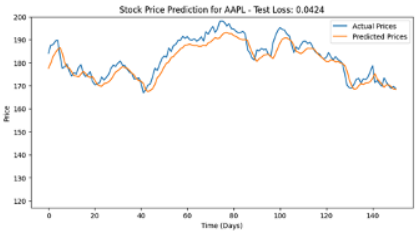
Experiment 2

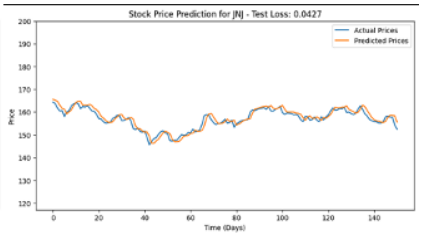




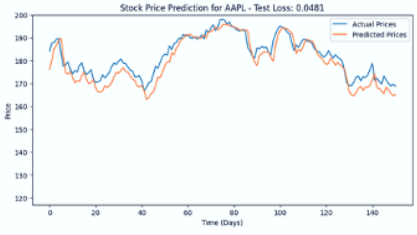


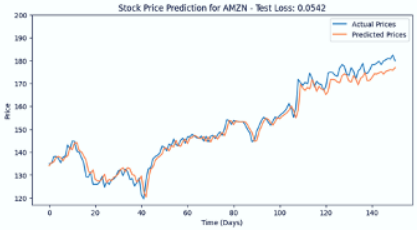
Experiment 3



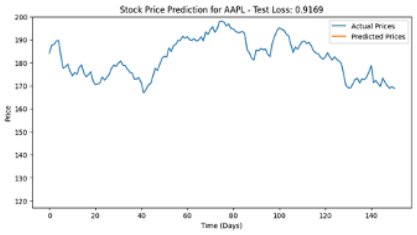


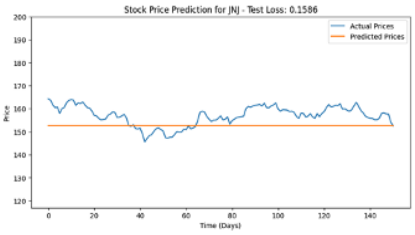
Experiment 4



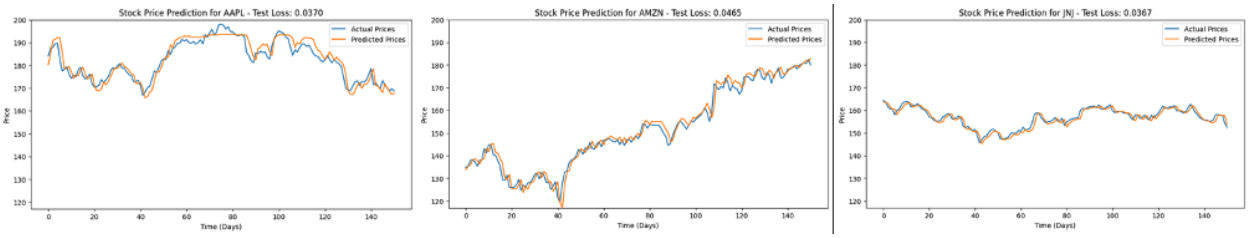


Experiment 5

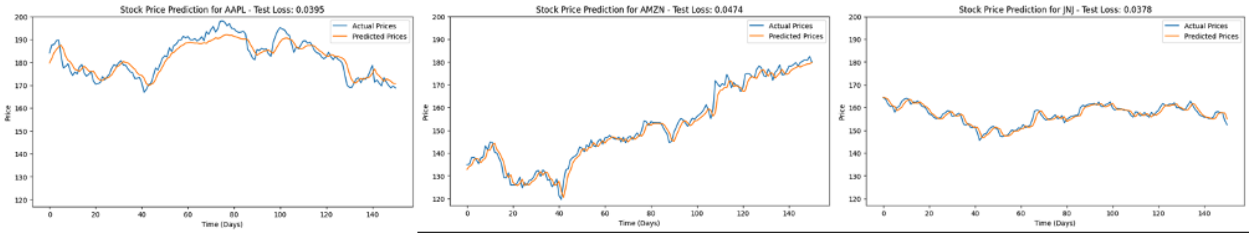




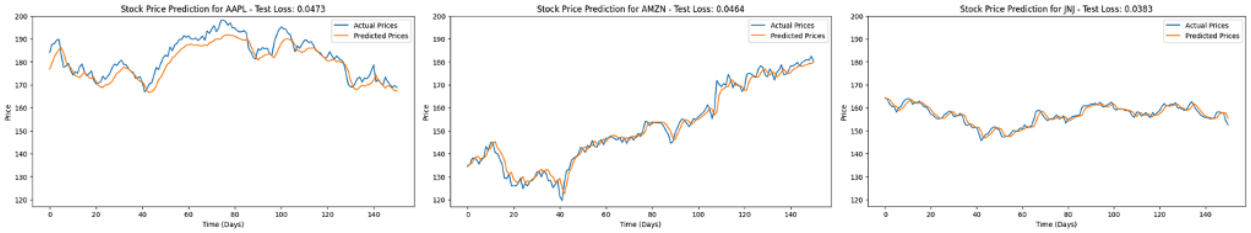
Experiment 6



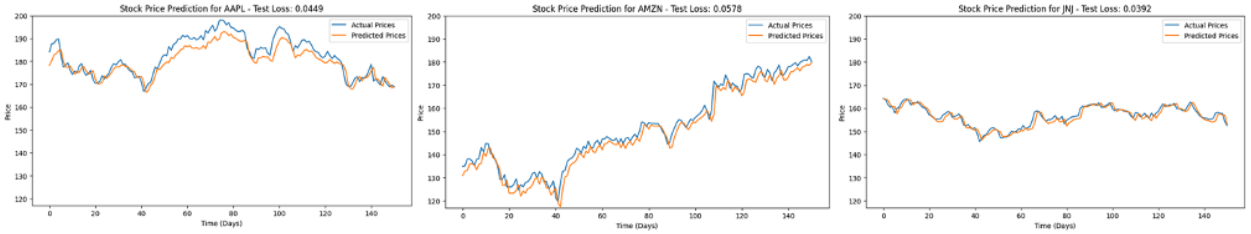
Experiment 7



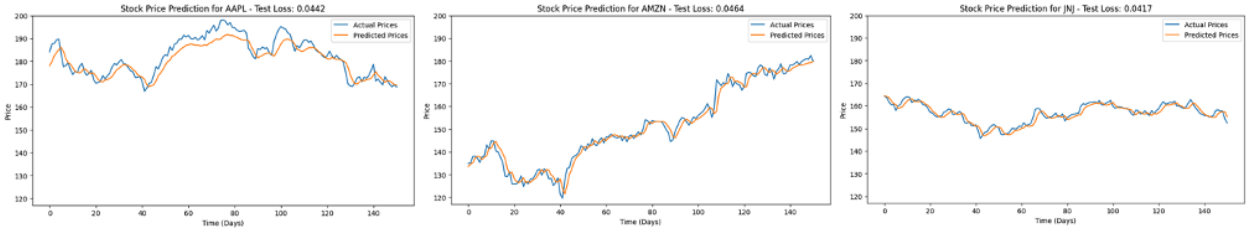
Experiment 8



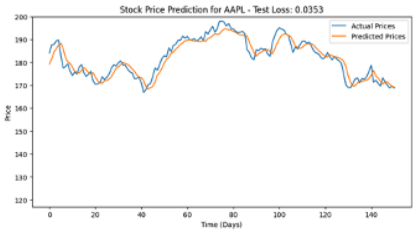
Experiment 9

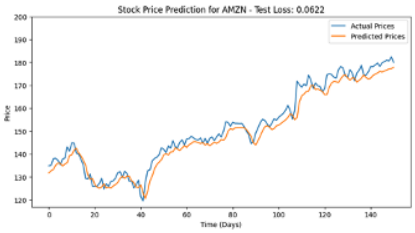


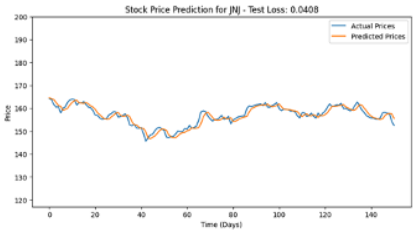
Experiment 10



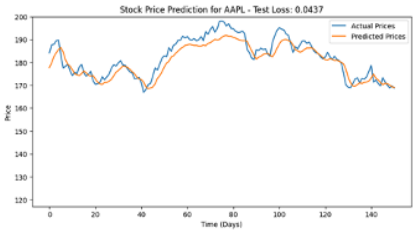
Experiment 11

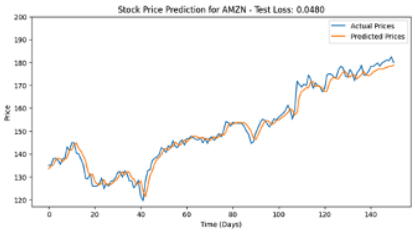


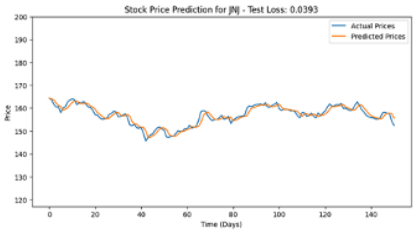




Experiment 12







Appendix B: Code for Model 1

```
# make model
class LSTM(torch.nn.Module):
 def __init__(self, input_size, hidden_size, stacked_size):
   super().__init__()
   self.hidden_size = hidden_size
   self.stacked_size = stacked_size
      self.lstm = torch.nn.LSTM(input_size, hidden_size, stacked_size,
batch_first=True)
     self.fc = torch.nn.Linear(hidden_size, 1) # fc is fully connected
layer, the 1 for predicting final/closing value

 def forward(self, x):
   batch_size = x.size(0)
   h0 = torch.zeros(self.stacked_size, batch_size, self.hidden_size)
   c0 = torch.zeros(self.stacked_size, batch_size, self.hidden_size)
   out, (hn, cn) = self.lstm(x, (h0, c0))
   out = self.fc(out[:, -1, :])
   return out

model = LSTM(43, 30, 6)
```

```
# For training model
def train_epoch():
 model.train(True)
```

```
 print(f'Epoch: {epoch + 1}')
 running_loss = 0.0

 for batch_index, batch in enumerate(train_loader):
       x_batch, y_batch = batch[0], batch[1]

       output = model(x_batch)
       loss = loss_function(output, y_batch)
       running_loss += loss.item()

       optimizer.zero_grad()
       loss.backward()
       optimizer.step()

       if (batch_index+1) %10 == 0:  # print every 100 batches
         avg_loss_across_batches = running_loss / 100
         print('Batch {0}, Loss: {1:.3f}'.format(batch_index+1,

avg_loss_across_batches))
         running_loss = 0.0
```

```
#for testing model
def validate_epoch():
   model.train(False)
   running_loss = 0.0

   for batch_index, batch in enumerate(test_loader):
       x_batch, y_batch = batch[0], batch[1]

       with torch.no_grad():
           output = model(x_batch)
           loss = loss_function(output, y_batch)
           running_loss += loss.item()

   avg_loss_across_batches = running_loss / len(test_loader)

   print('Val Loss: {0:.3f}'.format(avg_loss_across_batches))

loss_function  = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001) #could use Adam
```

```
epoch_size = 300

for epoch in range(epoch_size):
 train_epoch()
 validate_epoch()
```

Appendix C: Code for Model 2

```
def split_data(data, train_ratio=0.9):
   num_train = int(len(data) * train_ratio)
   return data[:num_train], data[num_train:]

train_data = {}
test_data = {}

for stock, data in all_data_scaled.items():
   train, test = split_data(data)
   train_data[stock] = train
   test_data[stock] = test
```

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
import numpy as np
input_size = 1
hidden_size = 50
num_layers = 3
epochs = 80
batch_size = 8
learning_rate = 0.001
dropout_rate = 0.2
class StockLSTM(nn.Module):
   def __init__(self, input_size, hidden_size, num_layers, dropout_rate):
       super(StockLSTM, self).__init__()
       self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True, dropout=dropout_rate if num_layers > 1 else 0.0)
```

```
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        out, _ = self.lstm(x)
        out = self.fc(out[:, -1, :])  # Take output of the last time step
        return out

def train_and_validate(data, model, optimizer, criterion, epochs,
batch_size):
    train_losses = []
    for epoch in range(epochs):
        model.train()
        dataset = TensorDataset(torch.tensor(data[:, 1:]).float()
.unsqueeze(-1), torch.tensor(data[:, 0]).float().unsqueeze(1))
        loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
        epoch_loss = 0
        for inputs, targets in loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item() * inputs.size(0)
        epoch_loss /= len(loader.dataset)
        train_losses.append(epoch_loss)
    return train_losses
# Plot training losses
plt.figure(figsize=(14, 7))
for stock, losses in all_losses.items():
    plt.plot(losses, label=f'{stock} Train Loss')
plt.title('Training Losses Across Stocks')
plt.xlabel('Epoch')
plt.ylabel('RMSE Loss')
plt.legend()
plt.show()
# Function to evaluate test data
```

```
def evaluate_test_data(test_data, model, criterion, scaler):
    model.eval()
    with torch.no_grad():
        inputs = torch.tensor(test_data[:, 1:]).float().unsqueeze(-1)
        targets = torch.tensor(test_data[:, 0]).float().unsqueeze(1)
        outputs = model(inputs)
        test_loss = criterion(outputs, targets).item()

        predictions = outputs.view(-1).numpy()
        actuals = targets.view(-1).numpy()

        # Inverse transform predictions and actuals
        dummy_features = np.zeros((len(predictions), 7))  # 7 lag features
        predictions_full = np.column_stack((predictions, dummy_features))
        actuals_full = np.column_stack((actuals, dummy_features))

        predictions_inversed = scaler.inverse_transform(predictions_full)
[:, 0]
        actuals_inversed = scaler.inverse_transform(actuals_full)[:, 0]

        return predictions_inversed, actuals_inversed, test_loss
def RMSE(outputs, targets):
    mse_loss = nn.MSELoss()
    mse = mse_loss(outputs, targets)
    rmse = torch.sqrt(mse)
    return rmse
# Training models and evaluating each stock
for stock in dowjones_comps:
    model = StockLSTM(input_size, hidden_size, num_layers, dropout_rate)
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    criterion = RMSE
    train_and_validate(train_data[stock], model, optimizer, criterion,
epochs, batch_size)
    predictions, actuals, test_loss = evaluate_test_data(test_data[stock],
model, criterion, scalers[stock])
```

```
    # Plotting predictions vs actuals with test loss in title
    plt.figure(figsize=(10, 5))
    plt.plot(actuals, label='Actual Prices')
    plt.plot(predictions, label='Predicted Prices')
    plt.title(f'Stock Price Prediction for {stock} - Test Loss:
{test_loss:.4f}')
    plt.xlabel('Time (Days)')
    plt.ylabel('Price')
    plt.ylim(117, 200)
    plt.legend()
    plt.show()

plt.figure(figsize=(14, 7))
for stock, losses in all_losses.items():
    plt.plot(losses, label=f'{stock} Train Loss')
plt.title('Training Losses Across Stocks')
plt.xlabel('Epoch')
plt.ylabel('RMSE Loss')
plt.legend()
plt.show()
```

References

[1] Bao, W., Yue, J., & Rao, Y. (2017). A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PLOS ONE*, *12*(7). https://doi.org/10.1371/journal.pone.0180944

[2] Hogg, Greg. (2023, April 8). *Amazon Stock Forecasting in PyTorch with LSTM Neural Network (Time Series Forecasting) | Tutorial 3* [Video]. YouTube. https://www.youtube.com/watch?v=q_HS4s1L8UI

[3] Shi, J., Jain, M., & Narasimhan, G. (2022, April 3). Time Series Forecasting (TSF) Using Various Deep Learning Models. *arXiv preprint arXiv:2204.11115*.

[4] Pedregosa et al., (2011), JMLR 12, pp. 2825-2830, https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html

[5] Yadav, A., Jha, C. K., & Sharan, A. (2020). Optimizing LSTM for time series prediction in Indian Stock Market. *Procedia Computer Science*, *167*, pp. 2091–2100. https://doi.org/10.1016/j.procs.2020.03.257

[6] PyTorch Contributors, (2023). *LSTM*. LSTM - PyTorch 2.3 documentation. https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html