Austin Vornhagen

12/3/2021

CSCI 8110

Semester Project

Introduction:

The topic of this report is a univariate LSTM deep learning model using time-series data from the stock market to try to predict price movements in the future. The problem to be solved is understanding an LSTM model and using a univariate LSTM model to predict the price direction of a financial instrument in order to grow a brokerage account. This paper will explore different tweaks to the LSTM deep learning model. If the solution is successful, this could be applied to the live markets to forecast the price level that has the highest probability of showing up in the future. Someone with a successful model could use this to aid their investment strategy or trading strategy giving them an "edge in the market".

Related work:

The related work that was reviewed for this paper is made up of two parts: Long Short-Term Memory background and stock prediction using LSTM.

Part 1: Long Short-Term Memory Background

Long short-term memory is an artificial recurrent neural network architecture. The most cited paper around when it was first introduced was:

- Hochreiter, Sepp & Schmidhuber, Jürgen. (1997). Long Short-term Memory. Neural computation. 9. 1735-80. 10.1162/neco.1997.9.8.1735.

The advantages of the LSTM are it has a feedback connection and can process sequences of data. An LSTM layer has a cell, an input gate, an output gate, and a forget gate. This was developed to help mitigate the vanishing gradient problem. The vanishing gradient problem happens when training artificial neural networks with optimizers like stochastic gradient descent. The gradient would then become so small that it wouldn't change, preventing the network from further training. The current understanding is for a RNN there is:

$$\frac{\partial h_{t'}}{\partial h_t} = \prod_{k=1}^{t'-t} \omega\sigma'(\omega h_{t'-k})$$

For the LSTM there is:

$$\frac{\partial s_{t'}}{\partial s_t} = \prod_{k=1}^{t'-t} \sigma(v_{t+k})$$

The product sums of both have a sigmoid term that when multiplied together $t' - t$ times can vanish. For RNN the gradient decays with $\omega\sigma'$ while the LSTM gradient decays with σ.

LSTM networks are also capable of holding long term dependencies. This is a result of the gates controlling what information is let through the cell. The first step is the forget gate layer, which is a sigmoid layer that decides what information to throw away by outputting a number between 0 and 1. A value of 0 means throw away everything, a value of 1 means keep everything. The second step is to decide what to store in the cell. This is done by the input gate which has a sigmoid layer and a tanh layer. The sigmoid layer decides which values to update and the tanh layer creates a vector of new candidate values that could be added to the state. Then the old cell state is updated to a new cell state. The output gate is made up of a sigmoid layer that decides what parts of the cell state we are going to output, then a tanh layer puts the cell state values between -1 and 1. We multiply the tanh output by the output of the sigmoid gate, so the LSTM only outputs the parts of the cell state we want.

There have been variations made to the LSTM. In 2000 there was a paper about an LSTM variant using "peephole connections" from its internal cells to its multiplicative gates to learn the distinction between sequences of spikes separated by either 50 or 49 discrete time steps without help of short training exemplars.

- F. A. Gers and J. Schmidhuber, "Recurrent nets that time and count," Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium, 2000, pp. 189-194 vol.3, doi: 10.1109/IJCNN.2000.861302.

In 2014 paper, a variation of the LSTM was the Gated Recurrent Unit or GRU. It combines the forget and input gates into a single update gate while also merging the cell state and hidden state.

- Cho, Kyunghyun & Merrienboer, Bart & Gulcehre, Caglar & Bougares, Fethi & Schwenk, Holger & Bengio, Y.. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. 10.3115/v1/D14-1179.

In a 2015 paper, a variation of the LSTM was the Depth Gated Recurrent Neural Network. It is presented as an extension of LSTM to use a depth gate to connect memory cells of adjacent layers. This introduces a linear dependence between lower and upper recurrent units.

- Yao, Kaisheng & Cohn, Trevor & Vylomova, Katerina & Duh, Kevin & Dyer, Chris. (2015). Depth-Gated Recurrent Neural Networks.

LSTM networks perform well at classifying, processing, and making predictions on timeseries data.

Part 2: Stock Prediction using LSTM

There exists a lot of literature on building portfolios using predictive models. These papers can be divided into three categories. The first category is doing stock prediction with regression, like ordinary least square regression and variants of OLS like penalty-based regression, and polynomial regression. These can be found here:

- J. Sen and T Datta Chaudhuri, "An alternative framework for time series decomposition and forecasting and its relevance for portfolio choice: A comparative study of the Indian consumer durable and small cap sectors", Jour. of Eco. Lib., vol. 3, no. 2, pp. 303-326, 2016.
- X. Zhong and D. Enke, "Forecasting daily stock market return using dimensionality reduction", Expert System with Application, vol. 97, pp. 60-69, 2017.

- S. S. Roy, D. Mittal, A. Basu, and A. Abraham, "Stock market-forecasting using LASSO linear regression model", Proc. of Afro-European Conf. of Ind. Advancements, pp. 371-381, 2015.

The second category uses econometric methods like autoregressive integrated moving average (ARIMA), cointegration, and quartile regression. These can be found here:

- Y. Ning, L. C. Wah, and L. Erda, "Stock price prediction based on error correction model and Granger causality test", Cluster Computing, vol. 22, pp. 4849-4858, 2019.
- L. Wang, F. Ma, J. Liu, and L. Yang, "Forecasting stock price volatility: New evidence from the GARCH-MIDAS model", Int. Journal of Forecasting, vol. 36, no. 2, pp. 684-694, 2020.
- Y. Du, "Application and analysis of forecasting stock price index based on combination of ARIMA model and BP neural network", Proc. of CCDC, pp. 2854-2857, Jun 9-10, Shenyang, China, 2018.

The last category is using machine learning, deep learning, reinforcement learning, and hybrids of each. These can be found here:

- S. Mehtab and J. Sen, "Stock price prediction using CNN and LSTM-based deep learning models", Proc. of Int. Conf. on Decision Aid Sc. and Appl. (DASA), pp. 447-453, Nov 8-9, 2020, Bahrain.
- S. Mehtab, J. Sen and A.Dutta, "Stock price prediction using machine learning and LSTM-based deep learning models", In: Thampi, S. M. et al. (eds.) Machine Learning and Metaheuristics Algorithms and Applications (SoMMA'20), pp. 88-106, vol 1386, Springer, Singapore.
- B. Yang, Z-J. Gong, and W. Yang, "Stock market index prediction using deep neural network ensemble", Proc. of IEEE CCC, pp. 3382-3887, Jul 26-28, Dalian, Chian, 2017.
- S. Mehtab and J. Sen, "A robust predictive model for stock price prediction using deep learning and natural language processing", Proc. of ICBAI, Dec 5-7, Bangalore, India, 2019.
- K. Nam and N. Seong, "Financial news-based stock movement prediction using causality analysis of influence in the Korean stock market", Decision Support Systems, vol. 117, pp. 100-112, 2019.

A major difficulty that is encountered when working with stock market time-series data is finding a model that will fit the data because of the comprehensive set of factors that influence a stock's price. Without the ability to take every possible factor into account, the data appears to be full of randomness and is very volatile. If a model were to really work properly, the model should be able to accurately predict the future price of a stock in order to be used by the investor to enter and exit the market profitably. This paper's selected method will be to explore the future price predictions using an LSTM model and decide if they are accurate enough to be used by an investor.

Methodology:

The methodology section describes the static pieces of the implementation. This does not include any of the details on results of experiments given by the models. Details on the experiments will be expanded on in the next section.

The goal of this is to explore different tweaks and variations to a univariate LSTM model to understand a univariate LSTM model and optimize it to make stock market predictions accurately. All the experiments were performed in Google Collaboratory. The implementation was written in the python language using the pandas, numpy, keras, tensorflow, and plotly libraries. The methodology can be broken into 12 steps:

1. Pull and Store Data Manually
2. Read Data, Remove Unused Columns, and Plot Data
3. Preprocess Data
4. Set Lookback and Format Data
5. Create Model
6. Compile Model
7. Fit Model
8. Perform Predictions
9. Reshape Data for Plotting
10. Plot Train Data, Test Data, and Prediction Data
11. Forecast Future Data
12. Plot Forecast

Step 1: Pull and Store Data Manually

The script used stock market data from the website tradingview.com. This data comes in Microsoft Excel Comma Separated Values File file type. This is candlestick chart data on the daily timeframe. It comes through with 5 columns: time (Unix timestamp), open (float64), high (float64), low (float64), and close (float64). The data was manually adjusted to remove any null records if present, and added a new column that converts the time (Unix timestamp) column to a simple date column named "date". Date is a more readable and user-friendly version of the sequence data and helps with understanding the plots later on. This data was then saved into the author's personal Google Drive account so that the implementation in Google Collaboratory could easily access the data.

The script used 5 different stocks as datasets: AMD, CRSR, TSLA, DFS, and PFE.

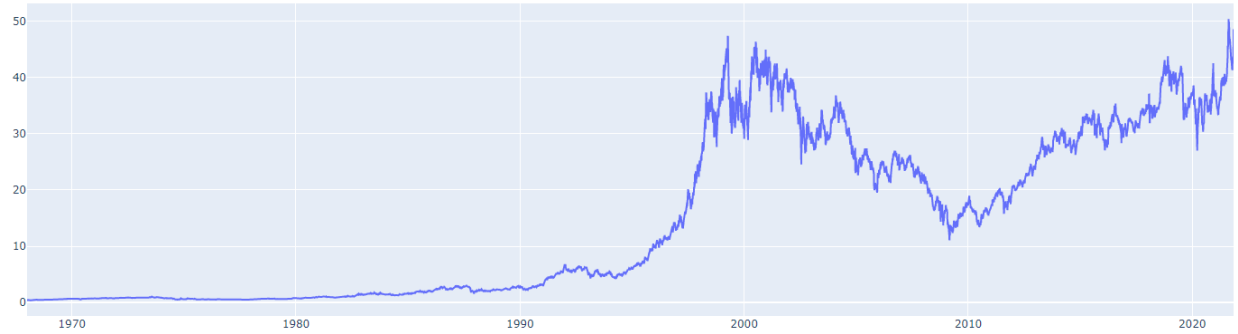
Step 2: Read Data, Remove Unused Columns, and Plot Data

The script used pandas read_csv( ) function to read the file. The stock market data file path was passed from Google Drive to this function. It returns a DataFrame which is a 2-dimensional labeled data structure with columns of potentially different types. The DataFrame information was then printed using the pandas info( ) function. This function has no input arguments and returns a summary of the DataFrame. It includes the number of rows, number of columns, column index, column name, amount of non-null rows for each column, datatypes of each column, and memory usage of the file. Below is an example:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13556 entries, 0 to 13555
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   time    13556 non-null  int64
 1   open    13556 non-null  float64
 2   high    13556 non-null  float64
 3   low     13556 non-null  float64
 4   close   13556 non-null  float64
 5   date    13556 non-null  object
dtypes: float64(4), int64(1), object(1)
memory usage: 635.6+ KB
None
```

The date column is then set in the DataFrame to datetime using the pandas to_datetime( ) function. This function takes the DataFrame[‘column name’] as input. The script then drops the time, open, high, and low columns since the close column and date column are all we need for a univariate LSTM model. This is done with a pandas drop function. This function takes input arguments of the columns to drop in a list, and inplace = True. The inplace argument set to True means perform the operation inplace and return None.

The data is then plotted using the Plotly library Figure() function. This function takes a Scatter() function as input. The Scatter() function takes the date column from the DataFrame as input for the x-axis and the close column from the DataFrame as input for the y-axis. This returns a scatter plot of the data. This scatter plot is then passed to the Figure() function which returns a figure object. The figure object is then plotted with the show() function. Below is an example:



Step 3: Preprocess Data

To preprocess the data the numpy library reshape() function is used to reshape the close column values in the DataFrame from a 1-D array with n elements, to a 2-D array with n arrays each with 1 element. This is done by passing in (-1,1) to the reshape function. Below is an example:

1-D

```
[ 0.429534  0.438586  0.446814 ... 44.82     43.85     48.61    ]
```

2-D

```
[[ 0.429534]
 [ 0.438586]
 [ 0.446814]
 ...
 [44.82    ]
 [43.85    ]
 [48.61    ]]
```

The data is then split so that the first 80% of the data is used to train and the last 20% of the data is used to test. This is done by finding the index of the row that is closest to being 80% of the length of the data. Then new variables are set for the close column and the date column in the DataFrame so that each has training and testing data.

Step 4: Set Lookback and Format Data

The lookback variable is an integer that represents how many periods the model will lookback and use when making a one period prediction.

Next, the timeseries data preprocessing function TimeseriesGenerator() from keras is used to create 2 datasets of sliding windows over the timeseries array provided. This is an old function that was replaced by timeseries_dataset_from_array () but the new function was not formatting the data as expected. There was a sliding window made for training data and another for testing data using the close column values. As input, this function takes a data argument that is the training data for the close column, a targets argument that is also the training data for the close column because the model is hoping to predict this same data, a length argument that specifies how many elements each window should contain, and a batch size argument for how many sliding windows and targets are processed at a time.

Step 5: Create Model

There were 5 different models created so that different LSTM approaches could be compared. All of the models are tf.keras.Model models that use the tf.keras.Sequential() class. This means the models are made of up linear stacks of layers and have training and inference features. Every model has at minimum 1 LSTM layer.

Model 1 – LSTM + Dense:

```
Model: "Model 1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 lstm_15 (LSTM)               (None, 10)                480

 dense_7 (Dense)              (None, 1)                 11

=================================================================
Total params: 491
Trainable params: 491
Non-trainable params: 0
_________________________________________________________________
```

Model 1 is made up of 2 layers. The first layer is an LSTM layer that has 10 nodes, uses ReLU activation, and takes a 2-D input shape that is (lookback, 1). The second layer is a Dense layer that is for the output of the model and has 1 node.

Model 2 – 2 LSTM + 2 Dropout + Dense:

```
Model: "Model 2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 lstm_16 (LSTM)               (None, 10, 256)           264192

 dropout_2 (Dropout)          (None, 10, 256)           0

 lstm_17 (LSTM)               (None, 256)               525312

 dropout_3 (Dropout)          (None, 256)               0

 dense_8 (Dense)              (None, 1)                 257

=================================================================
Total params: 789,761
Trainable params: 789,761
Non-trainable params: 0
_________________________________________________________________
```

Model 2 is made up of 5 layers. The first layer is an LSTM layer with 256 nodes, a return_sequences argument set to True, ReLU activation, and input shape like (lookback, 1). The second layer is a Dropout layer with a dropout rate of 0.3. The third layer is an LSTM layer with 256 nodes. The fourth layer is a Dropout layer with a dropout rate of 0.3. The last layer is a Dense layer that is for the output of the model and has 1 node.

Model 3 – LSTM + 3 Dense:

```
Model: "Model 3"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 lstm_18 (LSTM)               (None, 10)                480

 dense_9 (Dense)              (None, 200)               2200

 dense_10 (Dense)             (None, 100)               20100

 dense_11 (Dense)             (None, 1)                 101

=================================================================
Total params: 22,881
Trainable params: 22,881
Non-trainable params: 0
_________________________________________________________________
```

Model 3 is made up of 4 layers. The first layer is an LSTM layer with 10 nodes, ReLU activation, and input shape like (lookback, 1). The second layer is a Dense layer with 200 nodes. The third layer is a Dense layer with 100 nodes. The fourth layer is a Dense layer that is for the output of the model and has 1 node.

Model 4 – LSTM Autoencoder:

```
Model: "Model 4"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 lstm_19 (LSTM)               (None, 10, 128)           66560

 lstm_20 (LSTM)               (None, 64)                49408

 repeat_vector_3 (RepeatVect  (None, 10, 64)           0
 or)

 lstm_21 (LSTM)               (None, 10, 64)            33024

 lstm_22 (LSTM)               (None, 10, 128)           98816

 time_distributed_2 (TimeDis  (None, 10, 1)            129
 tributed)

=================================================================
Total params: 247,937
Trainable params: 247,937
Non-trainable params: 0
_________________________________________________________________
```

Model 4 is made up of 6 layers. The first layer is an LSTM layer with 128 nodes, a return_sequences argument set to True, ReLU activation, and input shape of (lookback,1). The second layer is an LSTM layer with 64 nodes, a return_sequences argument set to False, and ReLU activation. The third layer is a RepeatVector layer that takes the lookback as input. The fourth layer is an LSTM layer with 64 nodes, a return_sequences argument set to True, and ReLU activation. The fifth layer is an LSTM layer with 128 nodes, return_sequences argument set to True, and ReLU activation. The last layer is a TimeDistributed Dense layer with 1 node for the output of the model.

Model 5 – 2 LSTM + 2 Dropout + RepeatVector + TimeDistributed:

```
Model: "Model 5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_23 (LSTM)              (None, 128)               66560

 dropout_4 (Dropout)         (None, 128)               0

 repeat_vector_4 (RepeatVect  (None, 10, 128)          0
 or)

 lstm_24 (LSTM)              (None, 10, 128)           131584

 dropout_5 (Dropout)         (None, 10, 128)           0

 time_distributed_3 (TimeDis  (None, 10, 1)            129
 tributed)

=================================================================
Total params: 198,273
Trainable params: 198,273
Non-trainable params: 0
_________________________________________________________________
```

Model 5 is made up of 6 layers. The first layer is an LSTM layer with 128 nodes, ReLU activation, and input shape of (lookback,1). The second layer is a Dropout layer with a dropout rate of 0.2. The third layer is a RepeatVector layer with input of lookback. The fourth layer is an LSTM layer with 128 nodes, ReLU activation, and return_sequences argument set to True. The fifth layer is a Dropout layer with a dropout rate of 0.2. The last layer is a TimeDistributed Dense layer with 1 node for the output of the model.

After the model is created a summary() function is used to see a summary of the model.

Step 6: Compile Model

The model is compiled with the keras library compile() function. This function takes as input an optimizer argument set to adam and a loss argument set to mse. Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

- Kingma, Diederik & Ba, Jimmy. (2014). Adam: A Method for Stochastic Optimization. International Conference on Learning Representations.

MSE stands for Mean Squared Error which computes the mean of squares of errors between labels and predictions. The equation is:

$$loss = (y_{true} - y_{pred})^2$$

Step 7: Fit Model

The model is fit using the keras library fit() function. This function takes as input the formatted and preprocessed training data from the TimeseriesGenerator() function earlier, an epochs arguments as an integer that signifies the amount of epochs through the data the model will be trained with, and a verbose argument set to 1 meaning the function shows a progress bar while it is running.

Step 8: Perform Predictions

The model performs predictions using the keras library predict() function. This function takes as input the formatted and preprocessed test data from the TimeseriesGenerator() function earlier and stores the predictions in a variable. This function returns a numpy array of predictions.

Step 9: Reshape Data for Plotting

The training data, testing data, and prediction data are reshaped from a 2-D ndarray to a 1-D ndarray. This is done with the numpy reshape() function. The input argument (-1) was passed into each reshape function. The 1-D training data, testing data, and prediction data were stored in variables.

Step 10: Plot Train Data, Test Data, and Prediction Data

3 scatter plots are then created using the plotly library. These 3 scatterplots are for plotting the training data, testing data, and prediction data. The dates are the x-axis inputs and the closes for each period are the y-axis inputs. Then these 3 scatterplots are used as inputs to create a figure object. Lastly, the show() function is used to display the data. Below is an example of the plot:

Step 11: Forecast Future Data

The process of generating the forecasted data is started by reshaping the original close data into a 1-D array. A variable is declared and set that represents the amount of time in the future we want to forecast. A function predicter() was created that creates a list of forecasts. It takes as input n the number of periods to predict, and the model used previously. It gets a list of the last n values in the close data as a list of forecasts, then loops n times. Each loop, the latest n values in the forecast list are reshaped into a 3-D array of shape (1, lookback, 1). Then the predict() function is used to generate a forecast. This function takes the 3-D array as input, and stores the output in a variable. The output is the next period forecast. That forecast is appended to the end of the forecast list. The loop then repeats. Now the prediction list has n+1 values in it, and we pull the latest n values. It is now using the forecast from the first loop, to make a new forecast. After n loops, the prediction list has lookback + n values. The lookback values are part of the test data so all these are removed except the last test data value. This is so we can connect the forecast plot to the test data plot. This results in n+1 forecasts. The function then returns the forecast list.

The next function is predict_dates(). This function takes as input n the number of periods to predict. This function is for generating the new future dates for the forecasts. It pulls the last value of the date column from the DataFrame, then generates n+1 dates and stores them in a list. Since the stock market is not open on weekends, it only generates weekday dates. The function then returns the prediction dates.

Step 12: Plot Forecast

The last step is generating 2 scatterplots. The first scatterplot is a small subset of test data from the end of the test data. The second scatterplot is the forecasted values. The x-axis is for the date and the y-axis is for the close values. These scatterplots are put into a figure object. Then the show() function is used to display the plot. Below is an example of the plot:

Experiment:

In this section, the experiments that were performed and the results of those experiments are presented. 5 different LSTM models are compared on 5 datasets. The lookback value is set to 15 periods. The batch size of the training data is 20 and the batch size of the testing data is set to 1. Each model is compiled with the optimizer that implements the Adam algorithm and the loss function of mean squared error. Each model was trained on 100 epochs. The forecasts are predicting 30 periods into the future.

There were 5 datasets that were applied:

Company Name	Ticker	Exchange	Date Range
Advanced Micro Devices	AMD	NASDAQ	12/15/1972 – 11/19/2021
Corsair Gaming Inc	CRSR	NASDAQ	9/23/2020 – 11/19/2021
Tesla Inc	TSLA	NASDAQ	6/29/2010 – 11/19/2021
Discover Financial Services	DFS	NYSE	6/14/2007 – 11/19/2021
Pfizer Inc.	PFE	NYSE	1/2/1968 - 11/19/2021

The datasets are made up of active stocks. 3 datasets are from the NASDAQ exchange and 2 datasets are from the NYSE. The amount of data in each dataset varies from 54 years of data to just over 1 year of data.

Loss Values Generated:

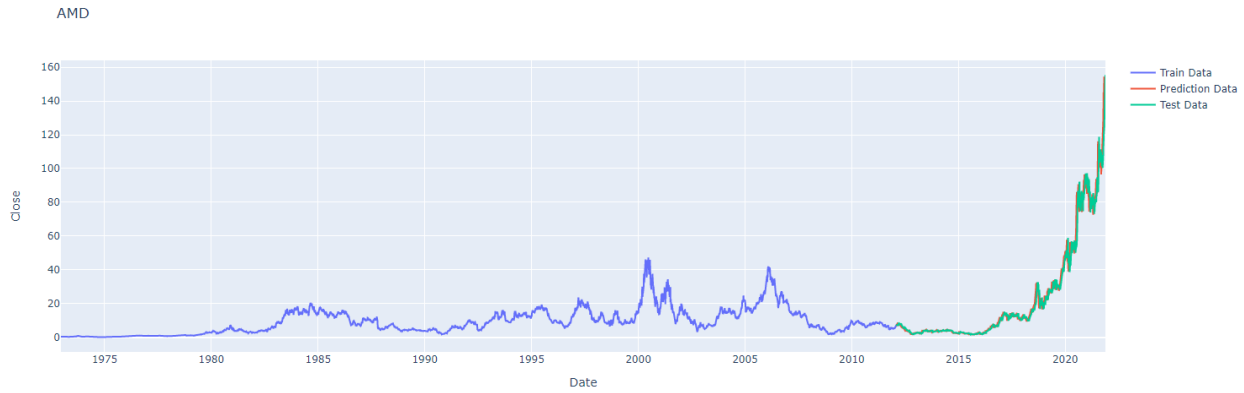
	AMD	CRSR	TSLA	DFS	PFE	Average
Model 1	0.2269	2.5501	1.7620	0.5672	0.1259	1.0464
Model 2	0.7205	5.8912	26.0886	8.4215	0.7728	8.3789
Model 3	0.2889	2.3697	1.9476	0.8669	0.6727	1.2292
Model 4	0.2325	1.8897	13.2991	10.1793	0.2113	5.1624
Model 5	0.8337	11.1112	12.0217	10.8401	1.0417	7.1697
Average	0.4605	4.7624	11.0238	6.1750	0.5649	

These are the loss values that were generated. Overall, the AMD and PFE datasets had the lowest loss scores with the average values of 0.4605 and 0.5649 respectively. This appears to be because the datasets were very big and the price action of the test data was similar to the training data. TSLA and DFS had very high loss values. The TSLA dataset had price action that was very different in the testing data than in the training data. DFS had price action that was somewhat different in the testing data than in the training data. CRSR was the smallest dataset but had the median average loss score. From these results, it seems like a larger size of dataset helps by giving the models more to learn, but it is perhaps more important to have training data that is like the testing data in getting lower loss scores.

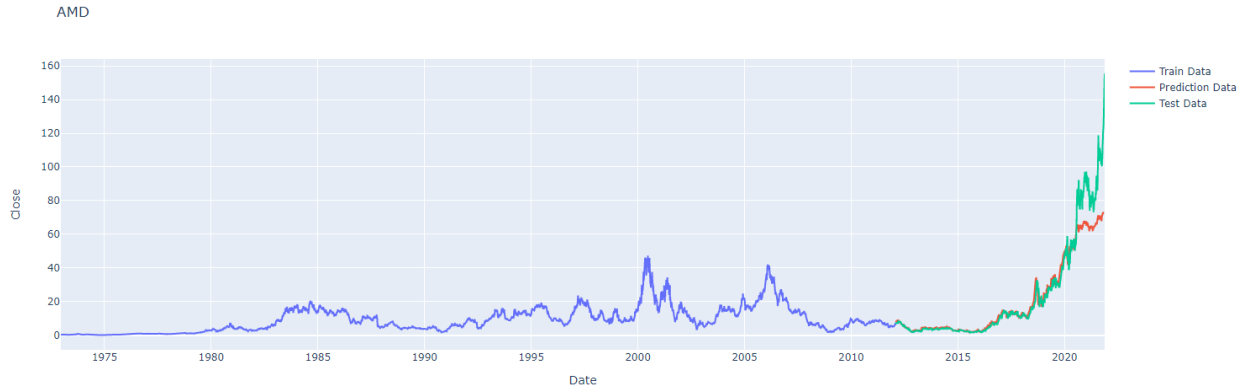
Model 1 and Model 3 had the lowest average loss scores of all the models. These models were also the most simple. Model 2 and Model 5 had the highest loss scores. These models were more complicated, and both contained Dropout layers. Model 4 had the median average loss score and was an LSTM autoencoder.

Training Data, Testing Data, and Prediction Data:

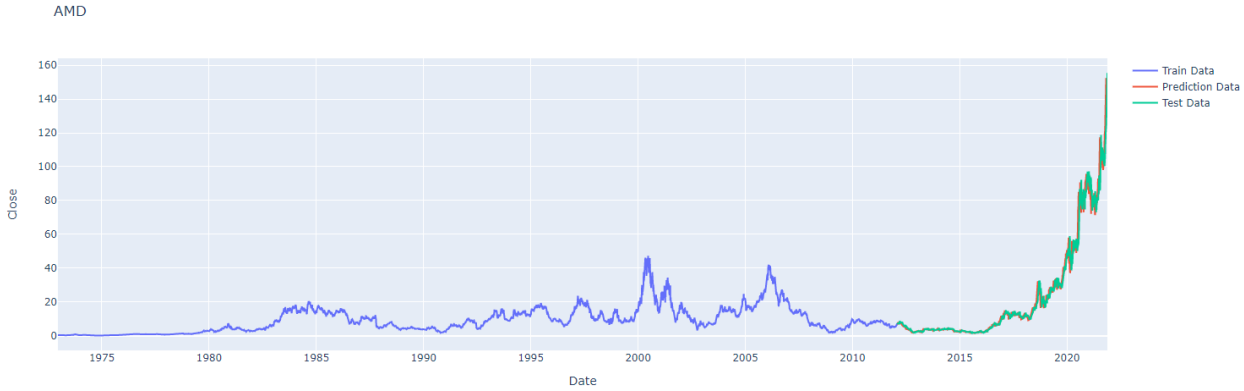
AMD dataset for Model 1



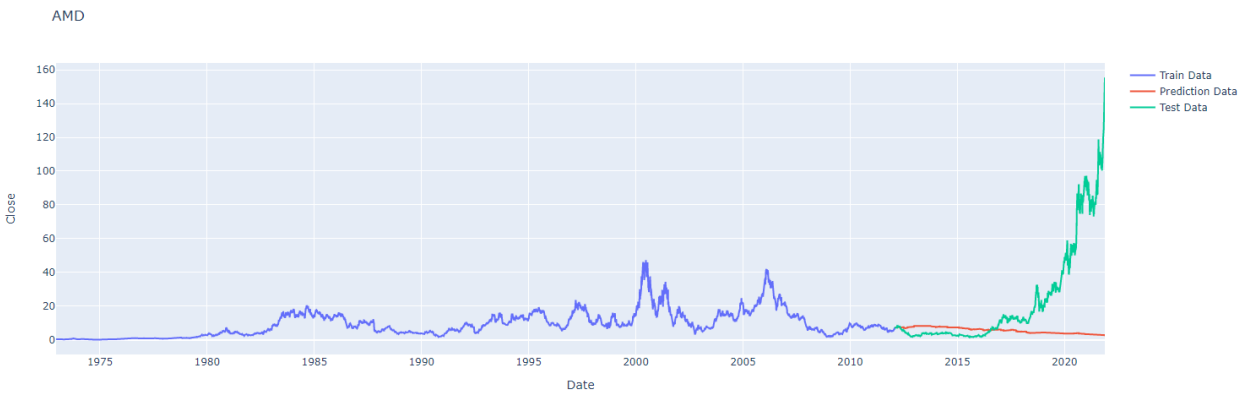
AMD dataset for Model 2



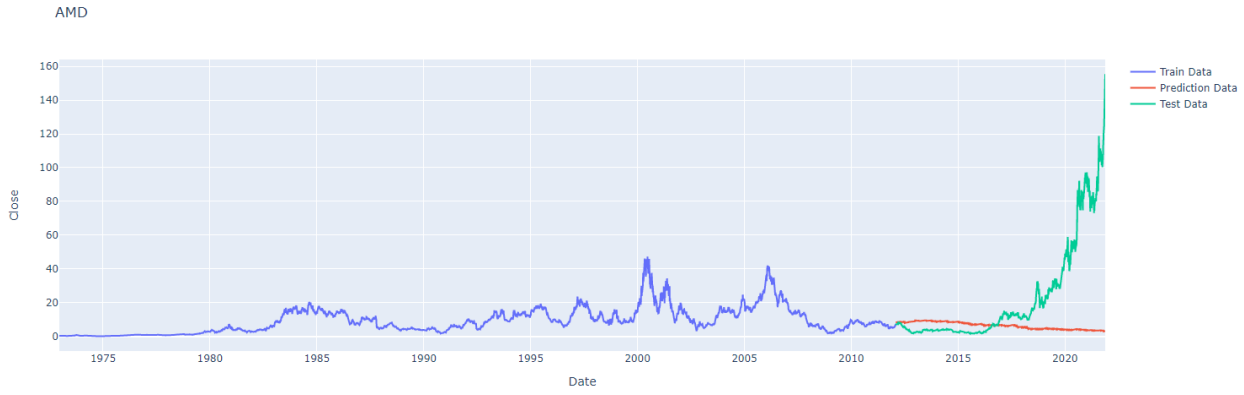
AMD dataset for Model 3



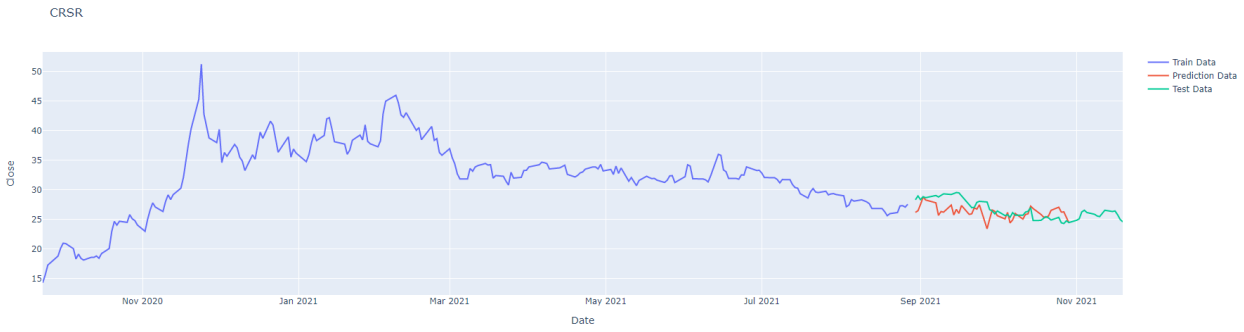
AMD dataset for Model 4



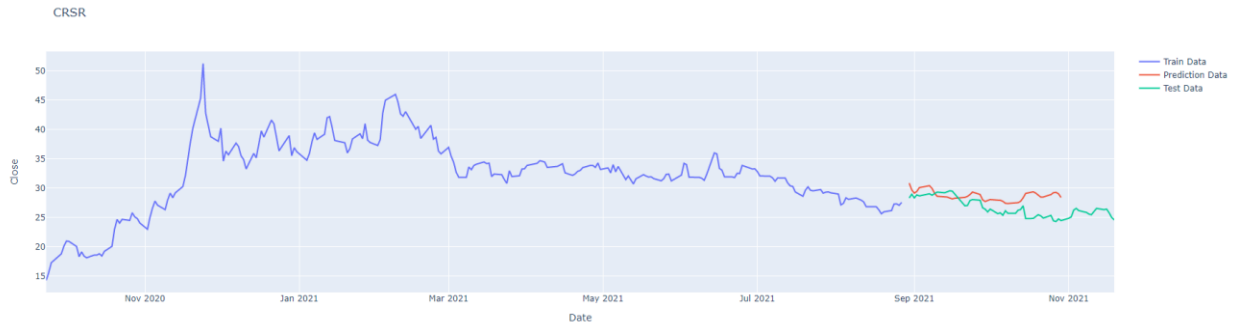
AMD dataset for Model 5



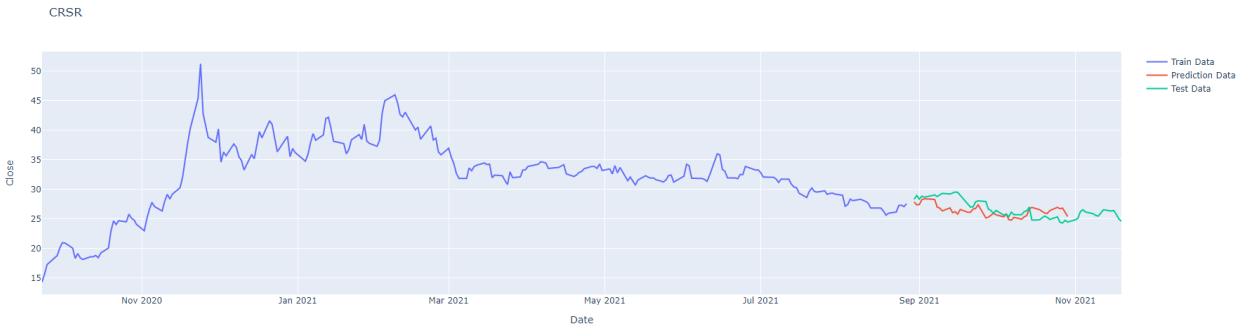
CRSR dataset for Model 1



CRSR dataset for Model 2



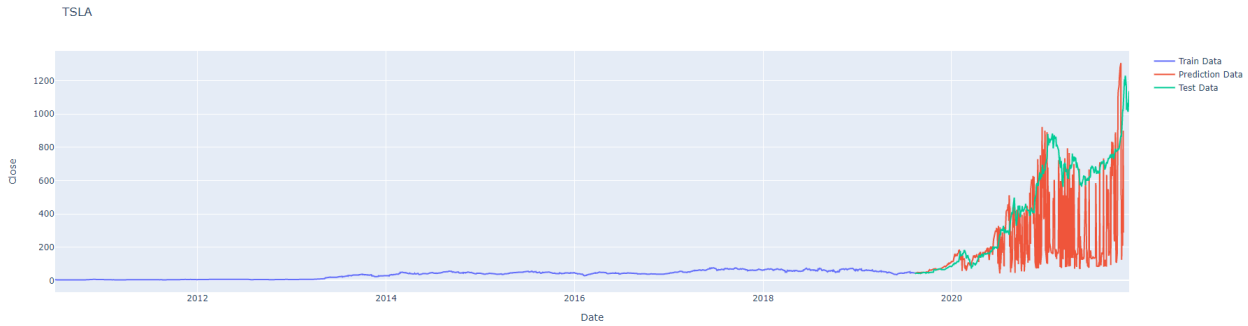
CRSR dataset for Model 3



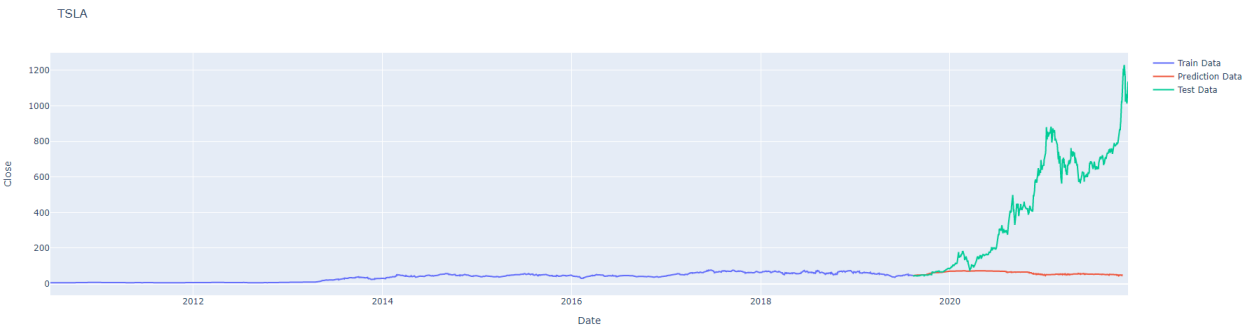
CRSR dataset for Model 4

CRSR dataset for Model 5

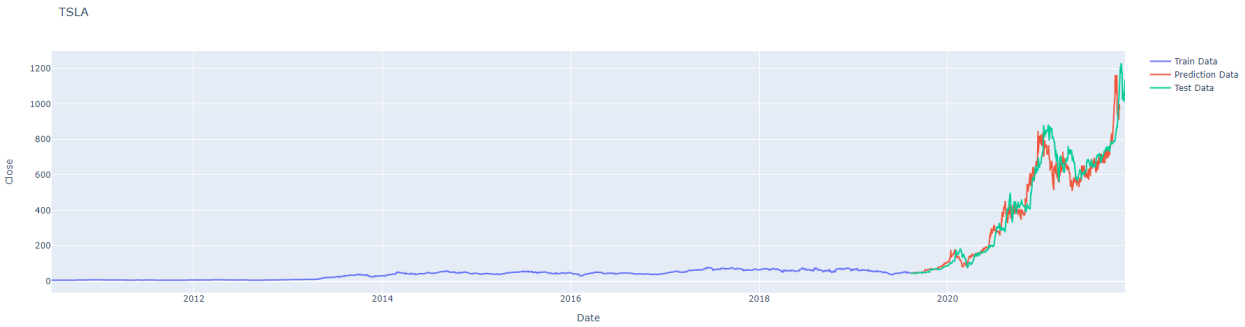
TSLA dataset for Model 1



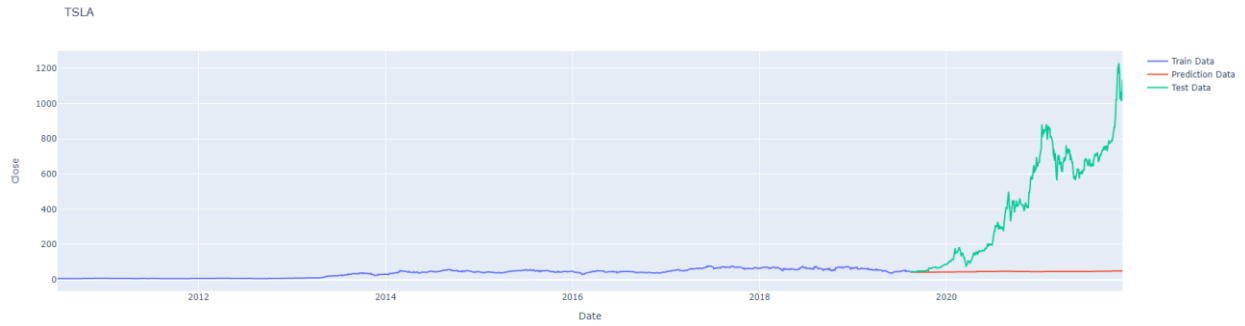
TSLA dataset for Model 2



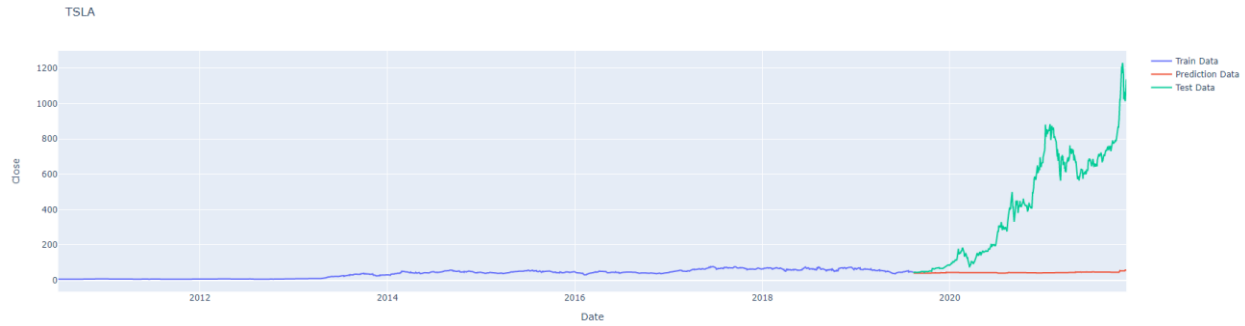
TSLA dataset for Model 3



TSLA dataset for Model 4



TSLA dataset for Model 5



DFS dataset for Model 1

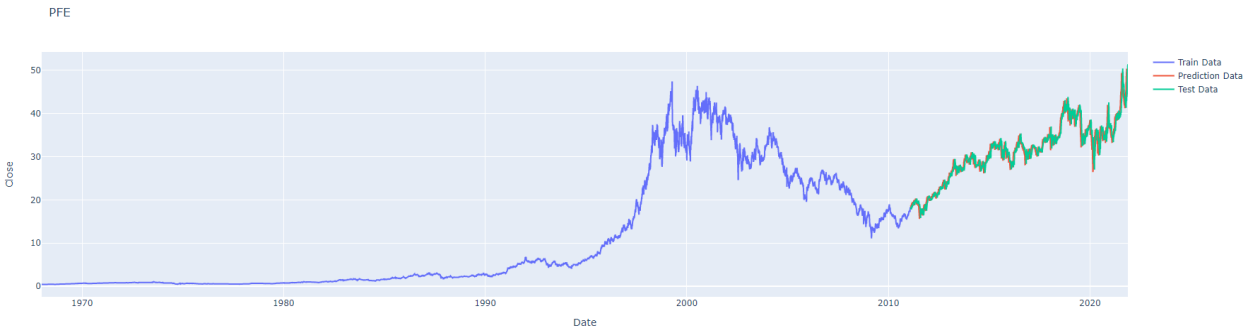
DFS dataset for Model 2

DFS dataset for Model 3

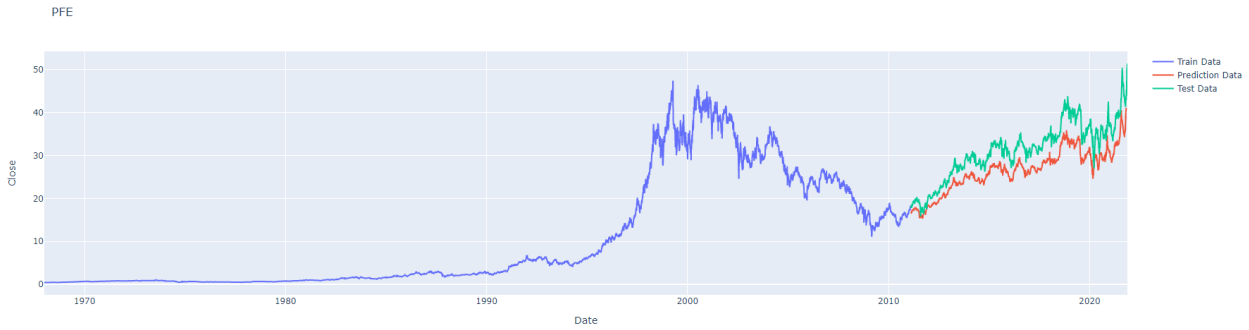
DFS dataset for Model 4

DFS dataset for Model 5

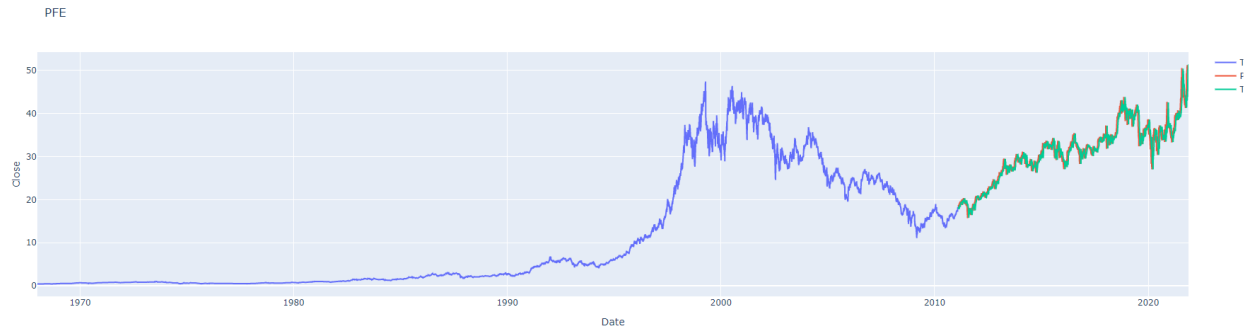
PFE dataset for Model 1



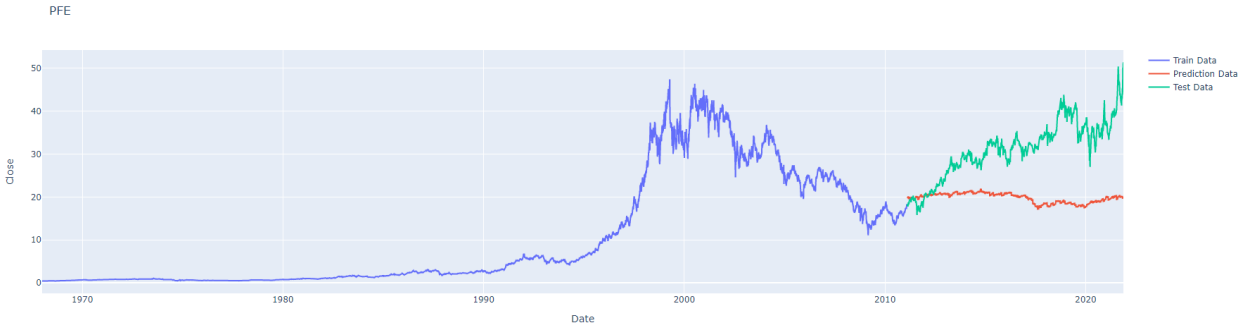
PFE dataset for Model 2



PFE dataset for Model 3



PFE dataset for Model 4



PFE dataset for Model 5

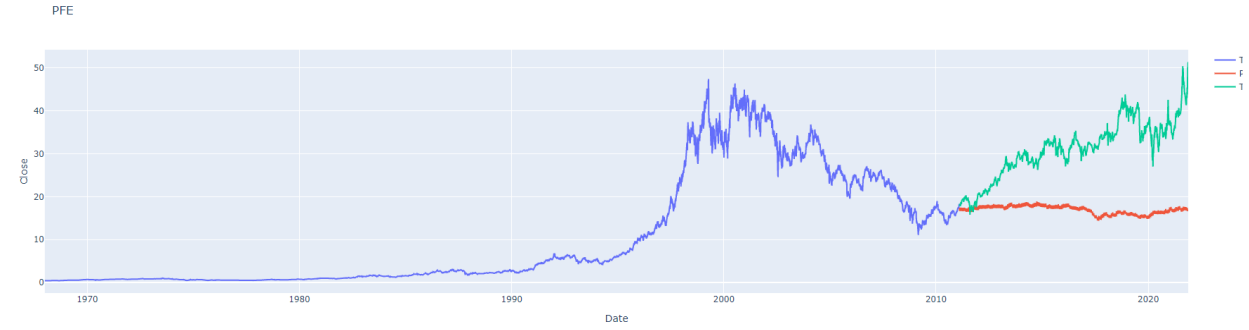


Table showing if prediction data was like testing data:

	AMD	CRSR	TSLA	DFS	PFE	Yes Count

Model 1	Yes	Yes	No	Yes	Yes	4
Model 2	Yes	Yes	No	Yes	Yes	4
Model 3	Yes	Yes	Yes	Yes	Yes	5
Model 4	No	Yes	No	No	No	1
Model 5	No	Yes	No	No	No	1
Yes Count	3	5	1	3	3	

These values were yes if the prediction data was similar in shape and value to the testing data. This was generated through sight only of the above plots. It is possible that this could be interpreted as "Was the run valid?" when deciding if this would be a good use for an investor.

The prediction data was most like CRSR. This may be a result of it having the smallest dataset. The prediction data was least like TSLA. This might be a result of the testing data that was very different from the training data. DFS, PFE, and AMD all had only 3 out of 5 models producing prediction data that was similar to the testing data.

Model 3 had the best performance generating similar prediction data to the testing data on every dataset. Model 4 and 5 had the worst performance generating similar prediction data to the testing data. Model 1 and 2 had relatively good performance generating similar prediction data on all datasets except TSLA.

Forecast Data:

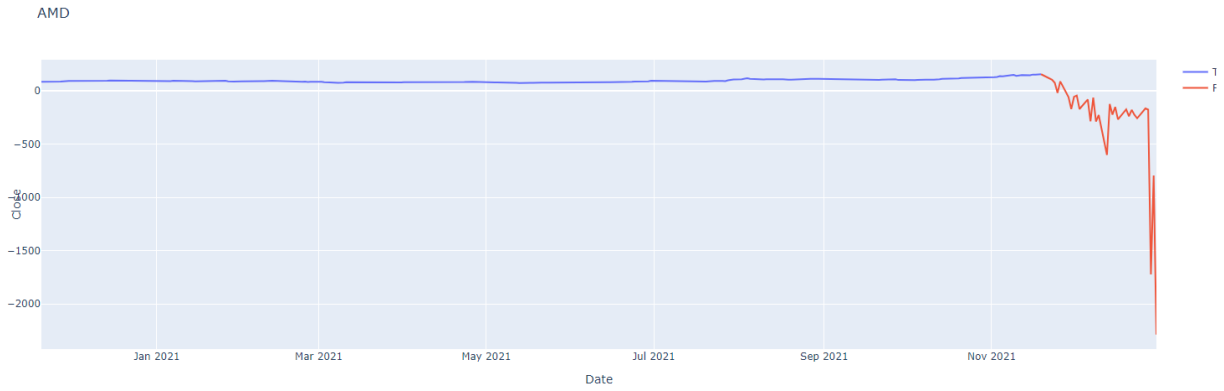
AMD forecast for Model 1

AMD forecast for Model 2

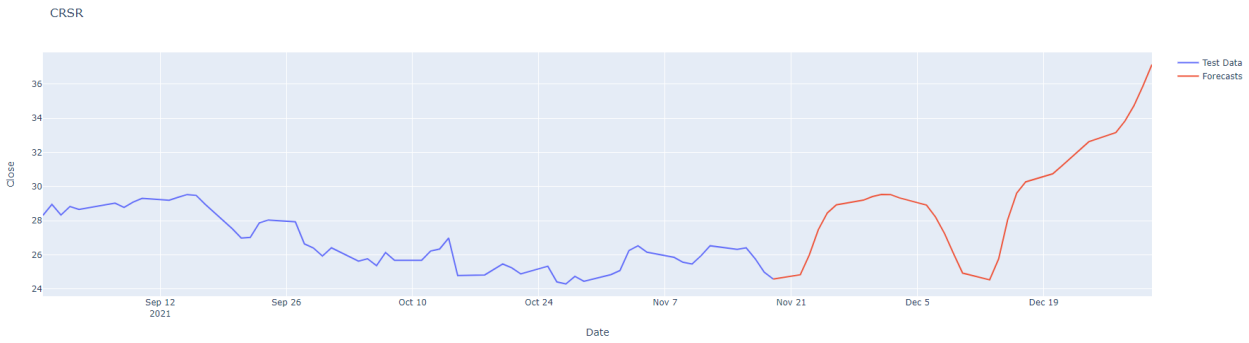
AMD forecast for Model 3

AMD forecast for Model 4

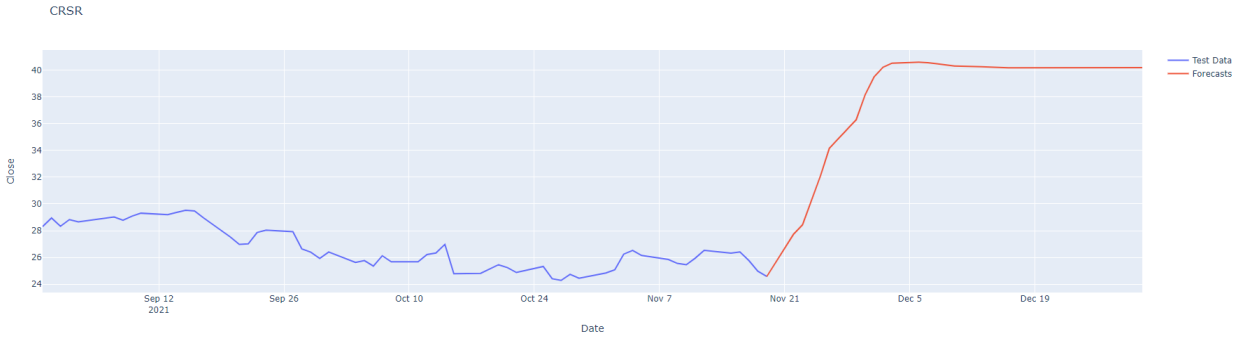
AMD forecast for Model 5



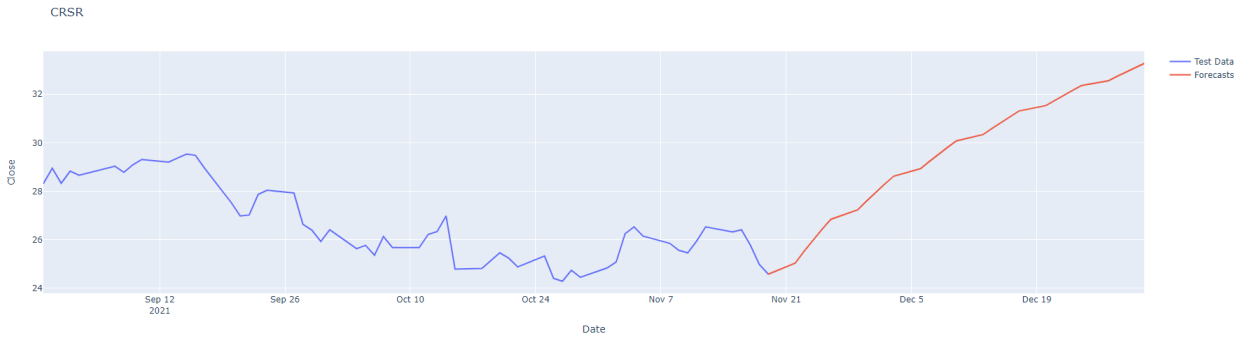
CRSR forecast for Model 1



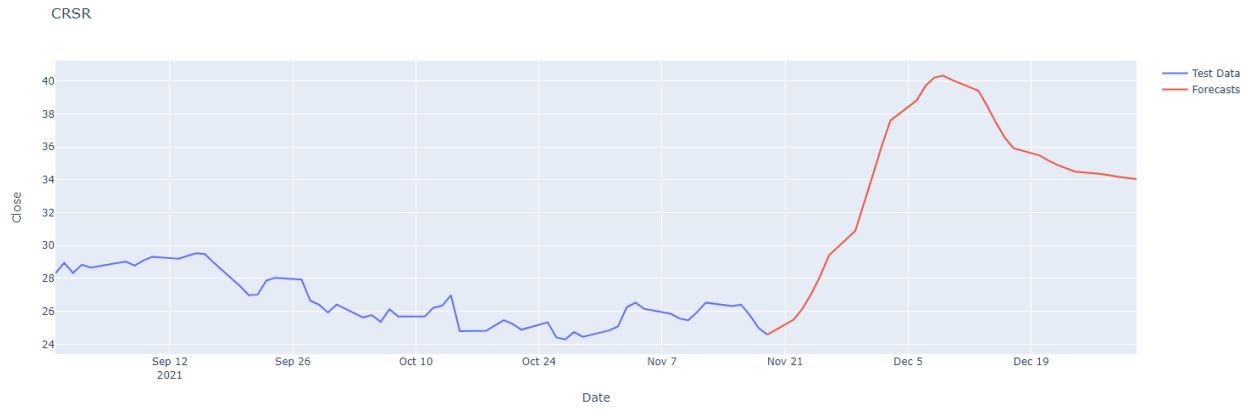
CRSR forecast for Model 2



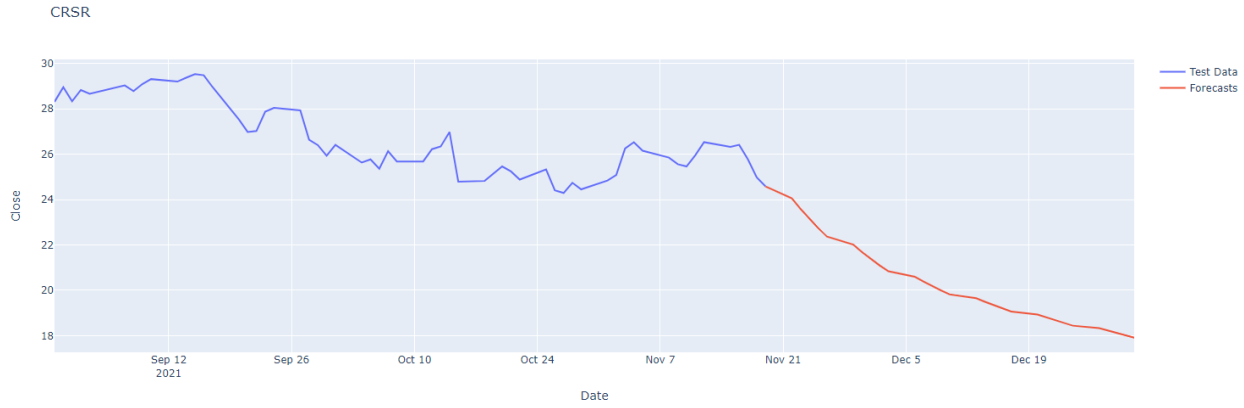
CRSR forecast for Model 3



CRSR forecast for Model 4



CRSR forecast for Model 5



TSLA forecast for Model 1

TSLA forecast for Model 2

TSLA forecast for Model 3

TSLA forecast for Model 4

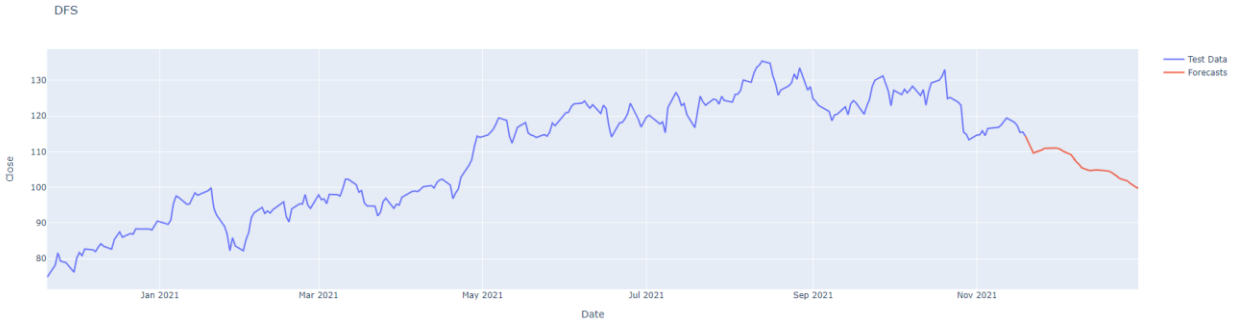
TSLA forecast for Model 5

DFS forecast for Model 1

DFS forecast for Model 2

DFS forecast for Model 3

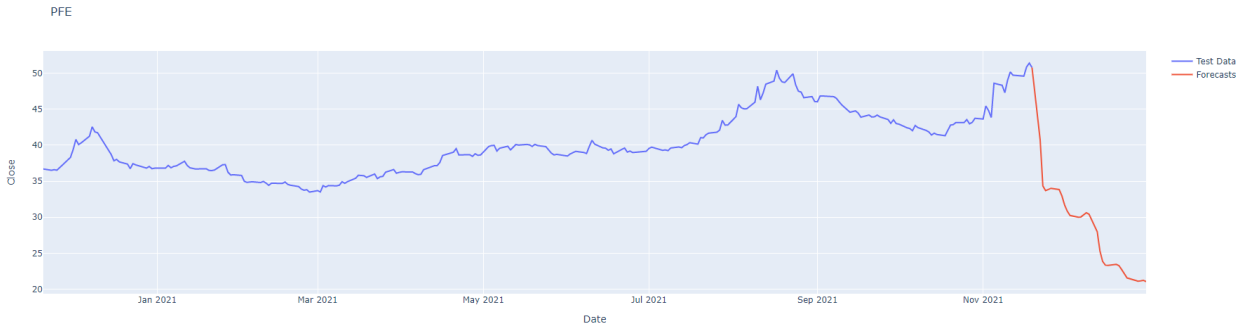
DFS forecast for Model 4



DFS forecast for Model 5

PFE forecast for Model 1

PFE forecast for Model 2



PFE forecast for Model 3

PFE forecast for Model 4

PFE forecast for Model 5

Forecast Up, Down, or Breakeven:

	AMD	CRSR	TSLA	DFS	PFE	Up Count
Model 1	Down	Up	Down	Down	Down	1
Model 2	Down	Up	Down	Down	Down	1
Model 3	Down	Up	Down	Breakeven	Down	1
Model 4	Down	Up	Breakeven	Down	Up	2
Model 5	Down	Down	Down	Down	Down	0
Up Count	0	4	0	0	1	

If the ending forecast value was higher than the last test value, then the forecast is predicting an increase in price. If the ending forecast value was lower than the last test value, then the forecast is predicting a decrease in price. If an investor was to use a model or group of models for forecasting and investment, this would be closer to useful output that could be translated into stock positions.

The Models are predicting that CRSR is most likely going up. The Models are predicting that AMD, TSLA, DFS, and PFE are most likely going down.

It is difficult to evaluate the models based on the direction of the forecasted values without a comparison of the data.

Because the author cannot predict the stock market with high accuracy, the author cannot evaluate the reasonability of the forecasts. A way of doing this is waiting 30 periods, pulling in the new close data, and comparing it to the forecasted values. Another method would be removing the last 30 periods of the current test data. In this way, the last 30 periods could be forecasted then could be immediately compared for accuracy. The forecast compared to the real data could show how much these forecasts deviate from the real price action. The findings could show an average deviation so the investor can know how much confidence to put on the forecast or a range to expect the future price to be in. This was not part of the research, but could be done in another attempt to get more useful results.

Features and Advantages from results:

It seems like deep learning can definitely be applied to timeseries data in order to make predictions and forecasts on the next n periods of the sequence. The further out the forecast, the harder it is to predict. It seems like LSTM models can be useful in forecasting timeseries data.

There were 3 sets of results: loss data, prediction data vs test data, and direction of forecast. From the loss data we rank the models like this where Model 1 has the lowest loss and Model 2 has the highest loss:

1. Model 1
2. Model 3
3. Model 4
4. Model 5
5. Model 2

From the prediction vs test data we rank the models like this where Model 3 had prediction data that looked the most like the test data and Models 4 and 5 had prediction data that looked the least like the test data:

1. Model 3
2. Model 1 and Model 2
3. Model 4 and Model 5

These two rankings would suggest that Model 1 and Model 3 would be the best for an investor to use to aid them in an investment strategy. Then the table of forecasts could be used to decide which direction to invest.

Investing/trading involves substantial risk. The author does not guarantee or otherwise promise as to any results that may be obtained from using this research report. Past performance should not be considered indicative of future performance. No reader should make any investment decision without first consulting his or her own personal financial advisor and conducting his or her own research and due diligence, including carefully reviewing any prospectus and other public filings of the issuer. These commentaries, analysis, opinions, and recommendations represent the personal and subjective views of the author and are subject to change at any time without notice. The information provided in this report is obtained from sources which the author believes to be reliable.

Conclusion:

This paper was a systematic investigation of LSTM networks and using them to predict stock prices. The methods included using date and closing price of 5 different publicly traded stocks to perform forecasting of future prices 30 periods after the end of the test data using univariate LSTM networks. There were 5 different model designs presented. These were compared with one another in order to evaluate each model's performance. The results showed that Model 1 and Model 3 performed the best on the 5 datasets having low loss scores and prediction data that resembled the testing data.

Runnable Codes:

```
import pandas as pd
import numpy as np
import keras
import tensorflow as tf
from keras.preprocessing.sequence import TimeseriesGenerator
import plotly.graph_objects as go
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout, RepeatVector, TimeDistributed
from keras.models import Sequential

# Read from file
df = pd.read_csv('/content/drive/MyDrive/Stock Price Data/NASDAQ_AMD_1D.csv')
# df = pd.read_csv('/content/drive/MyDrive/Stock Price Data/NASDAQ_CRSR_1D.csv')
# df = pd.read_csv('/content/drive/MyDrive/Stock Price Data/NASDAQ_TSLA_1D.csv')
# df = pd.read_csv('/content/drive/MyDrive/Stock Price Data/NYSE_DFS_1D.csv')
# df = pd.read_csv('/content/drive/MyDrive/Stock Price Data/NYSE_PFE_1D.csv')
print(df.info())

# Change date to datetime and get rid of all columns that are not date or close
df['date'] = pd.to_datetime(df['date'])
# df.set_axis(df['date'], inplace=True)
df.drop(columns=['time', 'open', 'high', 'low'], inplace=True)

# plot figure
fig = go.Figure([go.Scatter(x=df['date'], y=df['close'])])
fig.show()

# preprocess data
```

```
close_data = df['close'].values
close_data = close_data.reshape((-1,1))

split_percent = 0.80
split = int(split_percent * len(close_data))

close_train = close_data[:split]
close_test = close_data[split:]

date_train = df['date'][:split]
date_test = df['date'][split:]

# use timeseries generator to convert data to {features, target} format
look_back = 15

train_generator = TimeseriesGenerator(close_train, close_train, length=loo
k_back, batch_size=20)
test_generator = TimeseriesGenerator(close_test, close_test, length=look_b
ack, batch_size=1)

# Create Model # 1: LSTM + Dense
model = Sequential()
model.add(LSTM(10,activation='relu',input_shape=(look_back,1)))
model.add(Dense(1))

# Create Model # 2: 2 LSTM + 2 Dropout + Dense
# model = Sequential()
# model.add(LSTM(256, return_sequences=True, activation = 'relu', input_sh
ape = (look_back, 1)))
# model.add(Dropout(rate=0.3))
# model.add(LSTM(256))
# model.add(Dropout(rate=0.3))
# model.add(Dense(1))

# Create Model # 3: LSTM + 3 Dense
# model = Sequential()
# model.add(LSTM(10, activation = 'relu', input_shape = (look_back, 1)))
# model.add(Dense(200))
# model.add(Dense(100))
# model.add(Dense(1))

# Create Model # 4: auto-encoder-decoder
# model = Sequential()
# model.add(LSTM(128, return_sequences=True, activation='relu', input_shap
e=(look_back,1)))
```

```
# model.add(LSTM(64, return_sequences=False, activation='relu'))
# model.add(RepeatVector(look_back))
# model.add(LSTM(64, return_sequences=True, activation='relu'))
# model.add(LSTM(128, return_sequences=True, activation='relu'))
# model.add(TimeDistributed(Dense(1)))

# Create Model # 5: 2 LSTM + 2 Dropout + RepeatVector + TimeDistributed
# model = Sequential()
# model.add(LSTM(128, activation = 'relu', input_shape=(look_back,1)))
# model.add(Dropout(rate=0.2))
# model.add(RepeatVector(look_back))
# model.add(LSTM(128, activation = 'relu', return_sequences=True))
# model.add(Dropout(rate=0.2))
# model.add(TimeDistributed(Dense(1)))


# Show summary of model being used
model.summary()


# Compile Model
model.compile(optimizer='adam', loss='mse')


# Fit Model
model.fit(train_generator, epochs=100, verbose=1)


# Perform Predictions
predictions = model.predict(test_generator)


# reshape data for plotting
close_train = close_train.reshape((-1))
close_test = close_test.reshape((-1))
predictions = predictions.reshape((-1))


# Plot Train, Test, and Prediction Data
trace1 = go.Scatter(x = date_train, y = close_train, mode = 'lines', name
= 'Train Data')
trace2 = go.Scatter(x = date_test, y = predictions, mode = 'lines', name =
 'Prediction Data')
trace3 = go.Scatter(x = date_test, y = close_test, mode='lines', name = 'T
est Data')
layout = go.Layout(title = "Stock", xaxis = {'title' : "Date"}, yaxis = {'
title' : "Close"})
fig = go.Figure(data=[trace1, trace2, trace3], layout=layout)
```

```
fig.show()


# Forecast future close data
close_data = close_data.reshape((-1))

def predicter(num_prediction, model):
    # get last n entries in the array
    prediction_list = close_data[-look_back:]

    for _ in range(num_prediction):
        x = prediction_list[-look_back:]
        x = x.reshape((1, look_back, 1))
        out = model.predict(x)[0][0]
        prediction_list = np.append(prediction_list, out)
    prediction_list = prediction_list[look_back-1:]

    return prediction_list

def predict_dates(num_prediction):
    last_date = df['date'].values[-1]
    prediction_dates = pd.bdate_range(last_date, periods=num_prediction+1)
.tolist()
    return prediction_dates

num_prediction = 30
forecast = predicter(num_prediction, model)
forecast_dates = predict_dates(num_prediction)


# Plot forecasts
trace1 = go.Scatter(x = date_test[-252:], y = close_test[-
252:], mode='lines', name = 'Test Data')
trace2 = go.Scatter(x = forecast_dates, y = forecast, mode='lines', name =
 'Forecasts')
layout = go.Layout(title = "Stock", xaxis = {'title' : "Date"}, yaxis = {'
title' : "Close"})
fig = go.Figure(data=[trace1, trace2], layout=layout)
fig.show()
```