Recurrent Neural Networks to Predict Stock Data

James Wensel
Computer Science
Kansas Stat University
Manhattan, Kansas
jdwensel@ksu.edu

***Abstract*—The stock market is a seemingly random and unpredictable environment that does not lend itself to regular Neural Network approaches. Each day of stock market data does not exist in a vacuum, as each day the price of a stock is affected by its performance on previous days, as well as outside factors that cannot be taken into account in a neural network. While it is impossible to consider factors such as news surrounding a company or earnings announcements into predicting the stock market, it is however possible to make prediction based on previous days of data using Recurrent Neural Networks. This paper seeks to do just that, by implementing a Simple RNN, a LSTM RNN, and a GRU RNN, and comparing their performance on multiple different stocks.**

I. Introduction

According to the Bank of International Settlements, $5.1 Trillion is spent in the stock market each day [1]. With so much money being traded, it is no wonder it is a point of interest for data analysts. On top of the amount of money involved, there are over 3,500 stocks in the U.S. Stock market alone [2] that can be traded, and every stock generates hundreds to thousands of data points each day as stock prices change by the second. With not only so much money, but such ample data sources, there is an obvious draw to the stock market for data analysis and machine learning. While it may seem like a perfect place to practice applications of machine learning, there are important issues that need to be addressed to complete any kind of adequate stock market analysis.

The first and most important issue is that there are factors that affect the stock market that cannot be seen from simply looking at stock prices. The value of a stock is heavily tied to the company it is associated with. If the CEO of a company makes a statement about the company, it can have drastic impacts on the price of a stock, and that is simply impossible to account for. No data analysis or machine learning technique will be able to account for the performance or activities of the company and those who are associated with it. This means that it will never be perfectly possible to predict the future of any stock, no matter how good a model is. It is, however, possible to make predictions based on the available data that can make predictions as to the future of a stock *without* outside interference. This distinction shows that there is still value the process of analyzing stock market data.

The second important issue is the time dependent nature of the stock market. No single data point of a stock is unaffected by the others. Even disregarding the impact of factors outside the stock market, the nature of the stock market is temporal, as each data point is reliant on those before it. This means that any data model that attempts to predict the future performance of the market will need to have temporal dependencies as well. That is where Recurrent Neural Networks come in.

A recurrent neural network (RNN) is a type of neural network that allows for a time dependent input. They use the output of previous layers to affect the input of future layers. This provides the temporal quality that is necessary for stock market predictions, and they are the model of neural network that will studied in this paper. There are multiple kinds of neural networks, including Simple-RNN, Long Short-Term Memory RNN (LSTMs), and Gated Recurrent Units (GRU). These three model types will be explored in depth and trained with stock market data to see which performs best on the task of predicting future stock market values.

The Simple RNN is (obviously) the simplest of the three kinds of RNN. It works by maintaining its own hidden state [3], a set of weights that are updated in similar manner to normal feed-forward neural networks. It uses an activation function and both hidden state and input weights, which are updated like the weights of normal neurons in a non-recurrent feed forward network. The equation for calculating the hidden value and the output at any time are given in Eq (1-2), where σ represents the activation function, w_h is the hidden weight, and w_i is the input weight. Eq (2) shows that the output still has a set of weights, as it would in a feed-forward neural network, but the value it is being applied to is what makes the network recurrent. While the simplicity of Simple RNNs makes them easy to understand they do have one major problem: because of their simplistic nature, they are not able to apply long term effects of inputs. This means that the further away from the initial input, the less effect the RNN will see from the input even if it is still drastically important. That is where LSTMs come in.

$$hidden_t = \sigma(w_h * hidden_{t-1} + w_i * input_t)$$

Eq (1)

$$output_t = w_{output} * hidden_t$$

Eq (2)

The LSTM is a special kind of RNN that is designed for Long-Term memory [4]. It does this by maintaining a Cell State that is used in combination with both the current input and previous output to create the current output. The input and previous output are combined into multiple internal parameters that are used to update the Cell state, then once again combined at the end with the newly updated Cell state to find the current output. These new internal parameters, and the continued application of the input and output allow for much greater and longer time dependencies of the data.

Finally, the GRU works by combining the input and previous output again multiple times, but it does not maintain a cell state. It combines the input and previous output into two "gates": an update gate that tells how much of the previous output to keep and a reset gate telling how much information to forget. These gates are then applied to the previous output and current input to create an intermediate memory value, which is then combined one last time with the previous output value to get the current output [5]. It is the application over multiple steps of the reset and update gates that allow the GRU to filter the information that is being remembered. This allows the GRU to create a more long-term memory dependency then the Simple RNN.

II. Methodology and Approach

In order to perform stock market analysis, there are important steps that need to be taken before training a model. These include data preparation and model creation. These steps will be covered in the following subsections.

A. *Data Preperation*

First, it was necessary to acquire a dataset that can be used for model training. A publicly available dataset from Kaggle was used for this section. The data provides the Open, High, Low, Close, and Volume trading data for hundreds of days of over 7,000 stocks. Volume was not used for training as it does not provide any relevant information to the performance of a stock either good or bad, and a metric called Mid was instead used, which calculates the midpoint of the High and Low values for a given day.

There was a necessity to process this data, and a Stock Data class was created for this purpose using Pandas, Numpy, and Sci-Kit Learn. It takes as a parameter the Pandas dataframe generated from reading the csv of any stock and splits the data into a training and testing set, with 90% of the data being used for training and the last 10% for testing. This was not done using the Sci-Kit Learn train-test-split function, as is often the case with neural networks, but a custom function to split the data was created. This is because the typical split function splits the data and randomly assigns value to both the training and testing sets, making sure 90% of the values are in the training set and 10% are in the testing set. This is all done to create more fair training and testing sets. For the purposes of stock market analysis, this is not possible. It is necessary to keep the data in chronological order to maintain the temporal reliance of each datapoint, and this is why a new split function was created. The newly split data is then scaled to be between 0 and 1, with 0 corresponding to the lowest value of the current stock and 1 being the highest.

Finally, the data is reshaped so that 60 days of data will be input with every time step. This means that for each datapoint, the previous 60 days will be considered. 60 was selected because it is, at minimum, 2 months of previous data that will be considered when processing each day of data. This is more than ample time to account for the temporal relation of the stock price. After processing has been completed on the training set, an identical set of processing steps is performed on the test set. However, instead of using the test set generated when the data was split, the entire data sequence is used, including the 10% of data that was excluded. This is to allow correct application of the temporal relations of all the previous stock data on all the current data. It is then scaled and reshaped as previously occurred.

The last step of data preparation happens after the model has been trained and made predictions. In order to get the prediction value, the output of the model must be put through an inverse transform. This is because, at the time of prediction, all values are scaled to be between 0 and 1, so the prediction is also in that scale. It can go above 1 if the prediction is that the value of the stock will go higher than the previous peak but will most likely be between 0 and 1. The inverse transform of the data puts this value back into the realm of real values, giving the value predicted of the stock instead of the scaled value.

B. *Model Creation*

In order to generate Simple RNN, LSTM, and GRU models, a model generation program was created. This program uses Keras, a deep-learning framework built on top of TensorFlow, to define the different kinds of models that will be used. It allows the user to specific everything from optimization and loss functions to the type of recurrent layer it will use, and even whether to include dropout layers or not. Based on these user specifications, it creates a sequential model that feeds from one layer to the next. This structure can be seen in Fig (1) below. Then, the specified number and types of layers are added, each with the number of recurrent units per layer being specified by the user. A dropout is added after each recurrent layer if selected, and with the dropout rate as given. Finally, a fully connected dense layer is added which transforms the input from the final RNN layer to a single neuron, which will give the predicted value of our stock. This model is then saved for later training and testing use.

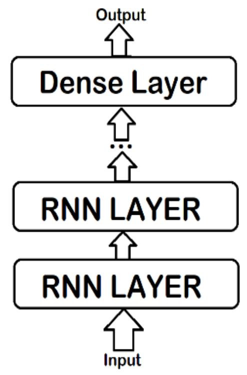

Fig (1)

C. *Optimizer, Loss, and Unit Values*

When building a neural network, one of the most important choices that determine how successful the model will be is what optimizer and loss functions to use. The optimizer functions reshape the model to be its most accurate form, while the loss functions calculate the distance from predicted values to true values during testing. In order find the best optimizer and loss function, tests were performed using the LSTM model with 4 LSTM layers, and dropout layers with a 0.2 percent dropout rate after each layer. These tests will be covered in the experiments section.

The last important model parameter is the number of RNN Units for each layer. A RNN layer is made up of multiple recurrent units of the same RNN type that are interconnected. These units each have their own internal connections between the previous output from that unit and the current input. These connections are what create the distinction between the different types of RNNs and cannot be altered. However, it is necessary to determine the optimal number of units as that will determine the dimensionality of the output space of each layer and is crucial to the success of the model. As with the optimizer and loss functions, tests were performed on an LSTM model with a similar structure and using consistent optimizer and loss functions. These tests will be covered in greater detail in the next section.

D. *Training*

After the data preparation and tests determining the best loss and optimizer functions, as well as the units for each RNN layer, 3 models were created to those specifications, with Simple RNN, LSTM, and GRU layers. These models were then trained on the stock data, using their High daily values. Once trained, the models then make a prediction on the testing set associated with the current training set. The predictions are put through multiple loss functions and the results are saved to a csv file to later be sorted and analyzed. After each step, a graph is also created, comparing the actual stock value and the predicted stock value over the current dataset, then the graph is exported. Examples of the graphs are shown in the results section.

III. Experiments

The first set of experiments were done to find the best optimizer function. Then, the best loss function was found. Next, tests were performed to find the optimal number of Recurrent Units for each layer. Finally, three models were trained with the dataset and tested on a set of 500 random values, and on a set of the 5 largest companies in the United States. The first three experiments were done with only a LSTM Network, and the final experiment was done with all 3 model types. The results of these experiments can be seen in the next section.

An important part of this research is the way in which the results were sorted. A python program was built using Pandas that allowed for a ranking of the different results. For each experiment, 6 different loss functions were used. The results of the loss functions were sorted in one of three ways to get the results. First, it created a list with the results from a single one of the loss functions. It then sorted the list from best to worst and gave each metric being tested a score equal to its index in the sorted array. It then repeated this process for each loss function, and after each step increased the score for each metric by its index in the sorted array for the current loss function. Once this had been completed, the entire set of results was sorted according to its score, called its “Rank”, with the lowest rank being the best metric.

The other two ranking methods followed the same procedure, but first scaled the data for each loss function in one of two ways. First, each experiment’s loss function results were scaled by the lowest value found, and then it was scaled by the highest value found. Then the same process was taken to find the rank as with the initial sorting. Once the list was created with the results from a single loss function, each element in the list was then divided by either the highest or lowest value in the list, depending on which sorting type the program was currently performing. Then, when calculating the score of each metric, the value used was the scaled value for the current metric instead of its index in the sorted array. The process then proceeded exactly as the first sorting method.

These three sorting methods each provide unique insight into the performance of the different results. The first, simply called rank, gives an overall ranking of the different metrics based on their performance in all the different loss functions. This is the best sorting method to show which metric performed best or worst. The next metric, scaling the results by the lowest value, shows how much worse each metric did then the best metric in that given loss category. For example, if for one loss value, the lowest metric got a 2 and the worst metric got an 8, the value for the low scaled sorting would give the first metric a 1 and the second metric a 4. This means it is 4 times worse than the best metric in that category. 4 is then added to the score for the second metric, and 1 is added to the score for the first. So, this sorting method tells, over all the different loss functions, how much worse each metric did then the best. The final method scaled each metric regarding the worst metric, which, in a very similar way to the other scaling method, tells how much better each metric is then the worst value for a given loss function. These three sorting methods are used for each of the experiments to see the results of the tests.

A. *Optimizer Function*

The Keras API offers 8 different options for optimizers. 5 of these are based on the Adaptive Movement Estimation, or ADAM, optimizer. The different types of optimizers were used in a LSTM Network with 4 Recurrent Layers and 50 units per layers. Mean Squared Error was used as the loss function for each of the tests. The network was then tested using the stock data for the tech company Apple. It was selected because it is one of the largest companies in the US, and the dataset had over 8000 datapoints for it alone. The predictions of this were tested using 6 of the 7 available loss metrics from the Keras API. The results were then ranked based on their performance in each metric, and these are discussed more in the next section.

B. *Loss Function*

As mentioned in the previous section, the Keras API includes 7 different loss functions. In the same manner to the previous tests, an LSTM Network with 4 Recurrent Layers was trained on Apple stock data. The optimizer used for these tests was Nesterov-accelerated Adaptive Moment Estimation, or NADAM. After training, the model made predictions based on the Apple stock data, and these predictions were tested on 6 of the 7 metrics and ranked based on performance in the same manner to the previous test.

C. *Recurrent Units*

In order to find the optimal number of Recurrent units, the same structure of LSTM Neural Network as the previous section was used. The optimizer function used was NADAM and the loss used was Mean Squared Error. The number of units tested began at 5, then 10-60 units was tested incrementing by 10 each time. Tests were also done using 32 and 64 units. In the same manner as the previous experiments, the model was trained on Apple stock data, and the predictions from the model were tested using 6 of the 7 loss metrics, with the results ranked.

D. *Training*

For this section, 3 different models were created. The first had Simple RNN layers, the second had LSTM layers, and the final had GRU layers as the Recurrent Layers. Each model had 4 Recurrent Layers, with a Dropout Layer after each that had a dropout rate of 0.2. Finally, a fully connected Dense layer was used to condense the results to a single value. When being trained, the data was trained in batch sizes of 32, with 10 epochs used for each stock.

These three models were then trained on the first 3,000 stocks using High daily values. Only 3,000 stocks were used instead of the entire 7,000 for two reasons. The first is not all stocks in the dataset were large enough to be fed into the neural network. As mentioned in the data cleaning section, 60 days of data was fed into the network at a time, so any stock that had less then 70 days was removed. This was to make sure that the stocks being used had enough data to be formatted properly, as well as ensuring there were at least 10 datapoints to be trained on for each stock. This left around 6,700 stocks that were useable for training. The second reason only 3,000 stocks was used is time. It took over 48 hours to train on 3,000 stocks, and there was not adequate time to train over the entire dataset. High values were not selected for a specific reason as no dataset inherently has more value over the others, and the training could be repeated with a different dataset for similar results.

Once the models were trained, they were tested on a random set of 500 stocks, and their predictions were measured by 6 of the 7 loss metrics. The average loss from each metric was calculated for each model and compared to see which model was better. The three models then ran again on a set of the stock data from the top 5 companies in the US: Apple, Amazon, Facebook, Google (Alphabet), and Microsoft. They ran on all 5 datasets to show that no one performs significantly better than the others, and the average loss was again calculated.

A final training test was done by creating new versions of the 3 models with new weights, and training/testing them on just the 5 top stocks to compare their results to the trained stocks. The results of this experiment, as well as all previous experiments, are discussed in the next section.

IV. Results

A. *Optimizer Function*

For this, the model was trained over 10 epochs with a batch size of 32. After the model was trained, the different results can be seen in Appendix A Table (1). The results from the rank sort can be seen below in Fig (2).

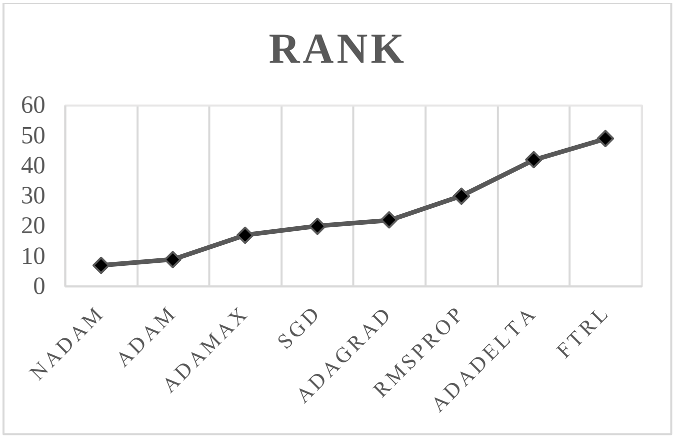

Fig (2)

This sort shows a clear distinction between the optimizer functions. NADAM and ADAM are the best, ADAMAX, SGD, and ADAGRAD all perform similarly, RMSprop is not bad, but is a little behind, and ADADELTA and FTRL are by far the worst metrics. This point is illustrated better by the high and low value scaled sorting. The graphs for this are shown in Fig (3-4).

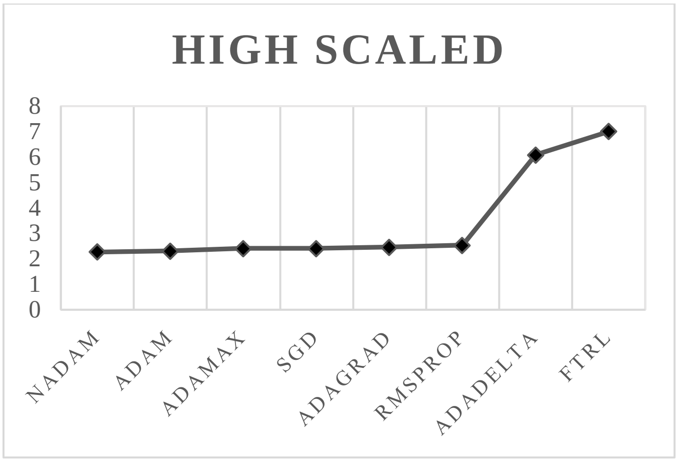

Fig (3)

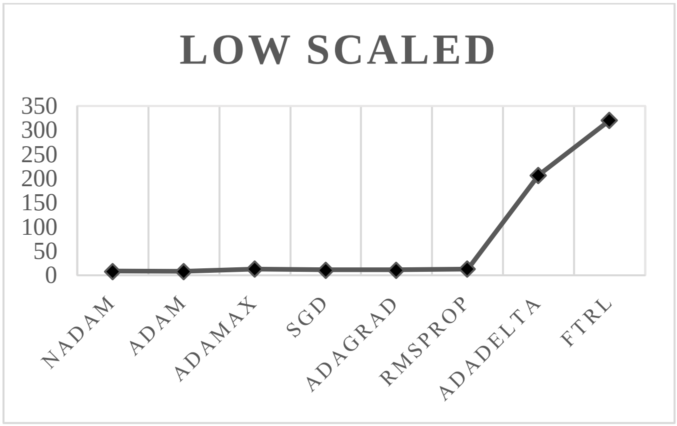

Fig (4)

As these two values clearly show, the ADADELTA and FTRL metrics are simply not able to compare with the other optimizers for this problem set. As a result, they are not offered as optimizers for the model creation tool designed for this paper. This also illustrates why NADAM is the default optimizer, and the optimizer used for the models created in later experiments of this paper. It was the best metric in all but the low scaled sorting, where it took second by .2 points.

B. Loss Function

The same process was used for this section, with NADAM as the optimizer function. The Model was tested using 10 epochs and a batch size of 32. The numerical results can be seen in Appendix B Table (2). The results from rank sort can be seen below in Fig (5). The loss functions used are Mean Squared Error (MSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), Mean Squared Logarithmic Error (MSLE), Cosine Similarity (CS), Huber (H), and Logarithm of the Hyperbolic Cosine (LC).

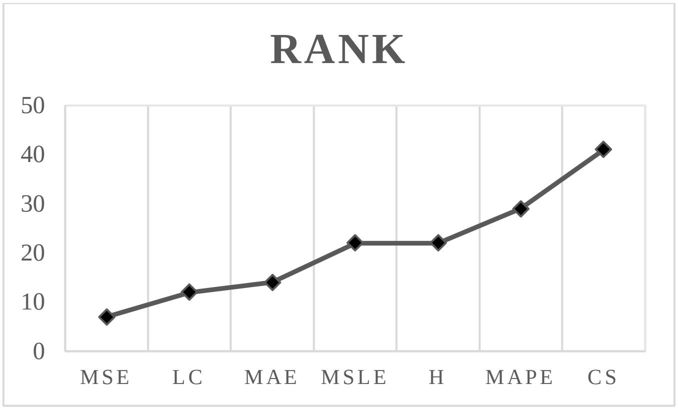

Fig (5)

In a similar manner to the optimizer there are a few different metrics that seem reasonable and a few that do not, but from this alone we can’t tell where the line is. We do know, however, that MSE, LC, and MAE are all very good measure, MSLE, H, and MAPE are also reasonable, and CS seems bad. Fig (6-7) provide more insight by showing the High and Low scaled sorting.

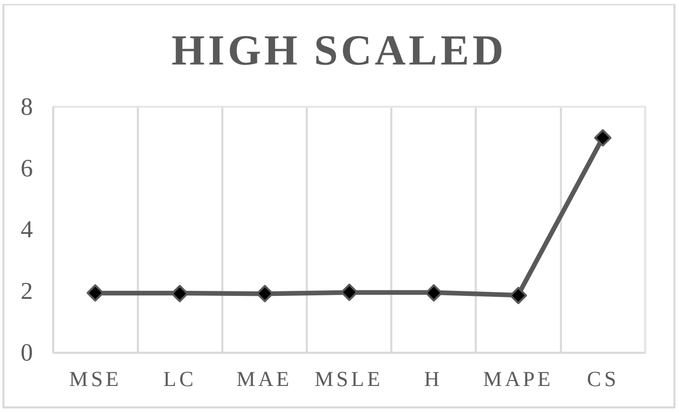

Fig (6)

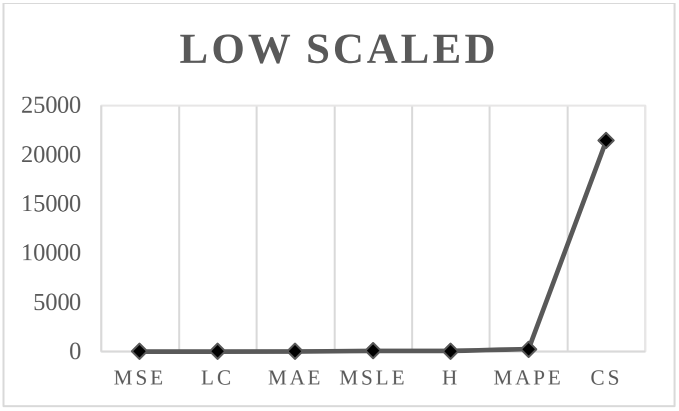

Fig (7)

These make the observations from the first graph more apparent. MSE, LC, and MAE are the best, H, MAPE and MSLE not as good but still reasonable, and CS is a terrible metric for this problem. For that reason, CS is not a possible option in the model creator, and it was not used to evaluate any of the other metrics in the previous section. An important note is Cosine Similarity is not a bad metric, but it does not apply well to this problem set so it will not be considered.

C. Recurrent Units

For this section, NADAM was used as the optimizer, and MSE was used as the loss function. The model was trained on Apple stock with a batch size of 32 and 10 epochs. The numerical results are shown in Appendix C Table (3), and the graphs for the 3 sorting metrics are seen in Fig (8-10).

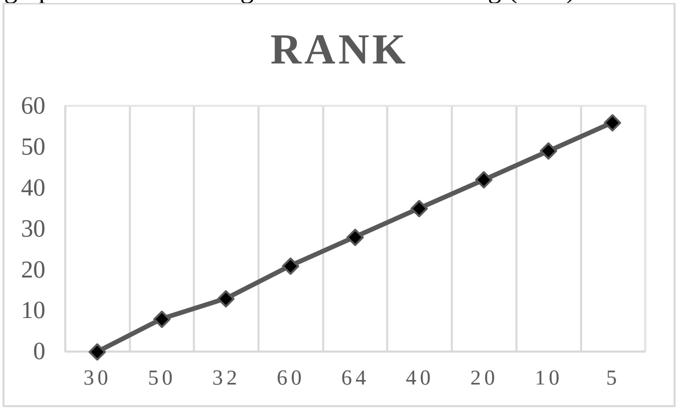

Fig (8)

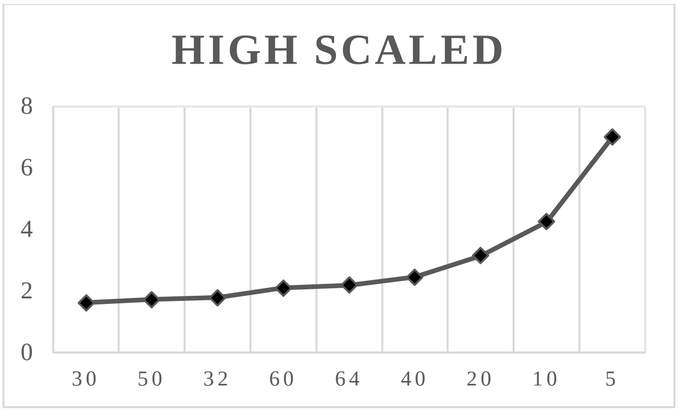

Fig (9)

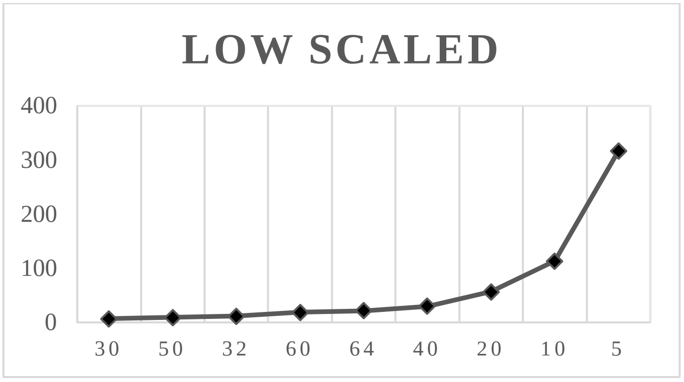

Fig (10)

Similar to the other experiments done before this, there is an obvious worst, and clear grouping for the other results. However, this is the first metric where the results were the same for each of the 3 types of sorting. The rank order was the order of each model setup. With this metric, no changes were made to the model creation tool, but 50 was used as the default instead of 30. While 30 performed better on this test, 50 was instead chosen to help better exploit the temporal nature of the data and make sure as many data points are being considered at a time as possible.

D. 3,000 Stock Training

For the models trained on 3,000 datapoints, as discussed earlier, there were 4 Recurrent Layers with a dropout after each. Each Recurrent Layer had 50 units, used the NADAM optimizer, and the MSE loss function. One model used Simple RNN layers, one used LSTM layers, and one used GRU layers. They were then tested on a random subset of 500 stocks. The numerical results from these tests can be seen in Table (4).

	Models					
	MSE	***MAE***	***MAPE***	***MSLE***	***H***	***LC***
Simple RNN	5.48E9	3018.2	23222	0.51	3017	3017
LSTM	5578.5	4.11	17.88	0.04	3.89	3.86
GRU	8.8E6	192.4	421.3	0.522	191.9	191.7

Table (4)

From this, the best model is very apparent. By far the best of these three is the LSTM network. It is not even very close. In every metric, the LSTM network is significantly better. This is also seen in the graphs that were generated by each model, 3 of which can be seen in Fig (11-13). Fig (11) is the RNN Graph, Fig (12) is the LSTM graph, and Fig (13) is the GRU graph. All three graphs are the actual price vs the predicted price of the Apple stock price. All graphs were made with the Matplotlib python library.

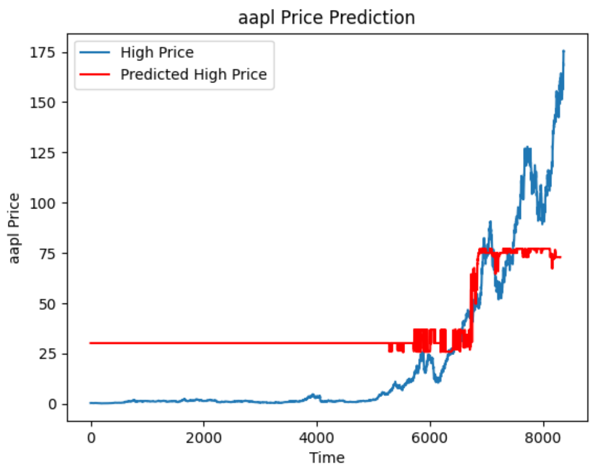

Fig (11)

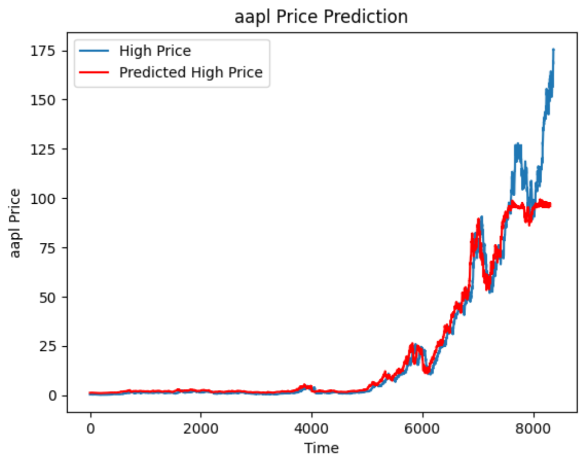

Fig (12)

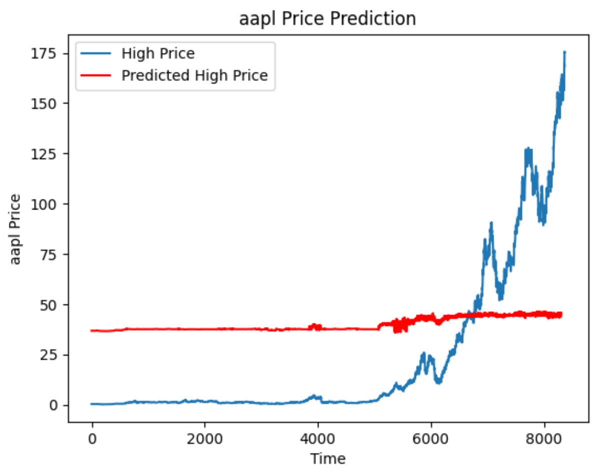

Fig (13)

While on the surface, it seems as though the LSTM model is simply the best, after careful examination of the other two networks results, the reason becomes apparent. The other two models suffered overtraining at some point. While this means no knowledge is gained about which of the three models performs better under ideal conditions, it does tell us one very important thing: LSTM neural networks are much more resistant to overtraining then the other models. While not the information that was desired to be gained from this test, still a very valid result.

E. 5 Stock Training

After the overtraining that took place with the 3,000 stock values, the 5 largest companies in the US were used as training data instead of the entire training set. Also, the models were trained on all 5 data sets (High, Mid, Low, Open, and Close) instead of using only the High data. The numerical results from this are shown in Table (5).

	Models					
	MSE	***MAE***	***MAPE***	***MSLE***	***H***	***LC***
Simple RNN	1215.4	11.51	24.53	0.20	11.10	10.99
LSTM	1117.9	8.49	17.46	0.06	8.08	7.98
GRU	2050.8	9.635	14.59	0.05	9.23	9.14

Table (5)

These values tell a much different story. They show that, while the LSTM model is still the best in half the metrics, the GRU model is actually better in the other half. The RNN model is also worse than both other models, but not significantly. This is also to be expected, as the other two can model long term relationships much better then a Simple RNN network can. To show their performance, graphs for the Apple stock High Value predictions of each model is shown in Fig (14-16), Fig (14) is the RNN model, Fig (15) is the LSTM network, and Fig (16) is the GRU network.

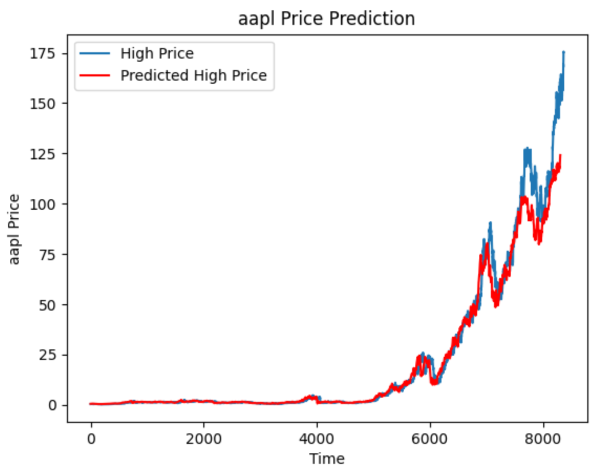

Fig (14)

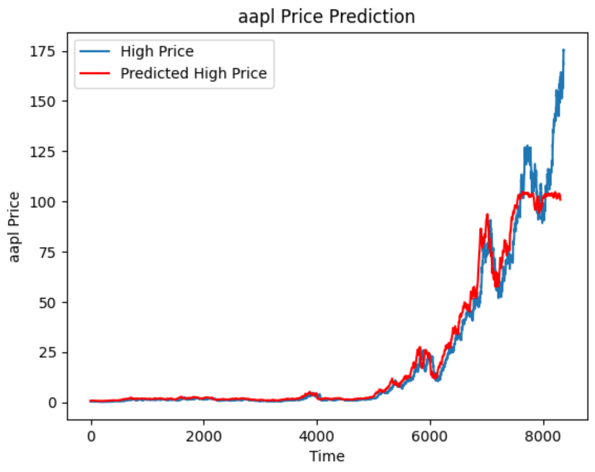

Fig (15)

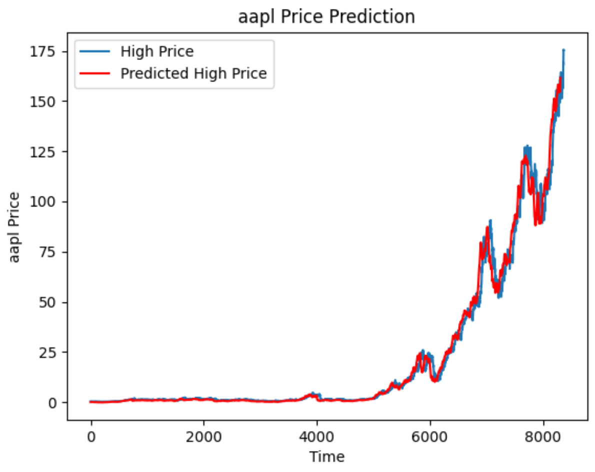

Fig (16)

From analyzing these graphs, the LSTM model was surprisingly the worst. GRU was by far the best predictor in the case, and even the Simple RNN did better than the LSTM. However, this is not always the case. For example, the

Microsoft High value is modeled best by LSTM, as can be seen in Fig (17-19) with Fig (17) being the Simple RNN, Fig (18) being the LSTM, and Fig (19) being the GRU model.

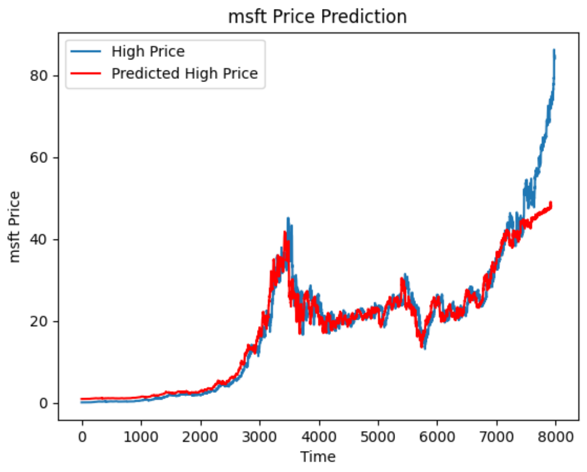

Fig (17)

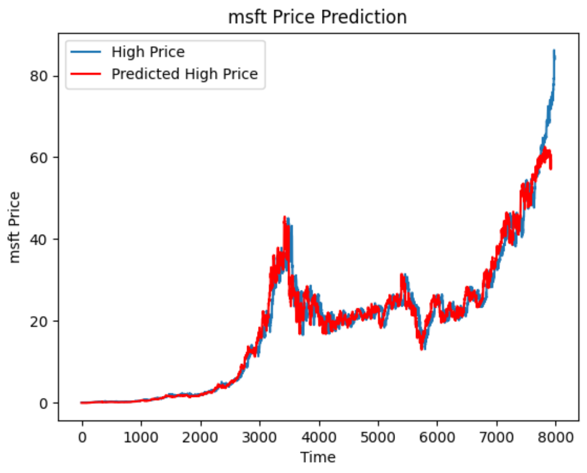

Fig (18)

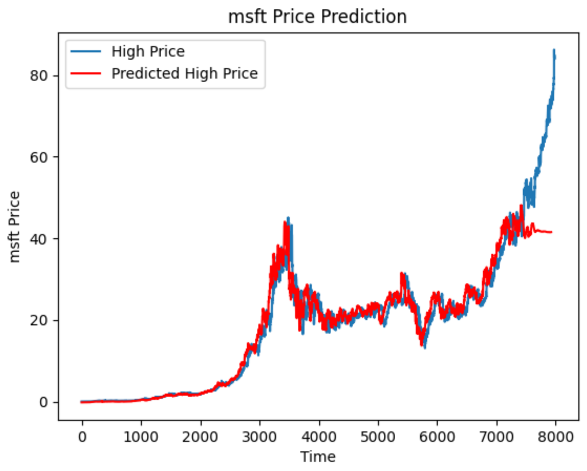

Fig (19)

With this case, the LSTM is the best model, RNN is the second best, and GRU is the worst. Ultimately, these show that, when overtraining has not occurred, all three perform well, but each model will have different stocks that it works better on.

V. Conclusion

For the experiments of the Optimizers, Loss Functions, and the number of Recurrent Units per Layer, the findings of this paper will not be applicable for any other research. The main reason is the tests were performed on the dataset for this problem, and the results of the tests performed will have different results in a different domain. It is extremely likely that the different optimizers and loss functions will have a different order when applied to a different problem set. However, the process outlined in this paper to find the results of those experiments can be applied to any other problem and any other type of Neural Network and will provide the same amount of information they did for this paper.

Once the models were created, two main conclusions quickly formed. The first is that the LSTM network was much more robust and was able to survive overtraining better then the GRU and Simple RNN models. This means that for the other two models, more care must be taken when training them to test different datasets, so the models do not overtrain. The second result is that, for the majority of stocks either the LSTM or GRU model will perform better, but neither can be determined until the models have run.

The final point to note is the performance of the models. While, for the majority, they were able to model very closely the stock data, they did struggle with staying consistent at the end of the time period that they were training on. This was mostly the case when the value of the stock was continuously climbing at the end of the time period the models were predicting. While this does mean the models would predict a dip in price when one does not exist, it is much better that the models are too conservative with their predictions compared to being too optimistic. If the model were too optimistic, and someone were to use the model to predict a stock's future value, then made stock purchases based on that data, they could stand to lose a lot of money. However, the models created here would instead predict the value to be lower than it would become. While this could cost the user potential profits, it would not lose them money.

VI. Further Research

The most obvious way this research could be extended is through the use of different Neural Networks. This would allow for the determining of which networks are well suited to this type of problem and which are not. Based on these results, model types could even be mixed to see if better results come from a mixed RNN model then a standard one. The problem could also be extended into different domains of Machine Learning such as Reinforcement Learning, where an agent could be trained by having it attempt to make the most money possible from the stock market. This would involve the model getting higher values from the reward function based on how much money was made from the transactions of buying and selling specific stocks. This could be a very interesting utilization, and the comparison of the performance between that model and the results from this paper could provide insight into the stock market, the relationship between neural networks and reinforcement learning agents, and the performance differences between the two types of machine learning algorithms.

Another way this paper could be easily extended is through the use of a Stock API to get daily stock data. It would then be easy to select a set of stocks to monitor, and have the model track them each day and predict what it believes the stock will do. Then, based on those predictions, make an assessment as to what should be done with each stock, i.e. should some be bought, some be sold, or all be sold. This could then easily be extended into a "financial advisor" program, where a model gives predictions of the best stocks to be invested in at a given moment, as well as which stocks it feels will not be successful in the long term.

The final interesting domain this research could be extended to is different kinds of investments. It could be applied to ETFs, or even cryptocurrency. The results of experiments done in those markets could be compared to results of the different models in the general stock market to see whether the models are better at predicting normal stocks, or any of these specialized investments. If good enough, the models could be used to compared expected returns to very safe investments, like bonds, to decide what is the most profitable investment strategy at a given time. This could easily be combined with the previous extension to make a robust financial advisor program that gives advice based on both long and short term expected returns from the market. The stock market is a very vast field, and the potential for improvements to this project are numerous.

VII. References

[1] Scutt, David. "Here's How Much Currency Is Traded Every Day." *Business Insider*, Business Insider, 2 Sept. 2016, www.businessinsider.com/heres-how-much-currency-is-traded-every-day-2016-9.

[2] Krantz, Matt. "Chasing Right Stocks To Buy Is Critical With Fewer Choices But Big Winners." *Investor's Business Daily*, 27 Nov. 2020, www.investors.com/news/publicly-traded-companies-fewer-winners-huge-despite-stock-market-trend/.

[3] Loye, Gabriel. "Beginner's Guide on Recurrent Neural Networks with PyTorch." *FloydHub Blog*, FloydHub Blog, 16 Aug. 2019, blog.floydhub.com/a-beginners-guide-on-recurrent-neural-networks-with-pytorch/.

[4] "Understanding LSTM Networks." *Understanding LSTM Networks -- Colah's Blog*, colah.github.io/posts/2015-08-Understanding-LSTMs/.

[5] Kostadinov, Simeon. "Understanding GRU Networks." *Medium*, Towards Data Science, 10 Nov. 2019, towardsdatascience.com/understanding-gru-networks-2ef37df6c9be.

VIII. Appendix

The following sections contain tables with the data form the different experiments. For the loss functions used: MSE – Mean Squared Error; MAE – Mean Absolute Error; MAPE – Mean Absolute Percentage Error; H – Huber; LC – Logarithm *of the Hyperbolic Cosine.*

A. Optimizer Function

	Optimizers					
	MSE	***MAE***	***MAPE***	***MSLE***	***H***	***LC***
ADAM	58.78	2.37	16.68	0.012	2.08	2.04
ADAMAX	38.50	2.67	43.24	0.049	2.25	2.19
ADAGRAD	99.17	3.46	13.20	0.015	3.19	16.34
ADADELTA	841.2	17.02	452.68	1.47	16.53	16.34
FTRL	1665	22.98	466.65	2.43	22.47	22.28
NADAM	37.99	2.08	22.48	0.022	1.75	1.70
RMSprop	109.21	3.84	20.25	0.024	3.50	3.43
SGD	71.19	3.02	23.68	0.022	2.67	2.64

Table (1)

B. Loss Function

	Loss Function					
	MSE	***MAE***	***MAPE***	***MSLE***	***H***	***LC***
MSE	46.27	2.42	26.99	0.022	2.07	2.03
MAE	56.11	2.68	10.16	0.008	2.43	2.37
MAPE	424.0	16.94	936.8	0.96	16.16	15.97
MSLE	32.79	3.83	191.1	0.40	3.34	3.16
CS	438279	661.12	42857	23.58	660.6	660.4
H	43.07	2.97	104.6	0.204	2.49	2.35
LC	58.38	2.55	15.09	0.011	2.26	2.21

Table (2)

C. Recurrent Units

	Recurrent Units					
	MSE	***MAE***	***MAPE***	***MSLE***	***H***	***LC***
5	1735	37.74	3151.3	6.42	37.25	37.06
10	490.8	20.21	1681.4	4.19	19.71	19.53
20	192.5	12.6	1064.5	2.92	12.11	11.93
30	7.52	2.51	181.8	0.44	2.02	1.86
32	18.73	4.14	259.4	0.51	3.64	3.46
40	74.88	7.94	658.5	1.91	7.45	7.27
50	11.26	3.09	243.82	0.65	2.61	2.45
60	35.95	5.50	457.5	1.33	5.02	4.84
64	44.75	6.17	506.1	1.48	5.68	5.50

Table (3)