Comparison of RNN and LSTM RNN on Time Series Prediction of Stock Prices

Ashley Handoko
Department of Computer Science
University of Texas at Dallas
Richardson, USA
ashley.handoko@utdallas.edu

***Abstract*— Creating accurate models for time series predictions have become more of an interest in the machine learning community and has many application that can be used in the modern world. One of the common use cases for time series predictions is for stock market predictions. Recurrent Neural Networks (RNNs) have been a relatively new model used on time series data. Research was done to compare the differences of a basic recurrent neural network and a variation of RNN called a Long Short-Term Memory (LSTM) RNN. Evaluation was performed on a test set of time series data to predict stock market close prices of an index, using the Root Mean Square Error (RMSE) to evaluate. Through trying to optimize both models, the lowest RMSE for the basic RNN and LSTM on test data respectively were 17.411932686041567 and 10.339394350893345 respectively. This research and evaluation suggests and supports the claim that LSTMs are more accurate on time series predictions, as RNNs are not able to properly retain information from longer time steps away.**

Keywords—recurrent neural networks, long short-term memory, time series prediction

I. Introduction and Background Work

Recurrent Neural Networks (RNNs) are a type of neural network that is able to take into consideration previous data, which is where it gets its name "recurrent". The basic neural network model has shown to be less performant on data where the ordering of the data influences the output, for example time series data. With recurrent neural networks, the network is able to take into consideration previous inputs and will perform its model fitting on the sequences of data it is provided. Therefore recurrent neural networks are often used to model time series data.

Vanilla recurrent neural networks have been shown to fall victim to vanishing gradient and exploding gradient issues. As suggested by their names, vanishing and exploding gradients are when gradients become too small or too large, respectively. Both cases are not ideal and can lead to poorly trained models. As potential solutions to these, specific implementations of RNNs have been developed as well, one of which is the long short-term memory (LSTM) recurrent neural network. RNNs seem unable to retain relevant information for long periods of times, but LSTM RNN models have been able to learn longer dependencies based on the previous data as well. The next sections will dive deeper into the algorithms between basic RNNs and LSTMs, as well as a comparison and analysis of using both models on the same dataset for time series prediction of stocks.

II. Theoretical and Conceptual Study

A. Basic Recurrent Neural Network

The general concept of a recurrent neural network is a neural network with hidden layers which are fed the same weights which allow them to understand and learn from the series passed through them and keep the relevant information in the time series data. As the term recurrent suggests, they perform the same calculations for each sequence of data passed through the model and computations are based on output from previous passes that it retains information on.

Similar to neural networks, the logic behind recurrent neural networks is very similar. The output is calculated by doing a forward pass on the data and back propagation is performed to calculate the error and adjust the training weights. A representation of a recurrent neural network cell and description of components can be found in Fig. 1. At the highest level, the recurrent neural network passes in input, has hidden states, and then produces an output.

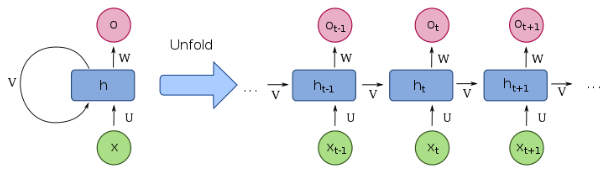

Fig. 1. Recurrent Neural Network Diagram (Figure 1 represents a recurrent neural network cell. On the left is the generalized representation of an RNN cell, with the input state (x), hidden state (h), and output state (h) as well as their weights (U, W, V) respectively. On the right is the RNN represented unfolded, representing the layers through time.) *[1]*

One major difference between recurrent neural networks and regular neural networks is the time component. Each of the calculations must be performed for each time step defined for the inputs passed to the network. For example, the back propagation step is actually back propagation through time – the backward propagation calculation must be performed through every time step unrolled. The depiction of the RNN cell unrolled can also be found in Fig 1.

B. *Long Short-Term Memory RNN*

The Long Short-Term Memory RNN is based on the basic RNN, with some additions. Recurrent neural networks are susceptible to “forgetting” information for long time sequences. As a result, LSTMs were created and they have shown to be more performant for time series predictions over longer time sequences and are able to handle the vanishing gradient problem of recurrent neural networks. LSTM cells have a few components to them, visible in Fig 2.

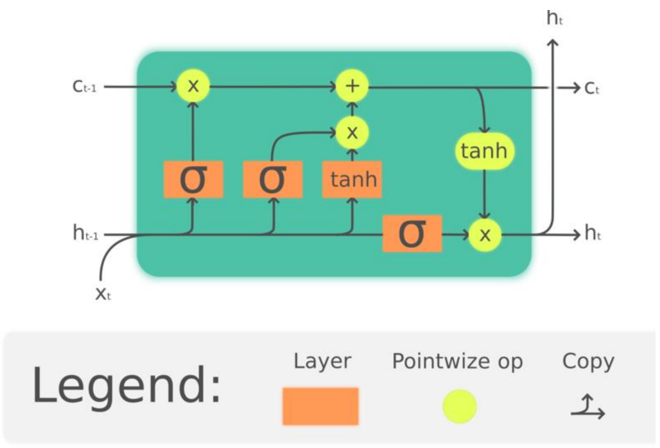

Fig. 2. Long Short-Term Memory Cell Diagram (Figure 2 represents a recurrent neural network cell.) *[2]*

Long short-term memory cells are composed of a forget gate, an input gate, and an output gate. The forget gate determines what it is important from previous data. The forget gate will use a sigmoid function, which takes in any value and produces an output between zero and one to represent the weight or importance of data to take into consideration. The input gate utilizes both the sigmoid and tanh functions to assign weights to the importance of the inputs passed in. The tanh function is basically a scaled version of the sigmoid function and takes in a value and produces an output between negative one and one. Lastly, the output gate takes into account the input and the memory of the cell to produce the output of that step.

III. Results and Analysis

Once understanding was reached on how both of the models work, research was done utilizing both the basic recurrent neural network and long short-term model for comparison. Both programs utilized PySpark notebooks to create the RNN models and perform the predictions and evaluations. The goal of the models was to predict the close stock price based on the time series data. The dataset for the time series prediction was a years’ worth of stock data from the National Stock Exchange for the INFOSYS stock. The stock data was provided from Kaggle, and for development purposes, the data was hosted publicly on Amazon Simple Storage Service (S3).

In both cases, the stock data was split into training and test sets and then the training data was passed through the model. Once the model was fit to the data, then the model was evaluated on their accuracy in predicting the stocks’ close price. The models were evaluated on their Root Mean Square Error (RMSE) as in (1), which is often used as an accuracy evaluation metric for regression problems.

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{Y}_i - Y_i)^2}{n}} \quad (1)$$

The RMSE values were calculated using both the training and test data.

A. *Basic RNN Log of Experiments*

For the basic RNN model, no libraries were used to perform the implementation of the recurrent neural network and a PySpark notebook was coding language for development. However, preprocessing for scaling the close stock price to be between zero and one was done using the SciKit Learn library MinMaxScaler [3]. The decision to use the SciKit Learn preprocessing scaler instead of the Apache Spark MLLib MinMaxScaler was based on the fact that the pyspark.ml.feature implementation for the MinMaxScaler does not provide an inverse transformation option that would be needed to inverse transform the final predictions easily [4]. Additionally, since research on recurrent neural network prior to starting development mentioned a potential issue of exploding gradients, gradient clipping was implemented to try and prevent that issue from occurring. In addition to some of the other parameters that were allowed to be defined for the recurrent neural network, I offered the ability to dictate which activation function was to be used, either sigmoid or tanh.

Once the recurrent neural network program was fully developed, experimentation was done to try and find the best parameters to train the basic implementation of the RNN and find the parameters which best optimized model and resulted in a low Root Mean Square Error (RMSE) on the test dataset. I spent roughly about a week on the implementation of the basic RNN and trying to find the optimal parameters to be able to find the lowest root mean square error value on the test set. The results of the experiments and any parameters I used can be found in Table 1.

TABLE I. LOG OF EXPERIMENTS USING THE BASIC RECURRENT NEURAL NETWORK

Experiment Number	Parameters	Results

1	learningRate = 0.001
	numEpochs = 30
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -10
	maxClip = 10
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 90:10
		Training RMSE = 27678.05777268728
		Test RMSE = 28165.829263181065

2	learningRate = 0.001
	numEpochs = 30
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 90:10
		Training RMSE = 760.2829782436548
		Test RMSE = 50.30948552045907

Identify applicable funding agency here. If none, delete this text box.

Experiment Number	Parameters	Results

3	learningRate = 0.001
	numEpochs = 30
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 769.5394583723573
		Test RMSE = 34.35453299784472

4	learningRate = 0.001
	numEpochs = 30
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 70:30
		Training RMSE = 854.1236676543032
		Test RMSE = 58.044899066948794

5	learningRate = 0.0001
	numEpochs = 30
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 24333.331481584737
		Test RMSE = 24889.957613722505

6	learningRate = 0.0025
	numEpochs = 30
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 15448.516239452483
		Test RMSE = 14876.531739050923

7	learningRate = 0.001
	numEpochs = 50
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 769.5394583723576
		Test RMSE = 34.35453299784462

8	learningRate = 0.001
	numEpochs = 50
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -125
	maxClip = 125
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 769.5394583723573
		Test RMSE = 34.35453299784472

9	learningRate = 0.001
	numEpochs = 50
	hiddenDimensions = 100
	backPropTruncate = 10
	minClip = -100
	maxClip = 100
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 769.5394583723576
		Test RMSE = 34.35453299784462

10	learningRate = 0.001
	numEpochs = 50
	hiddenDimensions = 100
	backPropTruncate = 0
	minClip = -100
	maxClip = 100
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 769.5394583723569
		Test RMSE = 34.35453299784509

11	learningRate = 0.001
	numEpochs = 15
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 10;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 769.5394583723303
		Test RMSE = 34.35453299786014

12	learningRate = 0.001
	numEpochs = 15
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 20;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 2358.3090901623336
		Test RMSE = 1750.7086890368118

13	learningRate = 0.001
	numEpochs = 30
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 20;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 656.3837142280333
		Test RMSE = 917.4583789816425

14	trainSplit = 0.8
	learningRate = 0.001
	numEpochs = 100
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 20;
	activationFunction = "sigmoid"
		Train/Test Split = 80:20
		Training RMSE = 722.9801770040374
		Test RMSE = 69.63992042473363

15	learningRate = 0.001
	numEpochs = 100
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 20;
	activationFunction = "sigmoid"
		Train/Test Split = 90:10
		Training RMSE = 724.7960987679251
		Test RMSE = 38.43644861094549

16	learningRate = 0.001
	numEpochs = 100
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -10
	maxClip = 10
	timeSteps = 20;
	activationFunction = "sigmoid"
		Train/Test Split = 90:10
		Training RMSE = 704.7217088923082
		Test RMSE = 10.589057733659732

17	learningRate = 0.001
	numEpochs = 100
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -100
	maxClip = 100
	timeSteps = 20;
	activationFunction = "sigmoid"
		Train/Test Split = 90:10
		Training RMSE = 710.1029949764699
		Test RMSE = 17.411932686041567

18	learningRate = 0.001
	numEpochs = 50
	hiddenDimensions = 100
	backPropTruncate = 10
		Train/Test Split = 80:20
		Training RMSE = 765.2807053683649

Experiment Number	Parameters	Results

	minClip = -10
	maxClip = 10
	timeSteps = 10;
	activationFunction = "tanh"
		Test RMSE = 37.191873440361746

19	learningRate = 0.001
	numEpochs = 100
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -10
	maxClip = 10
	timeSteps = 15;
	activationFunction = "tanh"
		Train/Test Split = 90:10
		Training RMSE = 750.6144171498265
		Test RMSE = 57.15831523059498

20	trainSplit = 0.8
	learningRate = 0.001
	numEpochs = 50
	hiddenDimensions = 100
	backPropTruncate = 10
	minClip = -10
	maxClip = 10
	timeSteps = 15;
	activationFunction = "tanh"
		Train/Test Split = 80:20
		Training RMSE = 689.7072398101923
		Test RMSE = 131.5089111877802

21	learningRate = 0.001
	numEpochs = 50
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -10
	maxClip = 10
	timeSteps = 15;
	activationFunction = "tanh"
		Train/Test Split = 80:20
		Training RMSE = 954.3528228610796
		Test RMSE = 1343.6684236445528

22	learningRate = 0.001
	numEpochs = 100
	hiddenDimensions = 100
	backPropTruncate = 10
	minClip = -10
	maxClip = 10
	timeSteps = 15;
	activationFunction = "tanh"
		Train/Test Split = 80:20
		Training RMSE = 760.9814149012685
		Test RMSE = 33.230681820957535

23	learningRate = 0.001
	numEpochs = 100
	hiddenDimensions = 100
	backPropTruncate = 5
	minClip = -10
	maxClip = 10
	timeSteps = 15;
	activationFunction = "tanh"
		Train/Test Split = 80:20
		Training RMSE = 760.9814149012706
		Test RMSE = 33.230681820955596

The results of the implementation of the RNN were very interesting. To start with the training RMSE was often higher than the test RMSE. I did some further digging into potential reasons this might occur, and some potential reasons include that the when the training set is large and the test set is small this may happen. However, looking at the test RMSE values, it appears that although the training RMSE was not high, the model was still able to well generalize the data and fairly accurately predict the stock prediction close price.

Looking at the loss values calculated per epoch during the model fitting, both the training and test error values were decreasing for each epoch which is a good sign that the model was learning based on the input data passed in. The improvement on loss was visible during every experiment.

Something to note is that at both ten and fifteen time steps and with a variety of parameter differences, the resulting test RMSE seemed to consistently only be able to get as low as around thirty three or thirty four. This suggests that these were likely some of the best parameters identified for the model.

Another observation to note, I noticed that the results of the RNN implementation were quite volatile. The range of the test RMSE values is quite large throughout the experiments I did. I believe this shows how important parameter tuning is when training models in order to get the most accurate model and true predictions. Even slight changes in the parameters being used could drastically change the results of the model. Additionally, while watching the error loss, it was possible to tell that the loss had not yet converged and needed to be trained further in some of these cases. Nevertheless, I did not notice that in general using either sigmoid or tanh activation function did not make a big difference in the final root mean square error calculated on the test set.

B. LSTM Log of Experiments

For the long short-term memory recurrent neural network, the keras library was used to help implement the LSTM RNN. Although, this was based on PySpark as well, preprocessing for scaling the feature vectors to be between zero and one was done using the SciKit Learn library MinMaxScaler similar to the basic RNN implementation since there is no inverse transform method available for the MLLib implementation. This was done as not all of the features were on the same scale, and normalization and standardization of features is one of the most important parts of preprocessing in order to produce an accurate machine learning model.

The keras library offered many parameters that could be defined as well as some predefined common functions used in machine learning that could be utilized. For example, keras has model optimizers built in that could be used, including: SGD, RMSprop, Adam, Adadelta, Adagrad, Adamax, Nadam, and Ftrl. This allowed me to test some model optimizers that I was not familiar with prior to starting this project.t Additionally, similar to my implementation of the basic recurrent neural network, it is possible to define the number of time steps, number of hidden dimensions, as well as the number of epochs to traverse when training the model and calculating loss.

Similar explorations as to what was done in the previous portion were repeated to try and find the optimal parameters for the LSTM RNN to best predict the close stock prices on the test set. Like mentioned previously, the results outputted the root mean square error on the training and test sets. The results of that experimentation can be found in Table 2.

TABLE II. LOG OF EXPERIMENTS USING THE LONG SHORT-TERM RECURRENT NEURAL NETWORK

Experiment Number	Parameters	Results

1	timeSteps = 10
	lstmUnits = 128
	dropoutRate = 0.2
	modelOptimizer = 'adam'
	modelLoss ='mean_squared_error'
	numEpochs = 30
		Train/Test Split = 80:20
		Training RMSE = 11.564867375920242
		Test RMSE = 43.81211427380087

2	timeSteps = 10
	lstmUnits = 128
	dropoutRate = 0.2
	modelOptimizer = 'adam'
		Train/Test Split = 90:10
		Training RMSE = 6.8043000213423435

Experiment Number	Parameters	Results
	modelLoss ='mean_squared_error' numEpochs = 30	Test RMSE = 23.190838617212208
3	timeSteps = 20 lstmUnits = 128 dropoutRate = 0.2 modelOptimizer = 'adam' modelLoss ='mean_squared_error' numEpochs = 30	Train/Test Split = 90:10 Training RMSE = 6.503753393608997 Test RMSE = 13.523329010046222
4	timeSteps = 20 lstmUnits = 128 dropoutRate = 0.5 modelOptimizer = 'adam' modelLoss ='mean_squared_error' numEpochs = 30	Train/Test Split = 90:10 Training RMSE = 6.064580817152961 Test RMSE = 11.072506999573259
5	timeSteps = 20 lstmUnits = 128 dropoutRate = 0.5 modelOptimizer = 'adam' modelLoss ='mean_squared_error' numEpochs = 50	Train/Test Split = 90:10 Training RMSE = 6.0874023505948465 Test RMSE = 10.979132818051669
6	timeSteps = 20 lstmUnits = 128 dropoutRate = 0.5 modelOptimizer = 'SGD' modelLoss ='mean_squared_error' numEpochs = 50	Train/Test Split = 90:10 Training RMSE = 7.809335000204635 Test RMSE = 15.278006711456293
7	timeSteps = 20 lstmUnits = 128 dropoutRate = 0.5 modelOptimizer = ‘RMSprop’ modelLoss ='mean_squared_error' numEpochs = 50	Train/Test Split = 90:10 Training RMSE = 6.514397901095763 Test RMSE = 10.83643008358828
8	timeSteps = 20 lstmUnits = 128 dropoutRate = 0.5 modelOptimizer = ‘Adadelta’ modelLoss ='mean_squared_error' numEpochs = 50	Train/Test Split = 90:10 Training RMSE = 45.237020450673484 Test RMSE = 42.70266578077335
9	timeSteps = 20 lstmUnits = 128 dropoutRate = 0.5 modelOptimizer = ‘Adagrad’ modelLoss ='mean_squared_error' numEpochs = 50	Train/Test Split = 90:10 Training RMSE = 11.704579014661418 Test RMSE = 11.931818424820266
10	timeSteps = 20 lstmUnits = 128 dropoutRate = 0.5 modelOptimizer = ‘Adamax’ modelLoss ='mean_squared_error' numEpochs = 50	Train/Test Split = 90:10 Training RMSE = 7.439768832980003 Test RMSE = 15.00737381423345
11	timeSteps = 20 lstmUnits = 128 dropoutRate = 0.5 modelOptimizer = ‘Adam’ modelLoss ='mean_squared_error' numEpochs = 100	Train/Test Split = 90:10 Training RMSE = 5.183084196297607 Test RMSE = 10.53959565856163
12	timeSteps = 20 lstmUnits = 256 dropoutRate = 0.5 modelOptimizer = ‘Adam’ modelLoss ='mean_squared_error' numEpochs = 100	Train/Test Split = 90:10 Training RMSE = 5.053555013838693 Test RMSE = 11.614997524091082
13	timeSteps = 20 lstmUnits = 100 dropoutRate = 0.5 modelOptimizer = ‘Adam’ modelLoss ='mean_squared_error' numEpochs = 100	Train/Test Split = 90:10 Training RMSE = 5.733013490017255 Test RMSE = 10.662242360775737
14	timeSteps = 20 lstmUnits = 100 dropoutRate = 0.5 modelOptimizer = ‘Adam’ modelLoss =’mean_squared_logarithmic _error’ numEpochs = 100	Train/Test Split = 90:10 Training RMSE = 5.500323393327395 Test RMSE = 12.439359123925406
15	timeSteps = 20 lstmUnits = 100 dropoutRate = 0.4 modelOptimizer = ‘Adam’ modelLoss ='mean_squared_error' numEpochs = 100	Train/Test Split = 90:10 Training RMSE = 5.80571463315959 Test RMSE = 10.339394350893345

I tried to focus more of my time optimizing the basic RNN implementation as that required not using the libraries and was not as obvious how to enhance. However, I was still able to do analysis on the LSTM model utilizing the keras library which took in similar parameters as well.

The LSTM was able to get decent test RSME values, with the lowest being 10.33939435. The LSTM model performed the best having a larger number of time steps (twenty versus ten in experimenting). This demonstrates that long short-term models are able to accurately make predictions on time series data over long sequences and is able to not only learn, but also remember important information based on previous data passed through the model.

C. *Comparison and Analysis of Results*

The results of the two models provide valuable insight into the LSTM model and its’ effectiveness on time series prediction. To more easily show the comparison between the two models, Fig. 3. shows a scatterplot for the fifteen lowest root means square error values for the basic RNN and LSTM RNN implementation on different amounts of time steps.

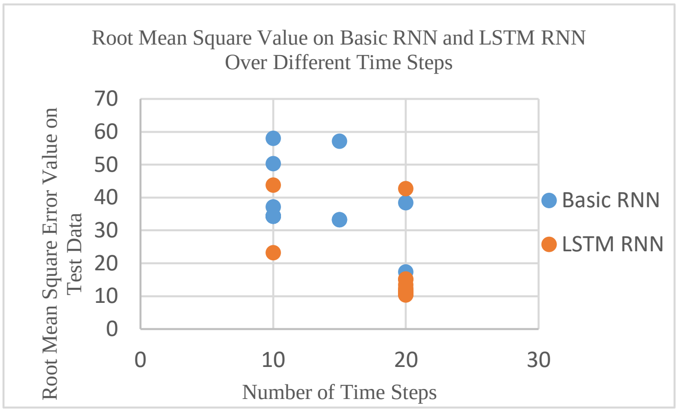

Fig. 3. Root Mean Square Value on Basic RNN and LSTM RNN Over Different Time Steps (Figure 3 represents a scatter plot comparison for the 15 lowest RMSE values on test data sets for the basic recurrent neural network and long short-term memory implementation)

One of the downfalls of the basic recurrent neural network model is that it is not always able to accurately remember and retain information on what it is important from earlier time steps With the results of the training set, it is evident that the long short-term model was able to more accurately predict the next close stock price in comparison to the basic recurrent neural network. Overall, irrelevant of the time step, more of the long short-term recurrent neural network root mean square error values were in a lower range that that of the basic recurrent neural network model.

Also one of the features of the keras LSTM model that I think greatly helped the performance and learning of the model was having an applicable dropout rate. Dropout is when a certain percentage or proportion of neural network cells are removed from the network. This is used for regularization in neural networks. Dropout allows the model to learn better and more robust and also decreases the chance that the model will over fit to the training data. The experiments backed this as well, and keeping all other parameters the same and increasing the dropout rate to a certain extent helped the model perform better.

Additionally, in comparing the results between the implementation of recurrent neural from scratch and the one utilizing the keras library, the LSTM model using keras did not seem susceptible to the same issues with having very high training RMSE. This was likely explained as the LSTM was able to remember long term patterns in the data and performed better on both the training and test data in comparison to the basic RNN model.

Overall both models were able to perform relatively well and predict with reasonable test RMSEs the next stock close price. However, the long short-term recurrent neural network model did prove to be more accurate in predictions.

IV. CONCLUSION AND FUTURE WORK

In conclusion, recurrent neural networks are geared towards being able to fit models on data where previous data is important or the ordering of data is important. RNNs are able to learn and remember what is important from previous steps in order to get a more accurate model. In general, recurrent neural networks are a fairly complex model which needs to be trained and tuned appropriately to get solid computations. With the proper parameter tuning, they can prove to be quite accurate and successful on time series data predictions.

For this paper, analysis was done on time series data prediction, specifically predicting the close stock price of a stock. Between the implementation from scratch and the LSTM utilizing the keras libraries, for future work, I think I would expand any future studies utilizing existing libraries. Libraries already offer the implementation and provide built in flexibility and ease in utilizing models, and more time could be spent trying to find the optimal parameters and finding the best model for the data. However, implementation from scratch did help provide a deeper understanding of how recurrent neural networks work and the mechanisms behind it, which I appreciated. Seeing the higher accuracy of results of the LSTM model on predicting the stock close prices showed how models are evolving to more accurately fit time series data.

It is evident the importance of recurrent neural network models and their ability to make predictions on time series data. The long short-term memory cell in particular shows a lot of promise in being able to more accurately remember information from previous data and perform future predictions. LSTMs in general can serve a lot of purpose outside of stock prices, as it works well with sequences of data. In addition to predicting stock prices, other common uses of the long short-term model can be for other time series predictions such as weather data , which includes tornado and hurricane data. Furthermore, other uses for using recurrent neural networks have been revolved around words and sentences, where sequence matters, such as accurate language translation, predicting next words in a sentence, or being able to generate possible suggestions based on input data.

A. *Future Work*

As mentioned in relation to testing done on the basic recurrent neural network, some inconsistencies with some test and training RMSE values could potentially be attributed to not having a large enough dataset. In the future if I were to try something similar, I would try a much larger dataset, maybe decades worth of data, and see if and how that impacted the results and accuracy on prediction.

Doing further research into recurrent neural networks in general has led me to some topics of interest and other possible application or models that piqued by interest that I would like to expand upon. For example, I found a research paper that utilized a clock-work recurrent neural network CW RNN) that outperformed the long short-term model when used to predict stream flow [5]. It would be interesting to try and compare the time series stock prediction using the CW RNN and compare the performance to the ones in this paper using the LSTM model as further extension of this research. Unfortunately, the CW RNN cell is not offered by keras in their sequential model layers currently, but it would likely be possible to build on top of the existing keras functionality to create a clock-work recurrent neural network in order to analyze its performance.

REFERENCES

[1] F. Deloche, Recurrent neural network unfold. 2017.

[2] G. Chevalier, The LSTM cell. 2018.

[3] "sklearn.preprocessing.MinMaxScaler — scikit-learn 0.23.2 documentation", Scikit-learn.org, 2020. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html. [Accessed: 09- Aug- 2020].

[4] "pyspark.ml package — PySpark 3.0.0 documentation", Spark.apache.org, 2020. [Online]. Available: https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.feature.MinMaxScaler. [Accessed: 09- Aug- 2020].

[5] Z. Mhammedi, A. Hellicar, A. Rahman, K. Kasfi and P. Smethurst, "Recurrent Neural Networks for One Day Ahead Prediction of Stream Flow", 2016