Time Series Stock Price Prediction with Recurrent Neural Networks

***Abstract*—This project presents a comprehensive investigation into the application of vanilla Recurrent Neural Networks (RNNs) for stock price prediction. In an era where financial forecasting plays a pivotal role in investment decisions, the study explores the efficacy of vanilla RNNs in capturing the temporal dependencies and patterns inherent in Texas Instrument's historical stock price data.**

Keywords—RNN, Stock Price, Prediction, Texas Instruments

I. INTRODUCTION

Stock price prediction remains a challenging yet essential task in the domain of financial forecasting, with far-reaching implications for investors, traders, and financial analysts. In recent years, the advent of deep learning techniques, particularly Recurrent Neural Networks (RNNs), has sparked significant interest in their application to this problem domain. This report delves into the exploration and analysis of employing RNNs, specifically vanilla RNNs, for stock price prediction.

The primary focus of this project is to investigate the efficacy of vanilla RNNs in capturing the temporal dependencies and patterns inherent in 5 years of histroical stock price data of Texas Instruments.

Furthermore, we conduct a thorough empirical analysis of vanilla RNN's performance in stock price prediction tasks. Through experiments conducted on real-world stock price datasets, we assess the predictive capabilities of vanilla RNNs, evaluate their accuracy, and compare their performance against the model applied with Keras Libraries SimpleRNN.

In our analysis, our team conducted a comparative evaluation with TensorFlow's RNN model, to increase prediction accuracy. The comparative analysis not only sheds light on the relative strengths and weaknesses of both approaches but also serves as a validation of our custom vanilla RNN implementation against an established framework like TensorFlow.

II. CONCEPT AND IMPLEMENTATION OF RNN

A. THEOReTICAL concept OF THE RNN MODEL

A neural network just takes in data as input variables (X) and gives output variables (Ycap). However, in a standard neural network the out of the input sequence does not matter, which makes a standard neural network extremely difficulty to use in time series prediction. To remedy this, we use RNN.

For the scope of this project, we are focusing on the many to one RNN model with a focus on mathematical input data as we wish to predict Stock Prices. In a standard feed forward neural network, the output of one hidden layer is fed to the subsequent hidden layer. The RNN however consists of a sequential input, sequential output, multiple timesteps and multiple hidden layers. Here each hidden layer value is calculated from the input values as well as previous time step values and weights (w) at the hidden layers from the same time steps.

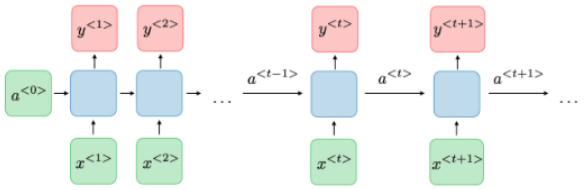

Fig II.A.1 A visual representation of an RNN

Fig II.A.1 represents a standard RNN with each hidden layer having data from the previous timestep as well as from the input. For each time-step t the activation $a^{<t>}$ and the output $y^{<t>}$ are expresses as the follows :

$$a^{<t>} = g_1\left(W_{aa}\, a^{<t-1>} + W_{ax}\, x^{<t>} + b_a\right)$$

$$a^{<t>} = g_2\left(W_{ya}\, y^{<t>} + b_y\right)$$

Where W_{ax}, W_{aa}, W_{ya}, b_a, b_y are coefficients that are shared temporarily and g_1, g_2 are activation functions.

The loss function of all time steps is defined based on the loss at every time step as follows -

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}(\hat{y}^{<t>}, y^{<t>})$$

The backpropagation is also done at each point in time and can be expressed as

- Total error is the sum of each error at time steps t

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W}$$

- Hardcore chain rule application:

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

RNNs often encounter the phenomenon of vanishing / exploding gradient, as it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers. To cope with this, we use the technique called gradient clipping where the numeric max value of the gradient is established hence capping the exponential increase/decrease of the gradient

While RNNs have several advantages such as being able to process input of any length while ensuring the model size doesn't scale with the input and considering historic information. They are often slow in computations, have difficulty accessing information form a long time ago and cannot consider any future input

B. Implementation of the RNN Model

The model contains a comprehensive framework for constructing, training, and utilizing RNN models, providing flexibility in network architecture and activation functions.

During the training phase, the class 'SimpleRNN' performs forward and backward passes through the network using the 'train' method. This method iterates over the training data for a specified number of epochs, during which the model

adjusts its parameters to minimize a defined loss function. The forward pass computes hidden states and output predictions, while the backward pass calculates gradients and updates the model weights through

To initialize our model, we start by defining essential variables, including the training and testing datasets, as well as the sequence length. The dataset is plit into training and testing subsets, with 20% allocated for testing and 80% for training. The sequence length, denoted as 50 in our case, signifies the temporal input window used to construct the time series data.

The model architecture is defined by specifying parameters such as input size, hidden size, output size, and activation function. These parameters configure the structure of the RNN. We initialize the weight parameters and provide flexibility in choosing the activation function, allowing options between 'tanh', 'sigmoid', and 'ReLU'. In our case, we found that the tanh activation function outperformed others in terms of minimizing the loss. Therefore, we opted to use the tanh activation function in our model.

In back propagation, we applied Backpropagation Through Time (BPTT) to compute gradients for the hidden-to-hidden weights ('dWhh') and input-to-hidden weights ('dWxh'). This involves recursively applying the chain rule to propagate gradients from the output layer back through the hidden layers to the input layer by iteratively computing these gradients, we update the weights of the RNN to minimize the loss function, thus optimizing the model's performance.

The clip function plays a crucial role in mitigating the issue of exploding gradients, a common challenge encountered during the training of deep neural networks. By constraining the values of a matrix within predefined upper and lower bounds, the clip function ensures numerical stability during gradient descent process.

Finally, the predict method in the code enables the generation of predictions using the trained RNN model. Given input sequences, this function computes corresponding output predictions, thereby facilitating the application of the model to real-world data for tasks such as sequence forecasting or classification.

C. Data Pre-processing

The time series data is raw and is pulled from the yahoo finance website. The collected data has the following column: Date, Open, High, Low, Close, Adj Close, Volume. There are 5 years of stock data and about 1260 rows. For simplicity, we will only focus on predicting the opening stock price i.e., the 'Open' for each day. Therefore, we remove all the other columns and visualize the opening stock price.

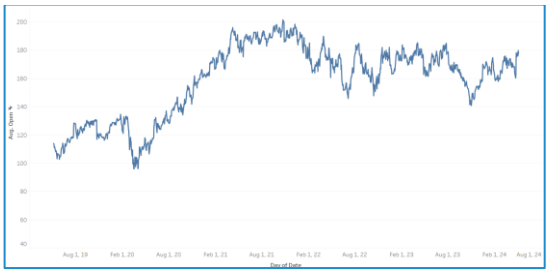

Fig I.C 5 year opening stock price trend of Texas Instruments

From the trend, the opening price has seasonality. This makes the time series nonstationary. A stationary time series has constant statistical properties like constant variance and has no seasonal trends in it.

To make a time series stationary, 'differencing' is a method that we have used here. For this time series, a seasonal difference would be sufficient. We have taken a seasonal duration of 30 days (about 4 and a half weeks). The 30-day period is selected based on the observation from the seasonal decomposition chart. As shown below,

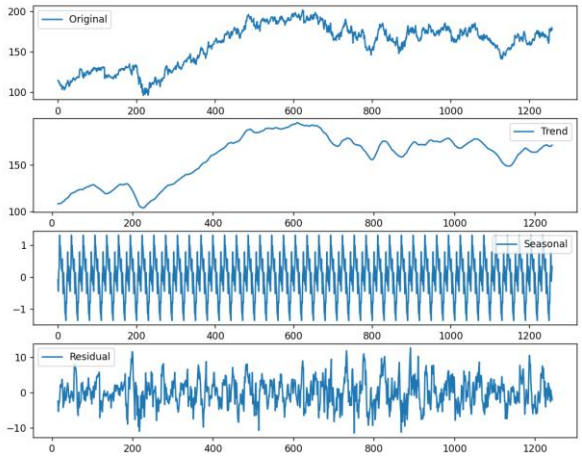

Fig II.C Seasonal Decomposition of time series for 30-day season.

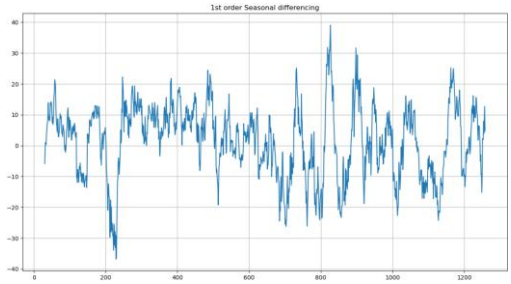

Fig III.C Time series after applying 1st order seasonal differencing.

Results of KPSS Test:	
Test Statistic	0.286812
p-value	0.100000
Lags Used	20.000000
Critical Value (10%)	0.347000
Critical Value (5%)	0.463000
Critical Value (2.5%)	0.574000
Critical Value (1%)	0.739000

Fig IV.C Result of KPSS hypothesis testing.

Once the difference is done, we conduct a KPSS hypothesis testing to ensure data is stationary. The null hypothesis of KPSS is "The time series is stationary" and the alternative hypothesis is that the "Time is not stationary". We are considering a 95% confidence interval and based on that the p-value is greater than 0.05 significance level. This suggests that we don't reject the null hypothesis.

We will supply our model with this seasonally adjusted data. Once the model learns to predict this time series, we can retrieve the original value of the opening stock price. Removing seasonality reveals hidden patterns in a time series that helps the model to learn these patterns associated with unpredictable economic variations, market situations etc.

D. Results and analysis

We ran this model against pre-processed data with a wide variety of parameters so see the possible results our model will output. We got various model outputs that we have included below -

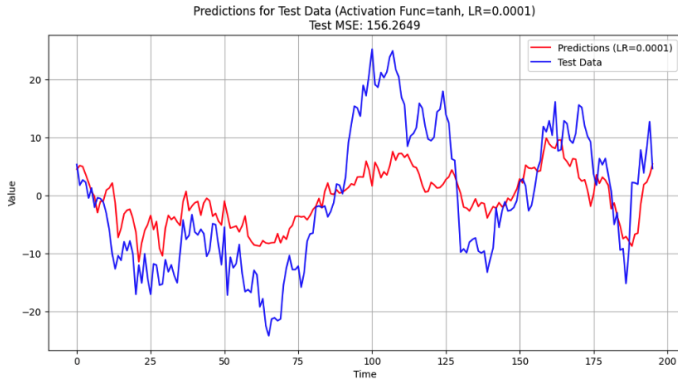

Fig II.D.1

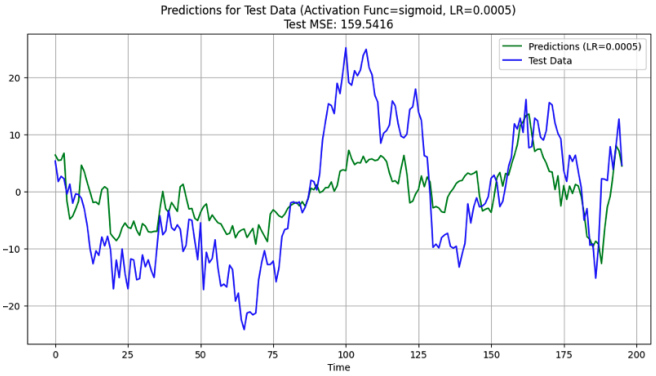

Fig II.D.2

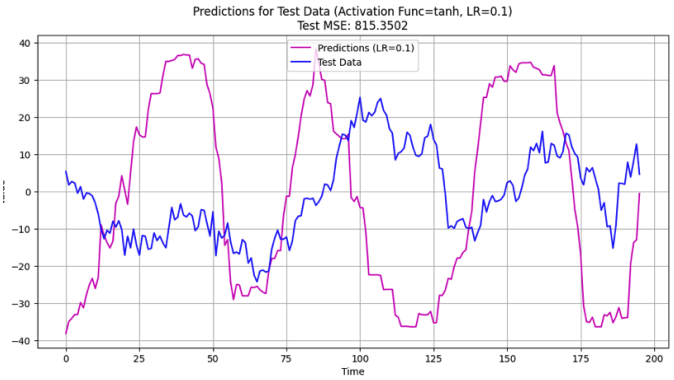

Fig II.D.3

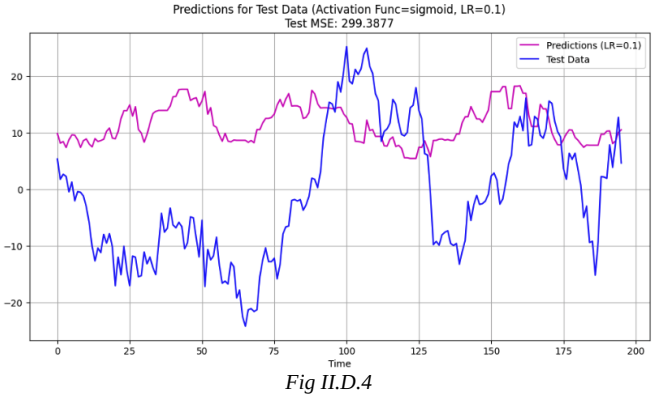

Fig II.D.4

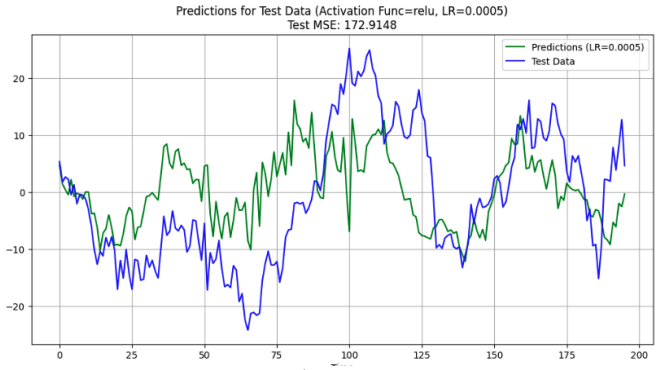

Fig II.D.5

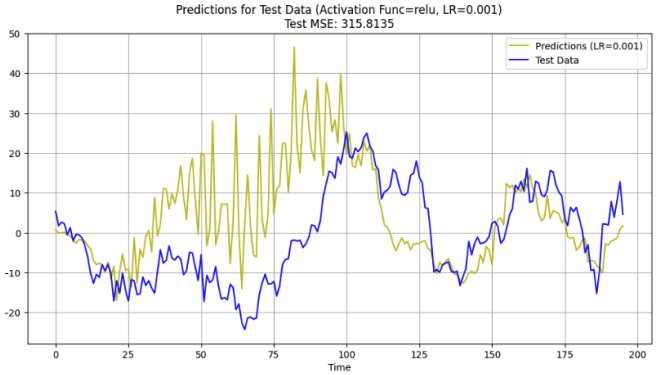

Fig II.D.6

Fig II.D.1, Fig II.D.2., Fig II.D.,3 Fig II.D.4, Fig II.D.5, Fig II.D.6 are output graphs from the algorithm

From the graphs we can see that the model got its best results when using tanh activation function and a learning rate, but we can observe that the model does weel when using the tanh and sigmoid activation functions with a small learning rate, as we increase the learning rate, we see that the model output deteriorates drastically. Similarly, the relu activation function does not give us satisfactory results, rather we have results with a higher loss.

In all three models we also notice the issue of exploding / vanishing gradient and see that our algorithm has to implement gradient clipping but i believe this is why our model is not able to perform well in certain time periods as when there is a sudden spike / drop in training data our model is not able to account for it because of the clipping.

We believe that using a GRU or LTSM model will significantly improve our model and account for these deficiencies

III. Model Comparison

To compare the performance of the RNN model built using NumPy library we have also created an RNN model using Keras' SimpleRNN.

	Training MSE (lowest)	Testing MSE (lowest)
RNN using NumPy	162.48	186.83
RNN using Keras	73.05	75.01

The results of library based (Keras) RNN is approximately 2 times better than NumPy model. There are several factors like not including bias weights, using stochastic gradient instead of Adam optimizer that is available out of the box in Keras library. Furthermore, this NumPy based RNN is the simplified version of Keras RNN model.

IV. Conclusion

Stock market prediction has been of great interest to many stakeholders like investors, traders, and analysts. Recent developments have increasingly used neural networks as their basis for forecasting. Among them RNN has gained a wider acceptance. We explored the application of vanilla RNN for stock prices prediction using real world dataset. Ensuring thorough data preprocessing for better performance.

The RNN model developed using NumPy faced two main challenges. One, vanishing gradient and second exploding gradient. Both issues have been addressed using techniques like gradient clipping and bptt truncate method.

Results suggest that RNN model performed well with tanh activation function along with a combination of lower learning rate. However, the performance of Keras based RNN model was 2 times better than the NumPy based model because advanced features like biases, special optimizers like Adam, improves performance in terms of MSE a lot.

Finally, the prospect of time series forecast does not end with RNN. Advanced variations of RNN like GRU (Gated Recurrent Unit) and LSTM (Long Short-Term Memory) have greater potential at predicting with higher accuracy over long sequences and this paves the way for future research in time series prediction using these techniques.

References

[1] https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

[2] https://github.com/VikParuchuri/zero_to_gpt/blob/master/explanations/rnn.ipynb