Deep Learning in Finance: RNNs for Stock Price Forecasting

Paridhi Awadheshpratap Singh
Student ID: a1865487
Email: a1865487@adelaide.edu.au

ABSTRACT

This study explores the application of Recurrent Neural Networks (RNNs) for stock price prediction, using a decade-long daily dataset sourced from Yahoo Finance [1]. The research focuses on comparing various RNN models, including Vanilla RNNs, LSTMs, and GRUs, to identify the most effective approach. Traditional methods for stock price prediction often struggle with intricate patterns in financial time series data. Our comparative analysis evaluates the strengths and weaknesses of different RNN variants. The dataset undergoes preprocessing, and various RNN structures are tested, with predictive performance assessed using common metrics like Root Mean Squared Error (RMSE). The research code and details are shared on GitHub for transparency and reproducibility. In conclusion, this research contributes practical insights to the application of RNNs for accurate stock price predictions, utilizing real historical data and comparing different models.

I. Introduction

This research addresses the challenge of forecasting financial markets, particularly stock prices, using machine learning techniques and artificial neural networks. Financial markets involve transactions of commodities like stocks, bonds, and precious metals between buyers and sellers.

Efficient Market Hypothesis, proposed by Burton G. Malkiel [2], suggests that predicting financial markets is unrealistic as price changes are unpredictable. However, there are opposing views, and recent research indicates that financial markets are predictable to some extent based on past experiences and undiscounted correlations among economic events.

Machine learning methods, including neural networks, have gained prominence in financial market prediction. Neural networks, in particular, are known for their noise tolerance and flexibility compared to traditional statistical models. Long Short-Term Memory (LSTM), introduced by Hochreite and Schmidhuber [3], is a recurrent neural network designed to handle long time lags and has been successfully applied in various domains. We have also employed other neural networks namely GRU, Stacked LSTM and bidirectional LSTM for the analysis.

The research aims to assess the feasibility of Recurrent Neural Networks (RNNs) in stock market forecasting by testing different configurations. Additionally, the study explores potential improvements for future research based on prediction results generated by various models [4].

II. Method Description

Recurrent Neural Networks (RNNs) are a type of advanced algorithms that can remember previous inputs when dealing with a large set of sequential data. Unlike other neural networks, RNNs have an internal memory, allowing them to perform the same function for every input while considering past computations. To achieve this, the output is copied and sent back into the network, influencing decisions based on both current and past inputs.

In traditional neural networks, inputs are independent of each other, but RNNs create connections between inputs, making them related. The loops within RNNs might seem mysterious, but in essence, they can be seen as multiple copies of the same network, each passing information to the next one. This interconnected approach highlights the relationship between different inputs, making RNNs effective for tasks involving sequential data [5].

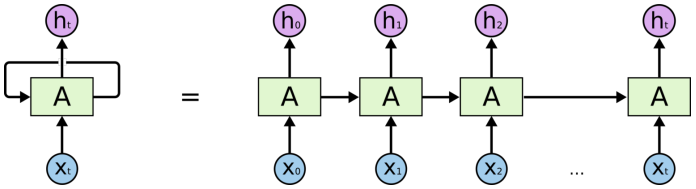

Fig. 1. RNN Architecture [5]

The figure above illustrates the architecture of a Recurrent Neural Network. In RNNs, the way information moves from input to output differs from other neural networks. While deep neural networks typically have different weight matrices for each part, RNNs keep the same weight across the network. It computes a hidden state (Hi) based on each input (Xi) using these equations:

$$h_i = \sigma(UX_i + Wh_{i-1} + B) \tag{1}$$

$$Y_i = O(Vh_i + C) \tag{2}$$

In these equations:

- 'h_i' represents the hidden state calculated using the input and previous hidden state.
- 'Y_i' is the output derived from 'h_i'.
- 'σ' is an activation function.
- 'U', 'W', 'V', 'B', and 'C' are parameters shared across time steps.

The state matrix 'S' stores the network's state at each time step 'i'. This design helps RNNs retain information over time, critical for sequential data tasks.

The Recurrent Neural Network (RNN) is composed of distinct activation function units, each tied to a specific time step. Within each unit lies an internal state known as the hidden state, embodying the network's past knowledge at a given moment. This hidden state evolves continuously, reflecting the network's evolving understanding of past information. It's updated iteratively through a recurrence relation: The current state (h_t) is computed as:

$$h_t = f(h_{t-1}, x_t) \tag{3}$$

Where:

- h_t signifies the current state.
- h_{t-1} denotes the previous state.
- x_t represents the input state.

Applying the activation function (tanh) involves:

$$h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t) \tag{4}$$

Where:

- W_{hh} is the weight at the recurrent neuron.
- W_{xh} stands for the weight at the input neuron.

The output calculation is formulated as:

$$y(t) = W_{hy} \cdot h(t) \tag{5}$$

Here:

- $y(t)$ denotes the output.
- W_{hy} represents the weight at the output layer.

These parameters undergo updates using Backpropagation. Given the sequential nature of RNN data, an adapted form of backpropagation, known as Backpropagation Through Time, is employed here to facilitate learning over sequences.

There exist four categories of RNNs, categorized based on the quantity of inputs and outputs within the network.

- **One to One:** This RNN type functions similarly to a conventional neural network, often referred to as a Vanilla Neural Network. It involves a singular input and produces a lone output.
- **One to Many:** Within this RNN category, a single input is associated with multiple outputs. A prominent application of this network type is observed in image captioning, where an image input leads to the generation of a sentence comprising multiple words.
- **Many to One:** In this RNN variant, numerous inputs are provided to the network at different stages, culminating in the generation of a solitary output. This type of network is commonly employed in sentiment analysis, where multiple words serve as input to predict the sentiment of the entire sentence.
- **Many to Many:** This neural network configuration involves multiple inputs and outputs corresponding to a given problem. A notable example is language translation, where a series of words from one language is inputted, and the network predicts a corresponding series of words in another language as output [6].

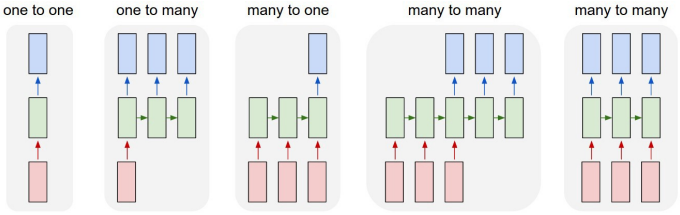

Fig. 2. Different types of RNNs [5]

A. Backpropagation Through Time (BPTT)

When training an RNN, we start by figuring out how off our predictions are from the actual results. This is done using a loss function (L). We then work to minimize this difference through forward and backward passes. The whole idea of backpropagation through time can be simplified in the following way: For each step in time, we take an input, process it through a hidden layer, and calculate our best guess for the label. During this, we also calculate how far off we are using the loss function. This process of calculating the loss is the forward pass. After that, we do the backward pass, where we figure out how we should adjust things to get better. Training an RNN is not straightforward because we're not only adjusting things as we go through different layers but also through different moments in time. So, in each time step, we have to add up all the past contributions to the current one. The challenge during training is managing how much importance we give to the hidden state [7].

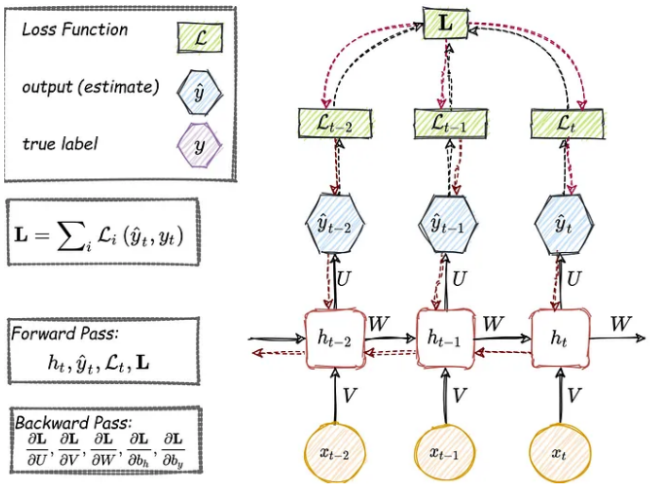

Fig. 3. Backpropagation Through Time (BPTT) [7]

When dealing with time-series data during backpropagation, we often encounter two issues: vanishing and exploding gradients.

- Vanishing Gradient: This occurs when a term gets very small very quickly, making it hard for the model to learn long-term patterns. It's like the information fades away. We call this the vanishing gradient problem.

- Exploding Gradient: On the flip side, there's the exploding gradient problem. Here, a term shoots up to infinity super fast, causing the process to become unstable, and the result becomes undefined (NaN).

To tackle these problems, we can use a method called Truncated Backpropagation Through Time (Truncated BPTT). It's like looking at our data through a moving window during training. Instead of doing a full sweep through the entire sequence, we focus on smaller chunks, making things faster and less complex. However, there's a trade-off. We might miss learning dependencies longer than the chunk length. Also, it's a bit tricky to spot vanishing gradients just by looking at the learning curve. For the vanishing gradient problem, there are

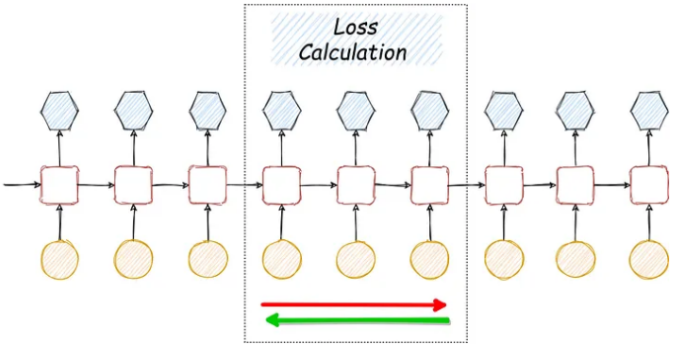

Fig. 4. Truncated Backpropagation Through Time (Truncated BPTT) [7]

other suggested approaches, like:

- Using ReLU activation function.
- Trying the Long-Short Term Memory (LSTM) architecture, where the forget gate might help.
- Initializing the weight matrix, W, with an orthogonal matrix, and using it throughout the entire training (W being orthogonal prevents exploding or vanishing).

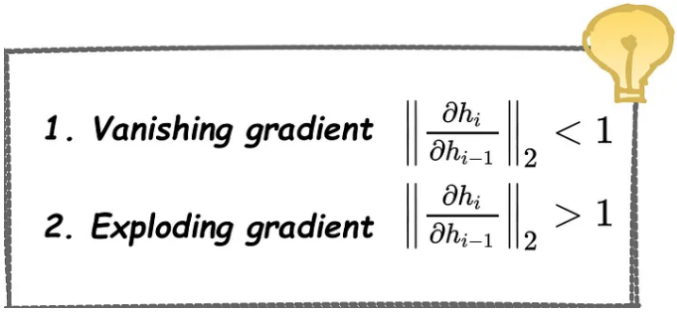

Fig. 5. Vanishing and Exploding Gradients [7]

Another technique is Gradient Clipping, where we set a threshold to control the magnitude of the gradient. It's like saying, "Don't let it get too big."

$$\text{Gradient Clipping: } \text{clip}(g, \text{threshold}) = \max\left(1, \frac{\text{threshold}}{||g||}\right)$$

By applying Gradient Clipping, we adjust the magnitude without changing the direction. As the derivative of the hidden state (h) is the culprit for exploding gradients, we clip the following:

$$\frac{\delta h(i)}{\delta h(i-1)}$$

Choosing the right threshold in Gradient Clipping is vital, and we aim to select the highest threshold that solves the exploding gradient problem by observing the gradient norm curve [7].

B. Long Short Term Memory network

A type of recurrent neural network (RNN) that can learn long-term dependencies in sequential data is called Long Short Term Memory network(LSTM). They have a special structure that allows them to store and manipulate information over time, using gates to regulate the flow of information. To

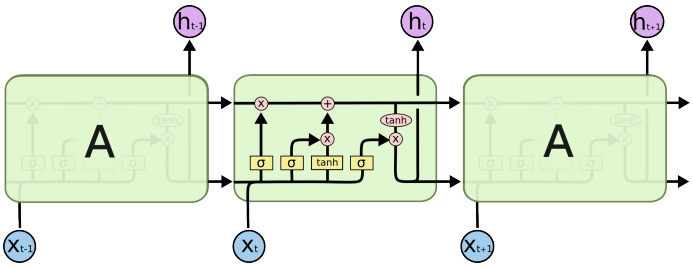

Fig. 6. The repeating module in an LSTM containing four interacting layers. [8]

overcome the limitations of standard RNNs, which struggle to learn from distant past inputs, LSTMs introduce a memory cell that can retain or forget information as needed. The memory cell is controlled by three gates: the forget gate, the input gate, and the output gate. These gates decide what information to keep, update, or output from the cell state, based on the current input and the previous hidden state [8]. The following steps illustrate how an LSTM works on a single input sequence element:

- **Forget gate:** A sigmoid layer that outputs a vector of values between 0 and 1, indicating how much of the previous cell state to forget. For example, if the input is a new subject in a sentence, the forget gate might want to forget the gender of the previous subject.
- **Input gate:** A sigmoid layer that outputs a vector of values between 0 and 1, indicating how much of the new input to add to the cell state. A tanh layer that outputs a vector of candidate values for the cell state. These two vectors are multiplied and added to the previous cell state (after applying the forget gate) to produce the new cell state. For example, if the input is a new subject in a sentence, the input gate might want to add the gender of the new subject to the cell state.
- **Output gate:** A sigmoid layer that outputs a vector of values between 0 and 1, indicating how much of the cell state to output as the hidden state. The cell state is passed through a tanh layer and multiplied by the output gate vector to produce the hidden state. For example, if the

input is a new subject in a sentence, the output gate might want to output information relevant to the next word, such as the number or the case of the subject [8].

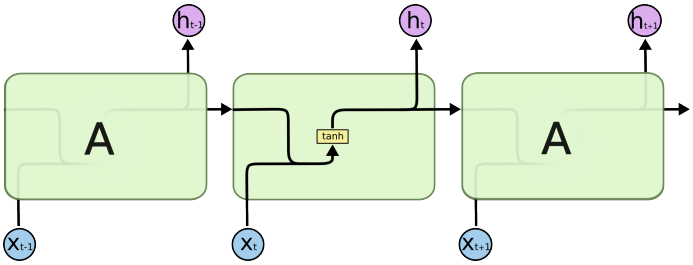

Fig. 7. The repeating module in an LSTM containing a single layers. [8]

C. Stacked LSTMs

A stacked LSTM is an LSTM model with multiple hidden LSTM layers, where each layer contains multiple memory cells. Stacked LSTM can learn complex sequence data by adding levels of abstraction of input observations over time. The input sequence is fed into the first LSTM layer, which

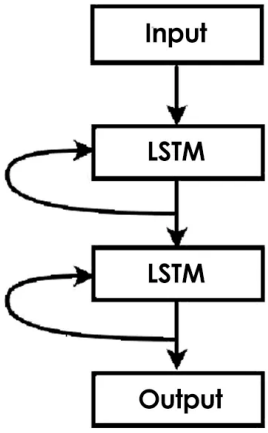

Fig. 8. Stacked LSTM Architecture [9]

outputs a sequence of hidden states. Each hidden state is a vector that contains information about the current and previous inputs. The output sequence of the first LSTM layer is then used as the input sequence for the second LSTM layer, which also outputs a sequence of hidden states. The final output of the network is the last hidden state of the second LSTM layer, which is then passed to a dense layer with a softmax activation function to produce the predicted class probabilities.

The stacked LSTM network can have more than two hidden layers, depending on the complexity of the problem and the data. Each hidden layer can have a different number of units, which determines the dimensionality of the hidden state vector. The number of units and layers can be tuned based on the performance and efficiency of the network. The stacked LSTM network can also use different types of LSTM cells, such as bidirectional LSTM, peephole LSTM, or gated recurrent unit (GRU), which have slightly different architectures and functions.

Stacked LSTM can potentially be more efficient and expressive than a single-layer LSTM, as it can operate at different time scales and capture hierarchical features of the data. Stacked LSTM has been successfully applied to speech recognition, natural language processing, and other challenging sequence prediction problems [9].

D. Gated Recurrent Unit (GRU)

GRU is a kind of recurrent neural network (RNN) that is similar to LSTM, but simpler and faster. It can learn from sequential data, such as words or numbers, by remembering or forgetting some information over time.

GRU does not have a separate memory cell like LSTM, but uses a "candidate activation vector" instead. This vector is updated by two gates: the reset gate and the update gate. The reset gate decides how much to forget from the previous memory, and the update gate decides how much to add from the new input [10].

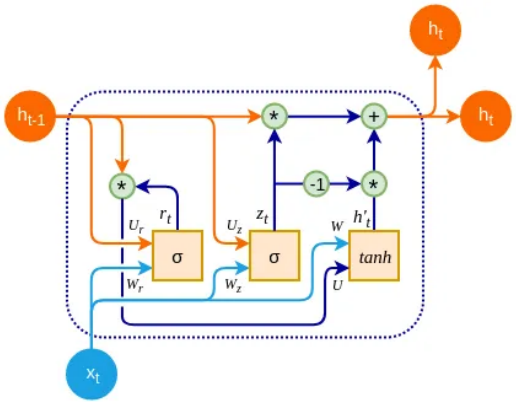

Fig. 9. GRU Architecture [10]

GRU has the following parts:

- **Input layer:** It takes the sequential data and gives it to the GRU.
- **Hidden layer:** It is where the memory is updated at each time step, based on the input and the previous memory.
- **Reset gate:** It decides how much to forget from the previous memory. It uses the input and the previous memory as inputs, and outputs a number between 0 and 1 for each memory element.
- **Update gate:** It decides how much to add from the candidate activation vector to the new memory. It uses the input and the previous memory as inputs, and outputs a number between 0 and 1 for each memory element.
- **Candidate activation vector:** It is a modified version of the previous memory that is reset by the reset gate and combined with the input. It uses tanh function that limits its output between -1 and 1.
- **Output layer:** It takes the final memory as input and produces the output. The output can be a number, a

sequence, or a probability distribution, depending on the task [10].

Certain advantages of GRU are:

- GRU is simpler and faster than LSTM, with fewer parameters to learn.
- GRU can handle long-term dependencies in sequential data by selectively remembering and forgetting.
- GRU can do well on many tasks, such as language, speech, and music.
- GRU can do both sequence-to-sequence and sequence classification tasks

While some of the disadvantages of GRU are:

- GRU may not do as well as LSTM on very long or complex sequences.
- GRU may overfit more than LSTM, especially on small data.
- GRU needs careful tuning of hyperparameters, such as the number of memory elements and the learning rate, to do well.
- GRU may not be easy to understand, since the gates can make it hard to see how the memory is updated.

Overall, GRU networks are a useful way to deal with sequential data, especially when there is a need for less complexity or more efficiency in the model [10].

III. Method Implementation

The entire experiment was conducted using Python as the primary programming language for stock price prediction. Keras is one of the libraries that we extensively used due to its high-level API, ease of use, compatibility with TensorFlow, modularity, community support, flexibility in handling various neural network architectures, and customization capabilities.

Utilizing Apple Stock data for predictions, our dataset comprises 7 columns (features) and 2517 rows (observations). A snippet of the dataset is displayed below.

Date	Open	High	Low	Close	Adj Close	Volume
2013-11-18	18.749643	18.828215	18.507143	18.522499	16.238518	244944000
2013-11-19	18.536785	18.692142	18.498928	18.555357	16.267321	208938800
2013-11-20	18.543928	18.586430	18.368929	18.392857	16.124859	193916800
2013-11-21	18.485714	18.614643	18.345358	18.612143	16.317110	262026800
2013-11-22	18.554285	18.648571	18.518929	18.564285	16.275150	223725600

Fig. 10. Apple Stock Prediction Dataset Overview

As we aim to forecast the closing price in our dataset, we have generated a plot illustrating the relationship between the closing price and the corresponding dates, organized by year. The depicted graph is: We first preprocessed our data to make our evaluation more precise and for accurate predictions. To this we employed the following methods:

- We first dropped all the rows with any missing values. To verify if there were any rows with missing values we compared the dimensions of the new dataset(with rows removed) with the original one and found that there were no missing values in the dataset.
- We then normalised the data by using the '`MinMaxScaler`' function to transform the features of our dataset within a specific range, in our case between 0 and 1.
- We created sequences capture temporal dependencies of our data, represent time series data for neural networks, and enable model training to learn historical patterns and make accurate predictions.
- Utilized Principal Component Analysis (PCA) to understand feature importance, aiming to simplify the model, reduce noise, and enhance interpretability in predicting stock prices.
- Finally we divided the dataset into training and testing subsets, with the training set covering daily stock price predictions from 2014 to 2020 and the validation set spanning daily predictions from 2021 to 2023. The depicted graph is:

Fig. 11. Closing Price Over Time

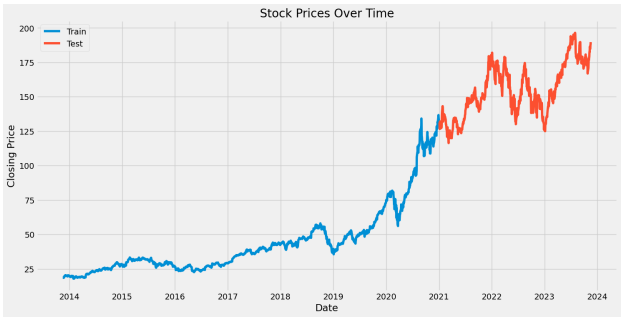

Fig. 12. Training and Testing Datasets

To prevent overfitting we have used an Early Stopping. Early stopping is an optimization technique used to reduce overfitting without compromising on model accuracy. The main idea behind early stopping is to stop training before a model starts to overfit [11].

A. LSTM Model

The architecture of our LSTM model consists of an input layer, four hidden layers and a final output layer.

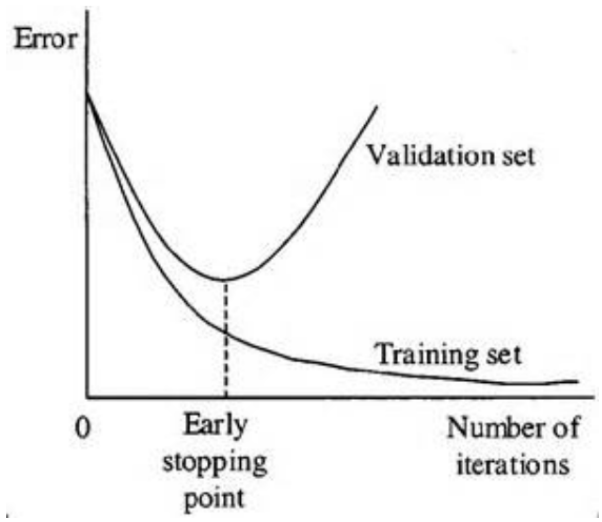

Fig. 13. Early Stopping [11]

- We start with a Initialising a sequential model, which is a linear stack of layers and start adding layers to it.
- First LSTM layer that is added to the model has 50 neurons, configured to return sequences for the subsequent layers. Dropout is applied to prevent overfitting by randomly setting a fraction of input units to zero during training.
- The second LSTM layer is similar to the first one. The *return_sequences=True* ensures that the output of this layer is suitable for the subsequent layer.
- We next add a third LSTM layer similar to the previous two. The choice of multiple layers allows the model to learn hierarchical features from the input sequence.
- The fourth LSTM layer is added without *return_sequences* since it is the last LSTM layer. This layer is responsible for capturing higher-level patterns in the input sequence.
- The output layer is the final layer of our network with one neuron since it is a regression task. The model will predict a single continuous value.

For training our model we have experimented both with the RMSprop Optimizer and Adam Optimising function.

RMSprop is a method used for training neural networks. It was introduced by Geoffrey Hinton, a pioneer in the field of deep learning. When we train neural networks, we face challenges like vanishing or exploding gradients, which happen when the data moves through the network. RMSprop tackles this issue by using a kind of average of the squared gradients to adjust the learning process. In simpler terms, it helps balance the learning rate, making big adjustments for large changes and smaller adjustments for smaller changes. In a nutshell, RMSprop adapts the learning rate as the network learns, making it more flexible and effective during training [12]. RMSprop's update rule:

$$v_{dw} = \beta \cdot v_{dw} + (1-\beta) \cdot dw^2$$

$$v_{db} = \beta \cdot v_{dw} + (1-\beta) \cdot db^2$$

$$W = W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}} + \epsilon}$$

$$b = b - \alpha \cdot \frac{db}{\sqrt{v_{db}} + \epsilon}$$

Fig. 14. RMSprop's update rule [12]

Adam, introduced by Kingma and Ba in 2014, is a method for optimizing stochastic objective functions using gradient information. It relies on adaptive estimates of lower-order moments. Many people in the machine learning community use Adam as it is considered one of the latest and most effective optimization algorithms. The update direction is determined by normalizing the first moment by the second moment. RMSprop's update rule:

$$\theta_{n+1} = \theta_n - \frac{\alpha}{\sqrt{\hat{v_n}} + \epsilon} \hat{m_n}$$

Fig. 15. Adam's update rule [12]

The loss function we employed is the Root mean squared error for our analysis. The primary metric frequently employed in regression tasks is the RMSE (root-mean-square error), which is computed as the square root of the average squared difference between the actual and predicted scores:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}} \tag{6}$$

In this equation, $y(i)$ represents the true score, $y_i\hat{}$ represents the predicted value, and 'n' represents the number of data points. Finally we train this model for different number of epochs and batch sizes for our analysis.

B. Stacked LSTM

The Stacked LSTM is similar to the normal LSTM model since both the stacked and normal LSTM models use LSTM layers, but the stacked model repeats the LSTM layer multiple times which makes the model architecture deeper. The stacked LSTM model has more parameters due to the increased number of LSTM layers which increases the model's complexity and capacity to capture intricate patterns in the data. We have added a Dropout layers after each LSTM layer with a dropout rate of 0.2 to prevent overfitting.

We have experimented with both Adam and RMSprop optimizers for our training and analysed the results for different number of epochs and batch sizes. We have also employed early stopping to monitor the validation loss and stop training early if there is no improvement for a specified number of epochs.

We have also experimented with both single-layer GRU and stacked GRU. The architecture of these models are similar to the LSTM models. GRU has a simpler architecture compared to LSTM. It consists of a single hidden state that combines

information from the current input and the previous hidden state. GRU merges the cell state and hidden state into a single state, making it computationally more efficient. A Stacked GRU involves stacking multiple layers of GRU units on top of each other. Each layer provides a hierarchy of representations.

IV. Experiments And Analysis

We started with a sequence length of 7 days for our models. To understand the trend better and the performance of the model we then changed the sequence length to 14 days(2 Weeks), 30 days(1 Month) and 60 days(2 Months). Upon experimentation we realised our model works best when we consider the weekly data that is when the sequence length is 7 days since increasing the sequence length in LSTM models for stock price prediction may initially improve performance as longer sequences capture more temporal patterns. However, there's a trade-off because longer sequences can also lead to vanishing or exploding gradient problems during training.. The results obtained for the different sequence lengths for LSTM model are in the table below.

Model	Sequences	Optimizer	Root Mean Square Error
LSTM	7	rmsprop	4.7708
LSTM	14	rmsprop	4.3555
LSTM	30	rmsprop	4.2614
LSTM	60	rmsprop	4.0041

TABLE I
Errors with changing sequences

After finalising the sequence length of 7 we experimented with the Optimising functions for different batch sizes. A table containing the results of the models for distinct optimizers, batch size and the corresponding Root Mean Squared error is:

Model	Optimizer	Batch Size	Root Mean Square Error
LSTM	rmsprop	32	4.7708
LSTM	Adam	32	4.4554
LSTM	SGD	32	4.3476
Stacked LSTM	rmsprop	32	5.5521
Stacked LSTM	Adam	32	5.6363
Stacked LSTM	SGD	32	26.7472
GRU	rmsprop	150	3.7123
GRU	Adam	150	3.6444
Stacked GRU	rmsprop	150	5.5384
Stacked GRU	Adam	150	4.4724

TABLE II
Error analysis with different optimizers

From the table we can observe the following:

- The LSTM model works best when we use the Stochastic Gradient Descent (SGD) optimization function, giving us the most accurate predictions with the lowest error (RMSE of 4.3476).
- For the stacked LSTM model, it performs better when we use the Root Mean Squared Propagation (RMSprop) optimizer, resulting in an error (RMSE) of 5.5521. This means RMSprop is a good match for the stacked LSTM architecture.

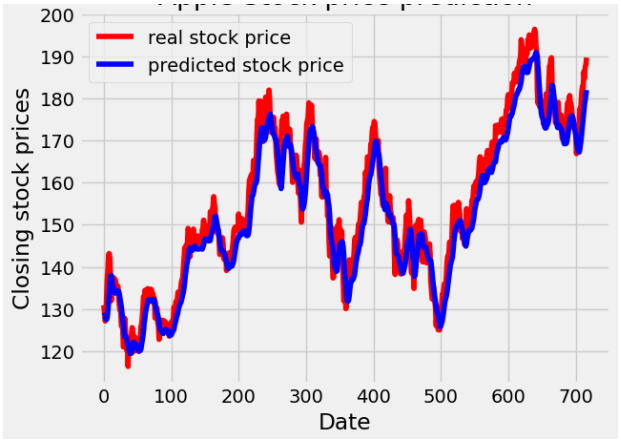

Fig. 16. Stock Price Prediction using LSTM and RMSprop

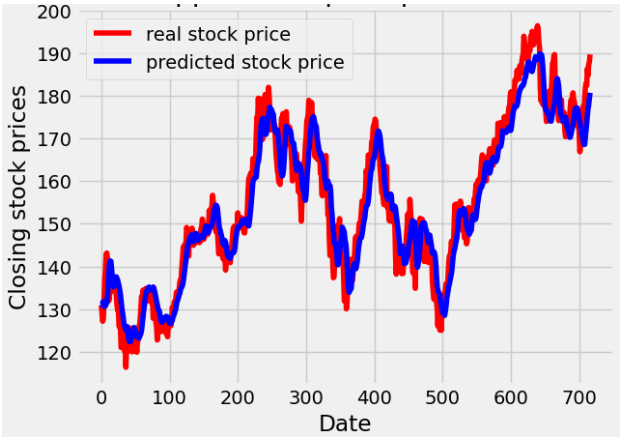

Fig. 17. Stock Price Prediction using Stacked LSTM and RMSprop

- Both the GRU and stacked GRU models do well when paired with the Adam optimizer. This optimizer choice gives competitive and effective results for both models, showing its suitability for capturing patterns in the data.

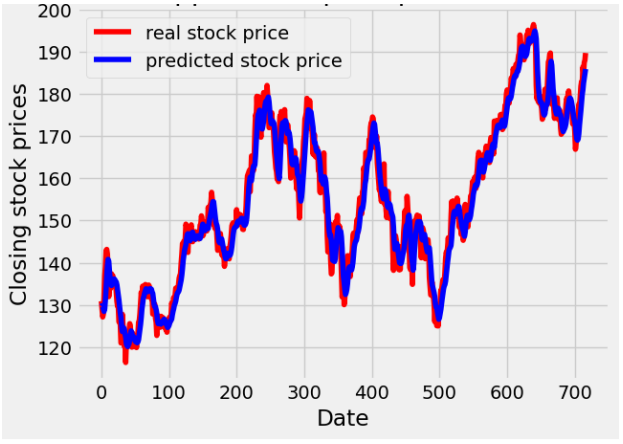

Fig. 18. Stock Price Prediction using GRU and Adam

In a nutshell, the type of optimizer used has a noticeable

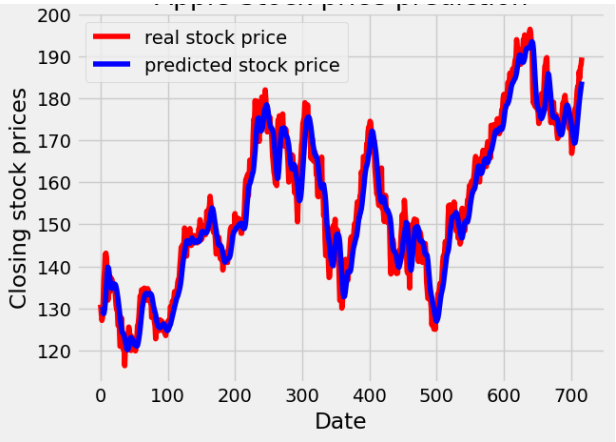

Fig. 19. Stock Price Prediction using stacked GRU and Adam

impact on the performance of these models. Choosing the right optimizer for each architecture is key to getting the most accurate predictions for stock prices. We have noticed that we get the best prediction using the Adam optimizer along with the GRU model.

V. Reflection on project

The project used different types of RNNs, LSTM, Stacked LSTM, GRU and Stacked GRU. Each type was chosen for specific reasons. LSTMs were picked for handling the vanishing gradient problem and catching long-term patterns. Stacked LSTMs aimed to get better at recognizing complex patterns. Adding GRU was done to compare its performance with LSTM but it was found that the most accurate predictions was done by the GRU model. Hence falsifying our assumption.

We adjusted hyperparameters, like learning rates, batch sizes, and epochs, to make the training process better. We used specific data sets to see how well the models could adapt to new information, using standard measures like Mean Squared Error (MSE) loss and Root Mean Squared Error (RMSE) to check how well they did.

Looking ahead, we have some ideas for future work based on what we learned. One idea is to explore ensemble models, combining predictions from different RNN types to benefit from their different strengths. We might also try adding more information, like technical indicators or sentiment analysis from news, to see if it makes the models better. Trying more advanced RNN types or combining RNNs with other types of networks could also be interesting. We could improve focus on important time steps by adding attention mechanisms. Applying dropout or other techniques to prevent overfitting, especially in deep models, is another option.

We could also experiment with dynamic learning rates to see if that helps models learn better during training, making them more accurate overall. Extending our analysis to longer time periods could help us understand how well the models work for medium to long-term predictions. Addressing these points sets the stage for making RNN models better at predicting stock prices.

References

[1] Y. Finance, "Yahoo finance," *Retrieved from finance. yahoo. com: https://au.finance.yahoo.com/quote/AAPL?p=AAPL.tsrc=fin-srch*, 2023.

[2] B. G. Malkiel, "Efficient market hypothesis," in *Finance*. Springer, 1989, pp. 127–134.

[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[4] Q. Gao, "Stock market forecasting using recurrent neural network," Ph.D. dissertation, University of Missouri–Columbia, 2016.

[5] georgiannacambel, "Google stock price prediction using rnn – lstm," *kgptalkie.com*, 2020 (Accessed 21. 11. 2023). [Online]. Available: https://kgptalkie.com/google-stock-price-prediction-using-rnn-lstm

[6] aishwarya.27, "Introduction to recurrent neural network," *www.geeksforgeeks.org*, 2023 (Accessed 21. 11. 2023). [Online]. Available: https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network

[7] B. Or, "The exploding and vanishing gradients problem in time series," 2020 (Accessed 21. 11. 2023). [Online]. Available: https://towardsdatascience.com/the-exploding-and-vanishing-gradients-problem-in-time-series-6b87d558d22

[8] C. Olah, "Understanding lstm networks–colah's blog," *Colah. github. io*, 2015.

[9] J. Brownlee, "Stacked long short-term memory networks," *URL: https://machinelearningmastery. com/stacked-lo ng-short-term-memory-networks*, 2017.

[10] Anishnama, "Understanding gated recurrent unit (gru) in deep learning," *URL: https://medium.com/@anishnama20/understanding-gated-recurrent-unit-gru-in-deep-learning-2e54923f3e2*, 2023 (Accessed 21. 11. 2023).

[11] A. Mustafeez, Zohair, "What is early stopping?" *URL: https://www.educative.io/answers/what-is-early-stopping*, 2023 (Accessed 21. 11. 2023).

[12] Sanghvirajit, "A complete guide to adam and rmsprop optimizer," *URL: https://medium.com/analytics-vidhya/a-complete-guide-to-adam-and-rmsprop-optimizer-75f4502d83be*, 2021 (Accessed 21. 11. 2023).