RNNs for Stock Price Prediction

Chpristopher Hamilton
The University of Adelaide
christopher.hamilton@student.adelaide.edu.au

Abstract

This paper presents methods of using Recurrent Neural Networks (RNNs) to predict stock prices based on historical data. Different types of RNNs were tested during the experiments conducted, with the Mean Squared Error (MSE) being used as the main indicator of how well a model performs in predicting data. From the multiple experiments that were run, the best performing model achieved an average MSE of 0.01753% on the test data, indicating that on average, there was a difference of 0.01753% between the model's prediction and the actual value. This was achieved using an Long Short-Term Memory (LSTM) model, after training on Google's Stock Data and then testing on other unseen stock data. This report goes into detail about how the results were achieved and exactly what experiments were run in the use of RNNs, as well as linking to the GitHub repository and Jupyter Notebook containing graphs generated during experimentation and the code used to reproduce the results. It was found that given historical data, predicting the next day's closing stock price can be done reliably with only small levels of error.

1. Introduction

Predicting stock prices is a problem that investors would like to solve to allow them to make more informed investment decisions in the future [13]. Different stocks on the stock market can have different levels of volatility, with some increasing and decreasing rapidly, and others being steadier. Stocks represent the ownership of a fraction of a company that issues them and are issues by corporations for the purpose of raising funds to operate the business. [4] To analyse the performance of stocks, two options are available, fundamental analysis and technical analysis. Fundamental analysis focuses on a company's assets, financial statements, management, strategy and performance with consumers. These factors, which all relate to a company's performance, can indicate how well an investment in the company can do in the long term, and will not change significantly based on short term news.

Technical analysis, on the other hand, focuses on measurable data from stocks including stock prices, historical returns and volumes of trades. Often used for short term trading, it aims to identify any trading signals and predict if a stock will increase or decrease from its current price. [7] Using Recurrent Neural Networks (RNNs) to predict a stock price will focus on a technical analysis of the stocks, using past prices and volumes to predict the next output. The model needs to be trained to handle each step, and an output is produced for each step using the input variables and the internal state of the network. The internal state of the RNN is made up of the weights that allow the network to perform prediction based on the input and is updated through training the network on labelled data with known outputs.

2. Background

The problem of predicting stock princes has multiple methods that can be implemented. A traditional "technical analysis is the study of the past price movements of an individual share or the market as a whole" [8]. Done using different types of charts like line charts, bar charts and candle stick charts, different trends can be established in the data to provide investors with guidance around the stock's future performance. For example, an uptrend in the data may indicate that it is likely that the price of the stock will continue to follow this trend, and continue rising. Data other than the price of the stock is also important in technical analysis. For example, if the volume of shares being sold is high at the time an uptrend is emerging, it may confirm that the stock's price is expected to follow that trend. [8] The relationship between multiple variables and how they affect the stock price is able to be modelled by a neural network to allow for predicting the price. Even though stocks can be volatile, technical analysis as a method of predicting stock prices is expected to have merit because of the expectation that investors will factor in a lot of the information that can come from a fundamental analysis when trading stocks, and this will have an effect on the data being analysed.

Traditionally, fundamental analysis of stocks requires the trader to gather information about companies, their lead-

ership, financial reports and more, and understand the effects that this information can have on the price of the company's stock. When attempting to perform fundamental analysis this way on many companies, investors must spend long amounts of time gathering data from many sources, processing and understanding it, then understanding patterns and making decisions based on it [1]. Artificial intelligence (AI) can be used to assist in fundamental analysis in a different way compared to in technical analysis, by gathering information from a company from sources like articles, press releases, reports, and social media and processing this information into a format that can be used to recognise patterns and make decisions. The use of tools which integrate AI with fundamental analysis are able to automate a lot of the tasks that investors would need to do, as well as understand patterns in the data that may not be clear to a human. However, they do require high quality data as inputs, since this input data will impact the predictions the model makes about the performance of the stock. It may also be unclear to the investor why exactly the model came to a particular conclusion through fundamental analysis due to the large amount of data that is used for prediction [1].

The application of RNNs for stock price prediction is essentially allowing a neural network to perform a technical analysis on the data. Except rather than using the charts to make predictions, the neural network uses trained weights and current inputs to produce a predicted output. This approach is similar to a technical analysis using charts, as data such as historical price and volume is used as part of the parameters of the network when making a prediction. Parameters of the RNN are able to be configured to specify how much of the historical data should be used when making a prediction. RNNs are "trained on sequential or time series data to create a machine learning (ML) model that can make sequential predictions or conclusions based on sequential inputs" [12]. Compared to other neural networks like convolutional neural networks (CNNs), RNNs, include a memory about information from prior inputs as part of the decision making to predict an output for the current input. RNNs are therefore able to be used in cases where the output for a given point in the sequence is impacted in some way by previous outputs in the sequence.

There are a number of different types of RNNs that could be applied to predicting stock prices based on the previous prices. In a standard RNN, the output for the time step depends on the current input and the hidden state from previous time steps. However, this model can be impacted by vanishing gradients and be unable to learn long term dependencies [12]. While this may be an issue for some tasks, it could be a valid model for predicting stock prices using a technical analysis based on the price and other data, since it is possible that using only the short term history may produce an accurate prediction. With this issue of being unable to learn long term dependencies in a standard RNN, a different RNN architecture called Long Short-Term Memory (LSTM) was developed. Where LSTM networks are able to apply historical information to the prediction based on the given input [10].

Given these possible methods for stock price prediction, this paper will investigate RNNs for stock price prediction in particular and how effective an RNN can be in predicting a stock's price based on previous information. It will compare standard RNN models as well as LSTM RNNs to understand the differences in performance when generating predictions.

3. Method

RNNs generate predicted outputs based on their current state, including weights for the input variables, and the input variables themselves. An activation function is used to perform the prediction based on these variables. The hidden state is also updated as information is fed through the network to allow previous states to have an effect on the current prediction. A simple RNN will be implemented in Python to test the performance of predicting the stock closing price for a day based on the current day open price, maximum price for the day, minimum price for the day and volume. This RNN will consider the history of the past days in order to make the prediction. The model will include weights from the input layer to the hidden layer, from the hidden layer back to the hidden layer and from the hidden layer to the output layer, where the recurrent nature of the model will require the weights from and to the hidden layer to be used.

We denote the length of the input n, the length of the hidden states k and the output size 1, since we are only predicting a single price. We can then denote the weights from the input layer to the hidden layer to be a matrix with k rows and n columns, called W_{xh}. Similarly the weights from the hidden layer to the hidden layer is denoted W_{hh} and is a matrix with k rows and k columns, and the weights from the hidden layer to the output layer denoted W_{hy} is a matrix with 1 row and k columns. These weights should all be initialised to random values as they will need to be trained for the task of predicting stock data. The hidden states of the RNN is denoted $\vec{h}$ and is a column vector with k elements, and the input is denoted $\vec{x}$, a column vector with n elements. [2] With this notation, stepping forward through the neural network can be achieved by taking a single input vector for this timestep $\vec{x_t}$, and calculating the next hidden state from it. The hidden state can be calculated with:

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

This will produce the next hidden state for the network, which is still a column vector with k elements. The predic-

tion for the output value can now be calculated with:

$$\hat{y}_t = f(W_{hy} \cdot \vec{h_t})$$

Where f is the activation function used for the network.

This will produce a prediction for the output and can be compared to the actual output value in order to calculate the loss. The mean squared error can be used for the loss function of this RNN.

$$L = \frac{1}{T}\sum_{t=1}^{T}(\hat{y}_t - y_t)^2$$

In order to calculate the mean squared error (MSE), we must calculate the squared error for each step. So for this step calculate:

$$l_t = (\hat{y}_t - y_t)^2$$

Once predictions have been made for all inputs, calculate the MSE with $L = \frac{1}{T}\sum_{t=1}^{T}(\hat{y}_t - y_t)^2$.

With these equations the algorithm for stepping forward through the network is as follows.

Let:

$$f(z) = ReLU(z) = \max(0, z), z \in \mathbb{R}$$

be the activation function for the RNN.

Let:

$$l_t = (\hat{y}_t - y_t)^2$$

be the error for one prediction.

Let:

$$L = \frac{1}{T}\sum_{t=1}^{T}(\hat{y}_t - y_t)^2$$

denote the MSE for all predictions.

For a set of training data: $\{(\vec{x_t}, y_t)\}_{t=1}^{T}$,

$\vec{h_1} \leftarrow \vec{0}$

for $t = 1$ to T **do**

$\vec{a_t} = W_{xh} \cdot \vec{x_t} + W_{hh} \cdot \vec{h_{t-1}}$

$\vec{h_t} = tanh(\vec{a_t})$

$\hat{y}_t = f(W_{hy} \cdot \vec{h_t})$

$l_t = (\hat{y}_t - y_t)^2$

end for

$L = \frac{1}{T}\sum_{t=1}^{T} l_t$

[2]

Note that in this algorithm, the activation function used could be experimented with, and in place of $ReLU(z)$, we could also use the following functions.

$$f(z) = leakyReLU(z) = \max(0.1z, z), z \in \mathbb{R}$$

$$f(z) = sigmoid(z) = \frac{1}{1+e^{-z}}, z \in \mathbb{R}$$

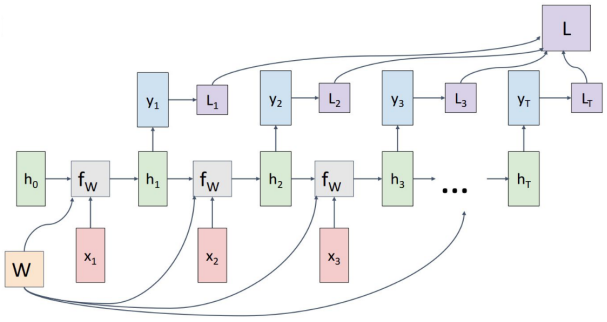

Figure 1. Diagram of forward stepping through an RNN [3]

$$f(z) = tanh(z), z \in \mathbb{R}$$

$$f(z) = elu(z) = \begin{cases} z & \text{if } z > 0, \\ ae^{z-1} & \text{if } z \leq 0 \end{cases} z, a \in \mathbb{R}$$

Figure 1 shows the process of stepping forward through an RNN with the inputs, hidden states, weights and outputs, as well as the loss calculation.

This algorithm allows predictions to be made based on the current inputs to the RNN as well as the weights of the RNN, and for the loss to be calculated to evaluate how good the predictions are. For a model to be performing well, the MSE should be as close to 0 as possible, as this would indicate very little difference between the predicted value of $\hat{y}_t$ and y_t. It can also be seen that the recurrent updating of the hidden layers is what makes the RNN recurrent in nature, and as more inputs are applied to the network, the hidden layers will update.

When training the RNN, the backpropagation through time (BPTT) algorithm needs to be used to update the weights of the network. The BPTT algorithm involves calculating the gradients of the weights at each step, going backward through the inputs, and updating the weights W_{xh}, W_{hh} and W_{hy} iteratively. The backpropagation is only done in the training stage for the network, and a backward pass is made through all inputs after the prediction $\hat{y}_t$ has been calculated. It relies on a learning rate parameter to choose how significant each update to the weight based on the calculated gradients should be.

The backwards pass through the inputs to update the weights is as follows.

For a set of training data: $\{(\vec{x_t}, y_t)\}_{t=1}^{T}$, and predictions $\{\hat{y}_t\}_{t=1}^{T}$,

Let $learningRate = \mu$

$\frac{\partial l}{\partial W_{xh}} \leftarrow 0$, $\frac{\partial l}{\partial W_{hh}} \leftarrow 0$, $\frac{\partial l}{\partial W_{hy}} \leftarrow 0$

$\frac{\partial l}{\partial y_t} \leftarrow -2(\hat{y}_t - y_t)$

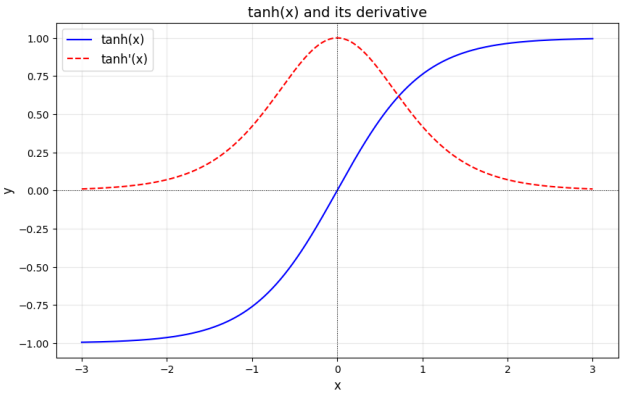

Figure 2. Tanh function and its derivative

for $t = T$ to 1 **do**

$$\frac{\partial l}{\partial \vec{h_t}} = W_{hy}^T \frac{\partial l}{\partial y_t} + \frac{\partial l}{\partial \vec{h_{t+1}}}$$

$$\frac{\partial l}{\partial \vec{a_t}} = (1 - \vec{h_t})^2 \frac{\partial l}{\partial \vec{h_t}}$$

$$\frac{\partial l}{\partial W_{xh}} \mathrel{+}= \frac{\partial l}{\partial \vec{a_t}} \vec{x_t}^T$$

$$\frac{\partial l}{\partial W_{hh}} \mathrel{+}= \frac{\partial l}{\partial \vec{a_t}} \vec{h_{t-1}}^T$$

$$\frac{\partial l}{\partial W_{hy}} \mathrel{+}= \frac{\partial l}{\partial y_t} \vec{h_t}^T$$

end for

$$W_{xh} \mathrel{-}= \mu \frac{\partial l}{\partial W_{xh}}$$

$$W_{hh} \mathrel{-}= \mu \frac{\partial l}{\partial W_{hh}}$$

$$W_{hy} \mathrel{-}= \mu \frac{\partial l}{\partial W_{hy}}$$

[14]

One of the deficiencies of the RNN BPTT algorithm is the occurrence of vanishing gradients if there are long term dependencies in the model. Due to the face that the weights W_{hh}, which connect the hidden layers to themselves in the model are updated iteratively based on the hidden states, the $\frac{\partial l}{\partial W_{hh}}$ gradient can often become small in its calculation, and as it approaches 0 it is unable to update the weights of the hidden state to hidden state. This occurs because the derivatives of the $tanh$ function are multiplied together repeatedly during the BPTT algorithm, as shown in the algorithm above with the calculation for $\frac{\partial l}{\partial \vec{a_t}}$ and $\frac{\partial l}{\partial W_{hh}}$. Because of this, it can be difficult or impossible for RNNs to learn the long term dependencies in the data [6]. This can be seen in Figure 2, as the maximum of the derivative is 1, and any other values will eventually lead to a vanishing gradient, given enough time steps.

One solution to the issue of vanishing gradients is to use a different RNN structure called a Long Short Term Memory (LSTM) network. Like standard RNNs, they include a recurrent block in their structure, but the internals of this recurrent block are different. As shown in the differences between Figure 3 and 4, LSTMs include the ability to add or remove information from the hidden states using "gates" [10]. The implementation of these gates is done using sigmoid functions, outputting a number between 0 and 1 representing the proportion of information to allow through [10]. This allows the LSTM to include information from the long term, and address the problem introduced by vanishing gradients in the standard RNN models.

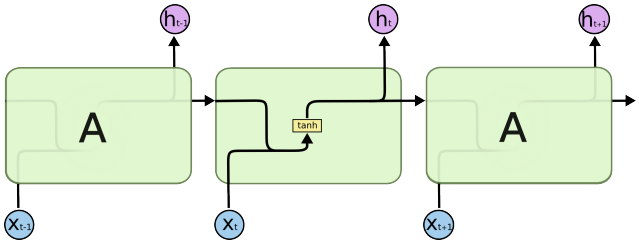

Figure 3. Repeating structure of RNN [10]

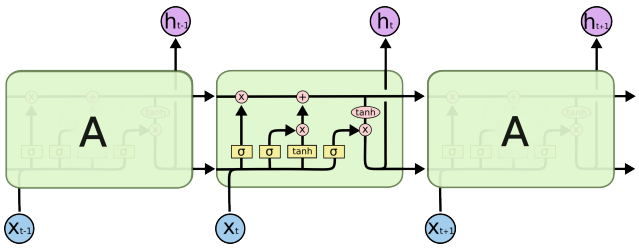

Figure 4. Repeating structure of LSTM [10]

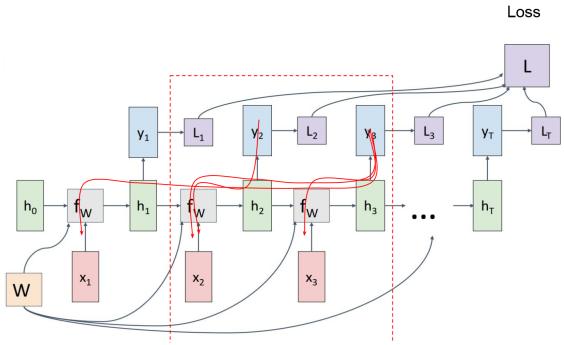

Figure 5. Sliding window RNN [3]

Another method that can be considered for training an RNN is the use of a "sliding window" over the data. This method uses only a section of the data when applying the BPTT algorithm, rather than all inputs, allowing the model to learn the short term dependencies, rather than try to learn how all previous history will affect the prediction [5]. The implementation of this model is similar to a standard RNN, however, the hidden states are reset for each window, preventing the inputs outside of the window from having any effect.

The analysis of each of these methods will provide insight into how they perform in the prediction of stock prices.

4. Experimental Analysis

To predict stock prices based on previous history using RNNs, the models described above were each implemented in Python and trained using labelled stock data which included the Date, Open price, Close price, Low price, High price and Volume of sales for each day. Training of the Standard RNN, Window RNN and LSTM was done for 20 epochs, where each epoch passes over the training dataset.

The stock's Closing price for the day was chosen as the response variable for the model and the other values of Open price, Low price, High price and Volume of sales were chosen as the predictor variables which the response is dependent on. The Closing price was chosen as the variable to predict, since investors would be able to use the closing price for the day to perform end of day trading. [11]. Having predicted values for the following day's closing price would allow investors to make the most profitable end of day orders for the next day.

The number of days to be used as part of the sequence to predict a new value is one of the hyper-parameters that needs to be chosen as part of setting up the model before training. In order to tune the hyper-parameters, a validation data set needed to be used, and it was decided that the Google stock test dataset could be used for validation, while testing would be completed on datasets for other stocks. In the tuning of the hyper-parameters, different values were tested, and the hyper-parameters that produced the smallest validation loss were then used to train models to test on the data from other stocks.

	# days in sequence		
Activation	3	5	10
ReLU	0.0396	0.0420	0.0687
Leaky ReLU	0.0395	0.0426	0.0674
ELU	0.0394	0.0428	0.0693
tanh	0.0434	0.0362	0.0465
Sigmoid	1.2799	1.9876	1.417

Table 1. Validation loss calculated for different activation functions and number of days in the sequence in RNN

As seen in Table 1, the parameters that produced the smallest validation loss were the tanh activation function with 5 days in the sequence and the ELU activation function with 3 days in the sequence. It can also be seen that the sigmoid activation function produced significantly higher validation loss values and is likely not a good activation function for this dataset. The ReLU and Leaky ReLU activation functions also produced validation loss values close to the minimum for 3 days in the sequence and could be worth using as the hyper-parameters for the final model.

Ten randomly selected stocks were chosen from the Kaggle dataset of stocks available at `https://www.kaggle.com/datasets/paultimothymooney/stock-market-data`, and labelled Stock 1 to Stock 10 for the purposes of this report, to test the trained models on. For consistency, all training of the models was done on the Google stock data training set that is included in the repository. Once trained, the models would then be tested on the ten different stocks, and the test loss value as well as the graph comparing predicted value to actual value would be used to determine how well the model performs. The loss values calculated and displayed in this section are normalised to a value between 0 and 1 to allow for comparing the performance of different models on different stocks. This was necessary, since the stocks each have different price ranges, and a higher error for one stock may have been caused by a higher price rather than the model actually not performing as well.

Stock	Simple	Window
1	0.0241	0.0224
2	0.0112	0.0103
3	0.0076	0.007
4	0.0099	0.0091
5	0.0257	0.0237
6	0.0143	0.0132
7	0.0107	0.0099
8	0.0091	0.0084
9	0.048	0.045
10	0.0044	0.0041
Average	0.01652	0.01531

Table 2. Test loss calculated for models with the tanh activation function

Since the tanh activation function with 5 days as the sequence length produced the lowest validation loss, it was tested first in the Simple RNN model and the RNN model with a sliding window. The results of this test are summarised in Table 2. It can be seen in this table that the Simple RNN model produced an average loss of approximately 1.65% and the RNN model with a window produced an average loss of approximately 1.53%. These values are very close and by looking at this data, it is possible the model with the window performs slightly better than the simple model.

Figure 6 also contains an example of a graph produced where the orange line shows the actual close price of the stock and the blue line shows the predicted value. All graphs are not included in this report due to their size, however this one is included to demonstrate that there was a clear difference in the predictions and actual values. Other generated graphs are able to be seen in the Jupyter Notebook in the GitHub repository or under the images directory in the same repository. The graph shows that the model

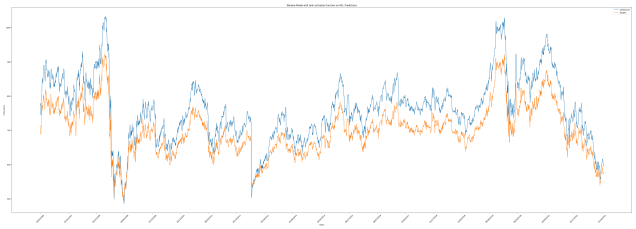

Figure 6. Example of graph produced from predictions with model using tanh activation function.

Figure 7. Example of graph produced from predictions with model using tanh activation function.

is somewhat following the actual value, but the predictions seem consistently higher than the actual close price of the stock. This indicated that there could potentially be better performance by using different hyper parameters, so the next best performing on the validation set was chosen to be tested.

Stock	Simple	Window
1	0.0005	0.0005
2	0.0001	0.0002
3	0.0001	0.0001
4	0.0001	0.0001
5	0.0003	0.0003
6	0.0002	0.0002
7	0.0001	0.0001
8	0.0001	0.0001
9	0.0005	0.0006
10	0.0001	0.0001
Average	0.0001990	0.0002078

Table 3. Test loss calculated for models with the ELU activation function

As shown in Table 3, the test loss values for this model were much lower compared to the tanh activation function, with an average loss for the simple model being 0.01990% and the average loss for the window model being 0.02078%. The increased performance of the model is also evident when comparing the graph for the same stock as in Figure 6. Figure 7 shows the graph generated for the same stock's close price, but this time using a model with the ELU activation function and a sequence length of 3 days instead. As shown in the graph, the model follows the actual price of the stock much more closely, confirming what is seen with the low error value.

Since the difference between the simple model and the model with the window is very minimal, we cannot confidently say that one model is better than the other. However, it is clear that the models trained using the ELU activation function and the sequence length of 3 days performs much better than the model trained with the tanh activation function for a sequence length of 5 days.

The final model that was tested on the data was the LSTM. As outlined in the code section below, this model uses the LSTM features built in to TensorFlow and Keras to define the model. This LSTM model was trained on the same data and the same stock data was also used to test it.

Stock	LSTM
1	0.000475869
2	7.94E-05
3	3.28E-05
4	6.41E-05
5	0.000231204
6	0.000125614
7	5.67E-05
8	3.50E-05
9	0.000608807
10	4.36E-05
Average	0.0001753

Table 4. Test loss calculated for the LSTM model

The test loss values for this model can be seen in Table 4. From the average loss value of 0.01753%, it is clear that the LSTM model performs similarly to the RNN models that are trained with the ELU activation function and a sequence length of 3 days.

The graphs showing each of the tested stock's actual closing prices and the predicted prices for each day can be found in the GitHub repository under the Assignment3/images directory. The experiments conducted can also be viewed in the Jupyter Notebook under a3.ipynb.

It is clear from the experiments conducted that the use of the ELU activation function with a sequence length of 3 days or the use of the LSTM model can produce highly accurate results when predicting the closing price of the stock on a day, based on the inputs and the data from previous days. The sigmoid and tanh activation functions did not perform as well on the validation or test data respectively and may be better suited for other types of data.

5. Code

The code written to implement each of the models and their training can be found at `https://github.com/CHamilton0/`

COMP-SCI-7318-Deep-Learning-Fundametals, under the 'Assignment3' directory. It has been implemented in Python 3.10.12, and including the use of libraries such as Pandas, Numpy and TensorFlow 2.18.0. To manage the dependencies in this project, the uv package manager is used. Follow the instructions at uv - Installation to install 'uv' for your system. Once 'uv' is installed, install the project's dependencies into a Python Virtual Environment with

```
$ python3 -m venv .venv
$ source .venv/bin/activate
$ uv sync
```

Each model has been implemented in a separate Python source code file, while the Jupyter Notebook file a3.ipynb includes the experiments done for training each model. The notebook has been developed and runs inside Visual Studio Code, using the above virtual environment. However, it is likely other notebook engines such as Jupyter Notebook or Google Colab could be used to run the notebook as long as the dependencies are set up correctly.

In order to load the stock data, helper scripts have been included with the code to convert the CSV stock data into Pandas data-frames. Due to the size of the stock data CSV files, they have not been included in the GitHub repository. Before running the Jupyter Notebook, the data needs to be downloaded to the 'Assignment3/data' directory in the repository.

1. Log in or register to Kaggle at https://www.kaggle.com/account/login
2. Navigate to https://www.kaggle.com/datasets/paultimothymooney/stock-market-data
3. Download the dataset as a zip file
4. Extract the stock_market_data directory under the data directory in the repository. The folder structure should now be Assignment3/data/stock_market_data/

The training function in the code is used to train the standard RNN and the RNN with the sliding window. This function takes in the specific RNN model that has been implemented, as well as the training data, number of epochs, and learning rate. For each epoch, the training data is iterated over and the algorithm described in the method section is applied for the forward pass and back-propagation through time. During training, the Mean Squared Error (MSE) is calculated between the prediction in training and the target value in order to see that the model is minimising the loss function as each epoch progresses.

The testing function in the code is used to test the RNN models, given the model itself, a set of test data, normalisation parameters that have been used and the sequence length. The test function produces predictions for each value in the test set, and computes the model performance by calculating the MSE. This test function only uses the forward stepping algorithm in the method section above. Models that have a smaller MSE value on the same set of test data are assumed to be performing better, since it implies that the difference between the predicted value and actual value is smaller.

With the models and methods to train and test defined, code to find the optimal activation function and sequence length for the model has been written. To find these optimal parameters, the simple RNN model is trained and tested with different sequence lengths and activation functions, on the Google Stock Dataset in the data directory under Assignment3 in the repository, using the test data from this directory as a validation set to choose the optimal hyper-parameters. After finding these optimal hyper-parameters, the different models are trained using the two sets of hyper-parameters which produced the lowest MSE on the validation set used. Testing is then done on five randomly selected stocks from the stock_market_data downloaded in the steps above and the code then uses Matplotlib to plot the predicted value against the actual value in the test dataset, giving a visual indication of how well the model is predicting the closing price. As well as the visualisation, the MSE is also calculated, giving a numerical indicator of performance on the test data used.

Finally, the code uses the Keras LSTM model implementation to test the performance of an LSTM model compared to the standard RNN models. This is trained using the model.fit() method, and then tested by using the model.predict() method for the test data. The MSE is also calculated for the test data in order to compare the performance with the other RNN models tested.

6. Conclusion

In the process of training different RNN models on stock data and then using them to predict the closing price of stocks on a given day, it has become evident that RNNs are a capable tool for performing a form of technical analysis on stock data, and may be able to be used by investors to make decisions. A simple RNN performed approximately as effectively on the data tested as an RNN using a sliding window or an LSTM RNN, and generalised well to previously unseen test data.

In future studies, it would be interesting to combine this RNN method of predicting stock prices with other AI models to include qualitative information as well as quantitative information. This could include automatically performing a fundamental analysis on the stock of a company, and including output from the AI model doing that analysis in the prediction of the stock price. With the low errors produced from the RNN method, including this extra informa-

tion could allow for predicting stock prices further into the future.

It would also be interesting to investigate how this method would apply to different types of securities such as cryptocurrency, since its price is often more volatile than that of stocks [9]. Due to its volatility, it is possible that the same method used for stocks would not work as effectively on cryptocurrencies, but this would need to be investigated.

References

[1] Alphanome.AI. Combining technical and fundamental analysis with ai, 2023. https://www.alphanome.ai/post/combining-technical-and-fundamental-analysis-with-ai.

[2] Javier Cárdenas. Implementing a rnn with numpy, 2021. https://quantdare.com/implementing-a-rnn-with-numpy/.

[3] Deep Learning Fundamentals. Lec6: Recurrent neural networks.

[4] Adam Hayes. Stocks: What they are, main types, how they differ from bonds, May 2024. https://www.investopedia.com/terms/s/stock.asp.

[5] Prerit Khandelwal, Jinia Konar, and Banalaxmi Brahma. Training rnn and it's variants using sliding window technique. In *2020 IEEE International Students' Conference on Electrical,Electronics and Computer Science (SCEECS)*, pages 1–5, 2020.

[6] Katherine (Yi) Li. Vanishing and exploding gradients in neural network models: Debugging, monitoring, and fixing, 2021. https://neptune.ai/blog/vanishing-and-exploding-gradients-debugging-monitoring-fixing.

[7] Katherine (Yi) Li. Machine learning for stock price prediction, March 2024. https://neptune.ai/blog/predicting-stock-prices-using-machine-learning.

[8] ASX Limited. Course 11 technical analysis, 2010. https://www.asx.com.au/content/dam/asx/investors/investment-options/shares-course-11-technical-analysis.pdf.

[9] Khanh Quoc Nguyen. Why are cryptocurrencies so volatile?, June 2024. https://acfr.aut.ac.nz/__data/assets/pdf_file/0004/926140/Why-are-cryptocurrencies-so-volatile-03-Jul2024.pdf.

[10] Christopher Olah. Understanding lstm networks, 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[11] Gordon Scott. End of day order: How it works, advantages, 2023. https://www.investopedia.com/terms/e/end-of-day-order.asp.

[12] Cole Stryker. What is a recurrent neural network?, 2024. https://www.ibm.com/topics/recurrent-neural-networks.

[13] Tristan Yates. 4 ways to predict market performance, October 2024. https://www.investopedia.com/articles/07/mean_reversion_martingale.asp.

[14] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. https://D2L.ai.