An Investigation Into Recurrent Neural Networks and Stock Price Forecasting

Brian Drake
University of Adelaide
THE UNIVERSITY OF ADELAIDE 5005 AUSTRALIA
brian.drake@student.adelaide.edu.au

Abstract

Stock price prediction remains a challenge within financial analytics. This study reviews current Recurrent Neural Networks (RNNs), including SimpleRNN, Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU) architectures, to predict Google Stock prices using historical time series data. The dataset used was sourced from Kaggle, encompassing 1258 data points from the 3rd of January 2012 to the 30th of December 2016. Each datapoint was characterised by Open, High, Low, Close, and Volume metrics. Of which 2 years (730 days) of datapoints with a 30 day context were used to predict 3 days into the future. They were preprocessed and the three model types were trained tested.

Model performance was evaluated using metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and R^2. Comparative analysis revealed that the basic stacked GRU performed best, achieving lower error rates and greater predictive accuracy. Further refinement through hyperparameter tuning, regularisation techniques, and feature addition were completed, although additional complexity did not yield better results.

The experimental results highlight the strengths and limitations of RNNs in financial forecasting. While the GRU model successfully predicted stock prices with reasonable accuracy, it struggled to replicate abrupt market fluctuations, particularly in Volume predictions. Thus, providing insights into model optimisation strategies but suggesting an RNN informed by ARIMA model for future research.

1. Introduction

Stock price prediction is a fundamental challenge in the financial industry. Garnering mass attention due to the potential to inform investment strategies and mitigate financial risk. Stock prices are influenced by various factors, such as market sentiment, economic indicators, and historical trends. Predicting these prices accurately requires models capable of capturing complex temporal relationships and sequential patterns inherent in time-series data [15].

To address this challenge, this paper employs Recurrent Neural Networks (RNNs), inclusive of Simple RNN, Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU), to predict Google stock prices. This paper will therefore be using RNNs to create a regressive model of a dataset to predict up to three days in the future. The dataset used for this, was provided by Sah via Kaggle [13]. It contains 1258 stock market datapoints in a date range of the 3rd of Janurary 2012 to the 30th of December 2016. Each datapoint has features Open, High, Low, Close, and Volume. This dataset was preprocessed and split into training, validation, and testing subsets, with sliding windows of historical data used as input to forecast future prices.

RNNs are particularly suited for time-series prediction due to their ability to maintain information across time steps. GRUs and LSTMs extend on this capability through their additional gate architecures, mitigating issues such as exploding and vanishing gradients [10]. In turn, causing them to be strong candidates. This paper compares these architectures based on their predictive performance through key metrics: Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R^2.

Through experimental analysis, this study not only aims to predict stock prices but also explore the effectiveness of hyperparameter tuning, architecture design, and regularisation techniques in optimising model performance. Shedding light on the practical applications and limitations of RNN-based models in the domain of financial forecasting.

2. Method

RNNs are a class of deep learning models that are designed to accept sequential data. Unlike other feedforward neural networks, RNNs possess the unique feature of retaining a memory of previous datapoint patterns in a hidden or internal state. Creating a system that ideally processes natural language, speech recognition, and time series forecasting [10]. Recent advances in transofrmers have enabled RNNs to be utilised for many more tasks, such as multi-

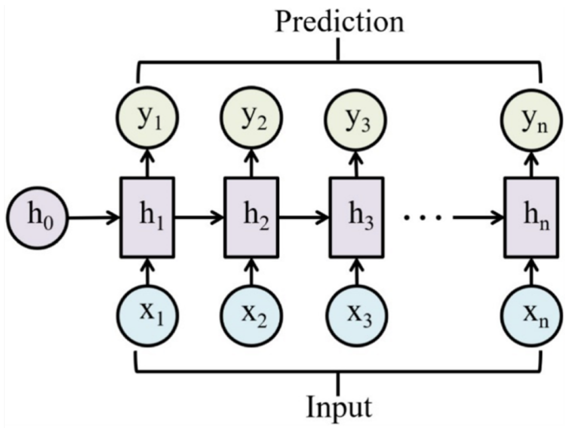

Figure 1. Generic RNN structure reflective of SimpleRNN [10]. Where h_n is the hidden state, x_n is the input tensor, and the y_n is the output tensor.

modal learning and real time decision making systems.

2.1. Simple RNN

SimpleRNN is a basic RNN structure offered by Keras. It is a deep learning network that takes in sequential time series data and then outputs either the final state or the sequence of hidden states. This is achieved recursively, whereby each time step and input are computed alongside the previous internal state. A weighted sum followed by an activation function, typically tanh, is also utilised to assist in non-linear pattern recognition [5]. Mathematically, this is represented as:

$$h_t = activation(W \cdot x_t + U \cdot h_t - 1 + b)$$

Where W is the weight matrix for the input, U is the weight matrix for the recurrent connection, and b is the bias term [5]. The structure of this can be seen in figure 1.

A limitation of SimpleRNN is the vanishing gradient effect. This occurs due to the single unit of internal state. As training progresses, this state is updated alonside every datapoint. This can cause the value to approach 0 and cause no learning to occur, minimising or halting learning [11].

2.2. LSTM: Long-Short Term Memory

LSTM RNN provides a more complicated approach to RNNs. Rather than only having one previous hidden state that is recursively updated as each new input is calculated, it contains a memory cell and three gates to control information flow. The memory cell stores information across long sequences, while the gates control input, what is discarded, and output. The gates are called the input gate, the forget gate, and the output gate, respectively [3] [5]. The structure of which can be seen in figure 2. The architecture is important for handling long term dependencies but is also the

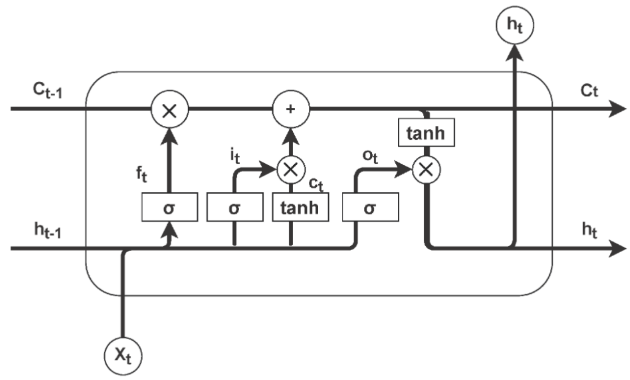

Figure 2. Simplified LSTM diagram [10]. Where i_t is the input gate, f_t is the forget gate, o_t is the output gate, g_t is the cell input, c_t is the cell state, and h_t is the hidden state. The input gate controls how much of the new input is written to the cell state. The forget gate decides how much of the previous cell state should be retained. The output gate determines how much of the cell state is used to compute the hidden state.

most computationally expensiven of all three models proposed.

2.3. GRU: Gated Recurrent Unit

GRU RNNs are another variant designed to address the vanishing gradient problem experienced in SimpleRNN while also simplifying the more complex LSTM architecture. Achieved by combining the forget and input gates into a single update gate and merging the cell state with the hidden state, reducing the number of gates and parameters [1]. The GRU therefore consists of two gates: the update gate z_t and the reset gate r_t. This can be seen in figure 3. Reduced parameters whilst also maintaining the ability to capture long term dependencies create a competitive model. Making GRU preferable for tasks that have limited datasets and computational resources. Further to this, GRU and LSTM typically have comparable performance dependent on the task.

2.4. Activation Functions, Bidirectionalism, and Dense Layers

Two popular activation functions exist for RNNs, ReLu and tanh. ReLu (Rectified Linear Unit) is an element wise function that takes the maximum of either 0 or the input tensor [5]. Mathematically represented as:

$$\text{ReLu} = max(0, X)$$

The tanh (hyperbolic tangent) activation function is an element wise function that scales data between -1 and 1 in a hyperbolic curve [5]. This is represented mathematically as:

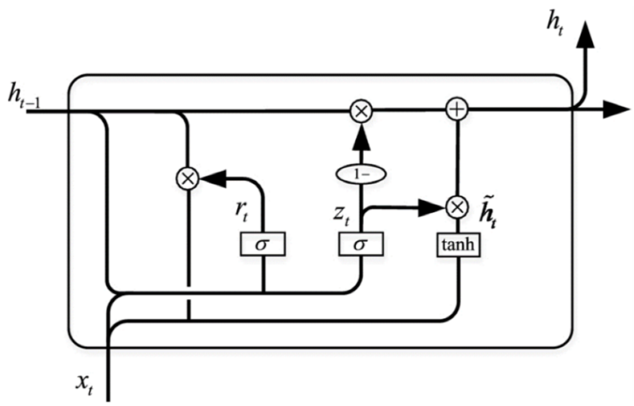

Figure 3. A diagram displaying the GRU architecture [10] Where z_t is the update gate, r_t is the reset gate, h_t is the hidden state, and x_t is the input.

$$\tanh(x) = sinh(x)/cosh(x)$$

These activation functions are used to create non-linear relationships between datapoints for the RNN to learn from. Where dependent on the dataset and the specific layers, either function can be utilised. For most RNN tasks, tanh is commonly employed due to the range being [-1, 1], reducing both the vanishing and exploding gradient effect.

Bidirectionalism can be employed for multi-layered RNNs. This occurs when the RNN is extended to process the sequence both forward and backward. Facilitating the network to capture context from both the past and the future, enhancing its ability to understand dependencies in more detail. This is achieved by creating two hidden states per time step, one for a forward pass and one for a backward pass [10]. Therefore, when using a multi-layered RNN structure, the final RNN layer cannot be bidirectional.

Typically, the output layer of a RNN is constituent of a dense layer. This is a layer where each neuron introduced connects to every other neuron/output state of the preceding layer. This structure enables dense layers to integrate information learned from previous layers and form complex pattern recognition. Each neuron in a dense layer has an associated weight and activation function, allowing data transformation and interpretation [9]. For the stacked structure employed in this paper, dense layers are used for the interpretation of passed hidden state data [14]. This also enables the models to output the regression based predictions.

2.5. Regularisation

Scaling is an important step for datasets before training in deep learning. Due to three main reasons: data representation, calculation time, and the exploding gradient effect. When models process data, the numerical values can vary in magnitude between features. If left unscaled, this can cause higher magnitude features to become over represented due to their relative size rather than impact. The inverse is also true; a lower magnitude feature can become under represented despite having a large impact [12]. If these numbers are very large, it can cause quite long and computationally expensive epoch training times. Further, if these numbers remain large, then the gradients calculated can experience the exploding gradient effect. Where gradients are calculating larger, causing the model to become unstable and therefore unable to learn effectively [11]. To mitigate these factors, MinMaxScaler was used to scale and translate features individually so that their range is [0, 1] [12]. Therefore regularising a the model by reducing feature misrepresentation, computation time, and the exploding gradient effect.

Dropout is another regularisation technique, occurring as a functional layer during training. This layer causes some elements of the input tensor to be randomly changed to zeroes. Where each node has an independent probability of being changed, as specified in the architecture [9]. Depending on the model, not only can different probabilities be passed, but multiple occurrences of dropout can be used. Dropout is intended to improve regularisation by preventing the co-adaptation of individual nodes [2]. Meaning that increased complexity of the model will be less likely to create redundant or silent nodes. In turn allowing for greater generalisation on new data.

L2 regularisaton is a method by which the overall model complexity has reduced impact on the loss being calculated during training. Where high regularisation is employed to reduce overfitting through normalisation of training losses. Causing regularisation to enact strongly on larger weight values but weaker on smaller values. This causes weights to trend toward but never reach 0 [5]. This is mathematically represented as:

$$\text{L}_2\text{Regularisation} = w_1^2 + w_2^2 + \ldots + w_n^2$$

2.6. Early Stopping

Early stopping is a callback method whereby a metric's performance is monitored during training. If the metric does not improve after a specified amount of epochs, then training stops [5]. This function can also roll back training to the epoch where the lowest metric was observed or remain where it has stopped. The target metric can vary as required, but typically validation loss is used. Validation loss was the metric used for early stopping in this study.

2.7. Performance Metrics

Multiple metrics were used to monitor model performance. Including Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and the Coefficient of Determination (R^2) [12].

The MAE measures the average magnitude of errors, calculated as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Where y_i is the actual value, and $\hat{y}_i$ is the predicted value. MAE ideally would approach 0, indicating high model prediciton accuracy [12]. Related to MAE, the MAPE evaluates error as a percentage of actual values. This metric emphasises proportional errors and would optimally also approach 0 [12]. Using the same nomenclature as MAE, this is defined as:

$$\text{MAPE} = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

MSE is a metric that focuses on penalising larger errors [12]. Mathematically expressed as:

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

While its square root is the RMSE. Providing error magnitude in the same units as the data, enabling clarity in interpretation [12]. This is calculated as:

$$\text{RMSE} = \sqrt{\text{MSE}}$$

The R^2 value is calculated as below.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

R^2 assesses the proportion of variance explained by the model. Ideally, values closer to 1 indicate a better fit [12]. Although values can be negative if the model performs worse than a simple mean based prediction.

2.8. Hyperband Tuner Optimisation

The hyperband tuner is an optimisation algorithm designed to efficiently tune hyperparameters through the application of random search and early stopping. It works by evaluating multiple hyperparameter configurations over varying numbers of epochs to identify promising candidates without fully training all configurations. Hyperband organises the process into iterations, allocating resources to many configurations initially and gradually focusing on the most performant. It uses a successive halving strategy, where poorly performing configurations are terminated early and the resources are reallocated to better performing ones. This approach enables faster convergence of optimal hyperparamters compared to a grid or random search [8].

Within the hyperband optimisation many hyperparameters can be tuned, inclusive of dropout, hidden state unit size, and activation functions. It can also be used to search for optimisers. RMSprop (Root Mean Square Propogation), SGD (Stochastic Gradient Descent), and Adam (Adaptive Moment Estimation) were candidates in this study. SGD updates model parameters iteratively by computing gradients of the loss function with respect to parameters and taking steps in the opposite direction [9]. While simple, SGD can struggle with convergence due to oscillations in its updates. RMSprop addresses this by maintaining a moving average of squared gradients for each parameter, enabling adaptive learning rates that slow down updates for frequently updated parameters and accelerate updates for less frequently updated ones [9]. Adam builds on RMSprop by incorporating both momentum (to stabilise updates) and adaptive learning rates. It combines the benefits of SGD and RMSprop, making it robust and suitable for a wide range of problems [9]. Adam was used in all models and found to be the most favourable after hyperband tuning.

Adam works granularly by using adaptive learning rates for each parameter by maintaining two moving averages [9]. The first moment is the gradient and the second moment is the squared gradient. Mathematically this is expressed as follows.

$$m_t = \beta_1 m_(t-1) + (1-\beta_1)\nabla L(w)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla L(w))^2$$

$$(m_t) = \frac{m_t}{1-\beta_1^t}, (v_t) = \frac{v_t}{1-\beta_2^t}$$

$$w = w - \eta\frac{m_t}{\sqrt{v_t}+\epsilon}$$

Where m_t is the first moment and v_t is the second moment. β_1 and β_2 are decay rates where, β_1 is usually equal to 0.9 and β_2 is usually equal to 0.999. Finally, ϵ is a small constant added to prevent division by zero error.

3. Experimental Analysis

The experimental analysis was conducted using python, alongside the Kaggle Google stock price training dataset. The provided test dataset was unused due to the scope of the study pertaining to 1-3 day future predictions rather than a 1 year forecast. The training dataset was read from a CSV (Comma Seperated Value file type) into a DataFrame. This was then visually scanned for inconsistencies. It was noted that timepoints from 03Jan2012 until 26Mar2014 contained Close values that were double the next day's Open value. To remove as much data as possible while maintaining coherency, the latest 2 years of data with a 30 day sliding window were selected. Of this subset, the first 65 values contained the erroneous values. They were maintained to provide more complicated relationships for the model to learn from enabling robustness and data completeness. The intact dataset was then preprocessed. The dates were converted into a datetime format and the numerical values, Open, High, Low, Close, and Volume, were all sanitised from strings into floats. Each of these features were then scaled using sklearn.preprocessing MinMaxScaler. The default parameters for this function were used, causing the dimensions to be independently scaled between 0 and 1.

Four helper functions were then developed to complete the analysis. They were create_dataset, predict_future, plot_predictions and evaluate_model. The function create_dataset transforms the dataset from a 2D array into a 3D format that is compatible with RNNs. It assumes the data is an array or dataframe with dimensions as [samples, features] and sorted in chronological order; it then outputs two tuples, X_train and y_traion, in the format of (day, sliding window, features). The function predict_future predicts future values for a given number of days for the model passed, outputting a list of predicted value arrays for each day. The plot_predictions function plots predicted values against true values using a list of feature labels as labels for the plots, outputting a 2 x 3 plot of features. Finally, the evaluate_model funciton calculates the MSE, RMSE, MAE and R^2 values for a set of predictions against the true values, outputting a tuple in the same order.

The now scaled training set was then split, such that the final 3 days would become a prediction comparison set, while the rest was retained. The retained data was then processed by the helper function, create_dataset, to make X_train and y_train subsets. Creating the final training dataset, consistent of 2 years (730 days) of datapoints each with 30 days of context.

TensorFlow was used to develop a stacked model that has architecture compatible and comparable between SimpleRNN, LSTM, and GRU. The specific structure can be seen in figure 4, this strucutre waas based on a stacked LSTM architecture [14]. Tanh activation functions were used in all layers to maintain comparability and enable non-linear data relationships to be explored. The first two layers had return sequences enabled, as this option is only required in intermediate layers, as they are only required for learning. All layers had 20% dropout to introduce regularisation. The layer units were made to be equal to the quantity of historical datapoints so that the mapping of temporal patterns is in the same dimensionality of learning. Scaling the learning and complexity as required.

An early stopping function via Keras was implemented that monitors validation loss for improvements over 10 epochs; if none are observed, then the model rolls back to the optimal epoch and discontinues training. This was implemented to reduce overfitting due to the size of the dataset. All models were compiled to optimise loss with

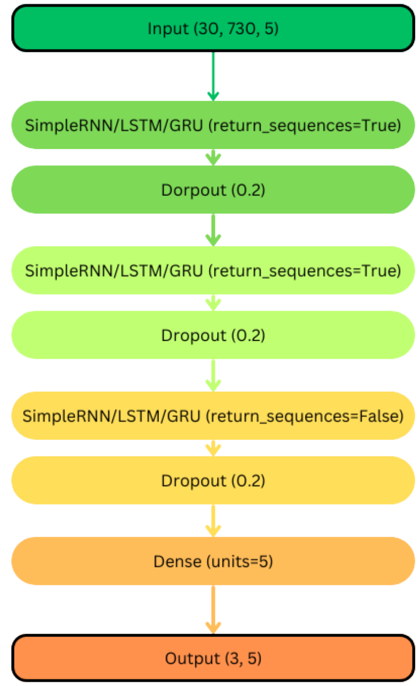

Figure 4. The standard architecture used for all three base models, SimpleRNN, LSTM, and GRU.

a MSE calculation using the Adam optimiser. In addition, MAE and MAPE metrics were used to observe performance due to the task being framed as a regression task. The model method .fit() has a keyword argument that was utilised to create an active 20% validation split during training. The results of the base model comparison are as seen in figure 1.

GRU was the best performing base model. To explore this further, a feature addition study was performed. Where complexity was slowly built upon the model and assessed to find an optimal architecture. The results of the 12 trials are as per figure 2. From these trials, additional complexity of any variety was found to be detrimental to the model's performance.

To supplement optimisation of the best model, the original GRU model, a Keras Hyperband was used. Variance was provided in the quantity of unit size (0.5, 1, 1.5 and 2 times the amount of temporal data points), activation function (tanh and ReLu), drouput rate (0.2, 0.3, 0.4, 0.5),

RNN Model	Epoch Count	MSE Score	RMSE Score	MAE Score	R2 Score
Simple	17	1.43	1.20	1.06	-16000
LSTM	51	0.0541	0.233	0.210	-620
GRU	23	0.0138	0.117	0.0983	-154
Hyperband GRU	24	0.0756	0.275	0.230	-851

Table 1. Performance metrics for different RNN models.

Model Trial	Epoch Count	R2 Score	Feature Changes
01	10	-4010	All layers `tanh` converted to `ReLU`
02	10	-2010	Bidirectionality added to the first layer, building on 01
03	10	-2690	`L2` regularisation at 0.005 added to all layers, building on 02
04	10	-1720	An additional dense layer was added after GRU layers, where dense=max(10, 0.5*past_days), building on 03
05	10	-2710	Unit quantity was multiplied by 1.5 in the first GRU layer, then 1.25 in the second GRU layer, building on 04
06	10	-1350	All layers changed back to `tanh`, building off 04
07	10	-1120	`L2` regularisation reduced to 0.001, building off 06
08	10	-1770	`L2` regularisation removed, building from 07
09	10	-1240	Third GRU layer removed, additional dense layer retained, building from 01
10	10	-872	`L2` regularisation at 0.001 added to all layers, building from 09
11	10	-1190	`L2` regularisation increased to 0.005, building from 10
12	10	-1290	Bidirectionality added to the first layer, building on 11

Table 2. Performance metrics and feature changes for the feature addition study.

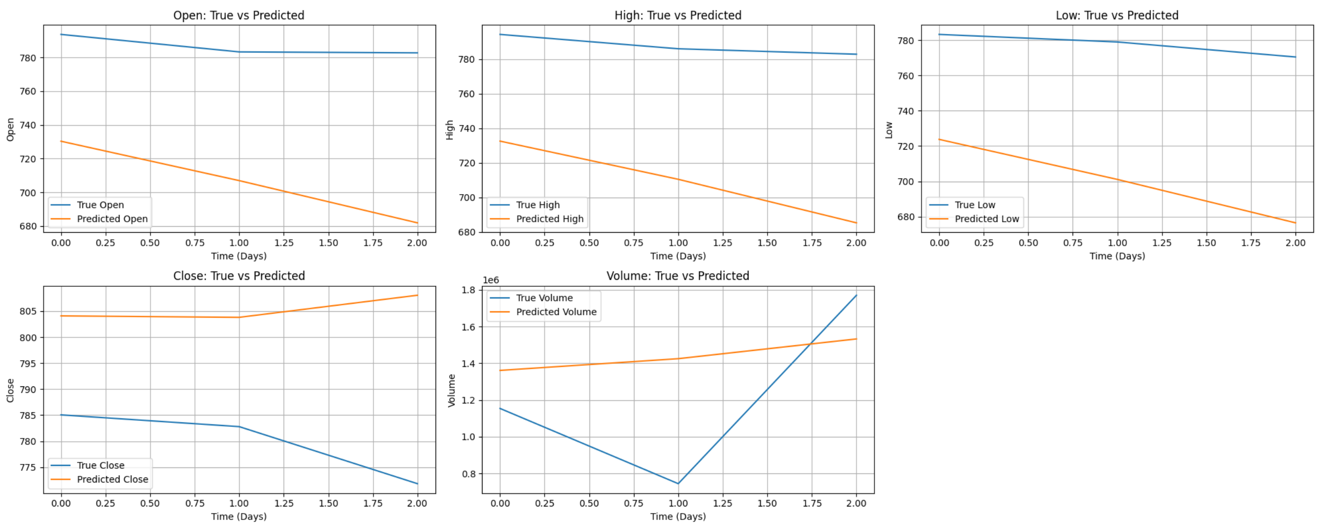

Figure 5. 3 day future predictions against the 3 day true testing set, using the original GRU model.

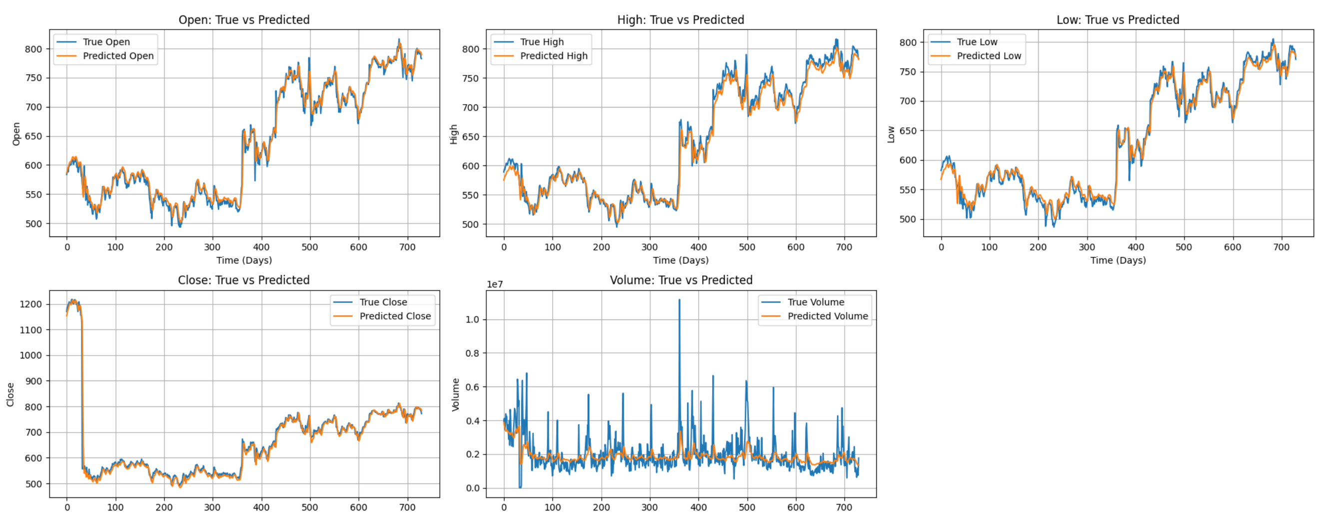

Figure 6. Predicted training values against true values, as made by the original GRU model.

and optimisers (Adam, RMSProp, and SGD). Where each feature was tuned at every instance within the architecture. The Hyperband was allowed up to 25 epochs to choose between features, based on the original model stopping at 23 epochs. The resultant optimised model was then trained, displaying an overall worse performance than the original model, as seen in figure 1.

The original GRU model, performing the best, was then given the original training data to make predictions. The outcome can be seen in figure 6 This task intends to characterise prediction patterns against true data. It can be seen that even though the model has learned to predict quite well with Open, High, Low, and Close, the model is having trouble capturing Volume. The peaks and troughs, although aligned in location, do not reflect the intensity. The model is more so predicting a local average for Volume. The model is also largely able to match all features, but is unable to emulate the sharp spikes that come with market data. This predictive behaviour is consistent when compared to the 3 day future testing set, as seen in figure 5. The model consistently under predicts the Open, High and Low, while over predicting the Close.

4. Code

Please find the code here.

5. Conclusion

Between the SimpleRNN, LSTM, and GRU experimental architectures proposed, GRU performed the best. This can be seen most clearly using R_2 scores. Where GRU achieved the lowest score of -154, followed by the optimised GRU at -295, then the LSTM at -620, and finally the SimpleRNN at -16000. The stacked structure, despite its simplicity, displays a greater aptitude for stock price learning. Even with a feature addition study and hyperband tuning, the original stacked GRU model was the most performant. This is likely due to the hidden units being proportional to the temporal data being provided. From the feature performance graphs in figure 6, it appears that the GRU model is able to predict peaks and troughs in the data to nearly the same intensity of which they are recorded. This is hindered in the Volume feature, where the model largely estimates a local average Volume value rather than the true Volume. From the 3 day forecasting data, as seen in figure 5, this behaviour is consistent. The model consistently underpredicts the Open, High and Low, while overpredicting the Close. Suggesting that the data is generalising well but is not accurate enough to form a confident investment strategy.

From this outcome, it can be seen that the GRU RNN although the best model derived, is still limited in accuracy. Further development could be performed with a larger ad more complete dataset to create a more acurate model. Excluding the Volume feature due to its speculative nature could provide an avenue for further refinement or be retained alongside a web crawler feeding data into a sentiment analysis Large Language Model (LLM).

A futher limitation of RNNs in the financial domain is incrementally trainign the model. Being that stock data is updated daily, creating more data to learn from, incremental training would be required to maintain predictive accuracy. This process can be computationally expensive, more so when implemented long term. Further, the model can experience catastrophic forgetting, where it may forget patterns learned from the original data [6].

Current investment techniques utilise ARIMA (AutoRegresive Intergrated Moving Average), a statistical modelling technique. This is largely used for time series forecasting by modelling the relationship between an observation and the residual errors from a moving average model. This technique is particularly effective for data that exhibits patterns over time, such as trends or seasonality [4]. According to current research [7], ARIMA outperforms LSTM models; applying the basis of ARIMA to a model could provide more research opportunities in machine learning.

References

[1] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014. 2

[2] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012. 3

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997. 2

[4] Mohammad Rafiqul Islam and Nguyet Nguyen. Comparison of financial models for stock price prediction. *Journal of Risk and Financial Management*, 13(8), 2020. 7

[5] Keras Community. *Keras Documentation*, 2015. Accessed: December 4, 2024. 2, 3

[6] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, Mar. 2017. 7

[7] J. P. Senthil Kumar, R. Sundar, A. Ravi, Srivatsa Kumar B R, Sandhya S. Pai, and Baiju T. Comparison of stock market prediction performance of arima and rnn-lstm model – a case study on indian stock exchange. In *AIP Conference Proceedings*, volume 2875, Melville, 2023. American Institute of Physics. 7

[8] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel

bandit-based approach to hyperparameter optimization. *arXiv (Cornell University)*, 2016. 4

[9] The Linux Foundation. *PyTorch 2.4 Documentation*, 2023. Accessed: September 12, 2024. 3, 4

[10] Ibomoiye Domor Mienye, Theo G. Swart, and George Obaido. Recurrent neural networks: A comprehensive review of architectures, variants, and applications. *Information*, 15(9), 2024. 1, 2, 3

[11] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013. 2, 3

[12] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 3, 4

[13] Rahul Sah. Google stock price dataset, 2024. 1

[14] Ayesha Sahar and Dongsoo Han. An lstm-based indoor positioning method using wi-fi signals. *ICVISP*, pages 1–5, 08 2018. 3, 5

[15] Pan Tang, Cheng Tang, and Keren Wang. Stock movement prediction: A multi-input lstm approach. *Journal of forecasting*, 43(5):1199–1211, 2024. 1