RNN Ensemble For Stock Market Price Prediction

Jonathan Markey - 1805363

Introduction

This report explores the development and evaluation of an ensemble Recurrent Neural Network (RNN) model tailored for predicting Apple's stock price. The core purpose of this report is to investigate whether an ensemble approach, integrating various RNN variants, can amplify the predictive accuracy for stock prices, particularly through collectively capturing short, medium, and long-term dependencies.

RNNs and their advanced variants, Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM) networks are neural networks that specialise in processing sequential data, making them suited for time-series analysis like stock market forecasting [1]. Each variant, however, comes with its inherent limitations. Standard RNNs, while efficient in handling short-term dependencies, grapple with long-term data due to issues like vanishing and exploding gradients. GRUs offer a more balanced approach to processing medium-term sequences, but may still falter with very long or short data sequences. LSTMs excel in long-term dependency recognition, but can be prone to overfitting when dealing with shorter sequences.

The ensemble method aims to counteract each model's limitations by utilising a Single-Layer Perceptron (SLP) to integrate the strengths of the individual models while covering for their individual weaknesses [2]. Fifteen RNN models, based on the three distinct RNN variants predicting for on five different sequence lengths, were developed to capture a wide spectrum of temporal patterns in the stock data. This methodology is posited to enhance the overall predictive power and offer a more nuanced understanding of the varying dynamics in stock price movements. The subsequent sections of the report will detail the architectural nuances of each RNN variant, the rationale behind their selection, the process of constructing the ensemble model, and the analysis of the performance compared to the base models.

In the context of stock market price prediction, RNNs face competition from architectures like Convolutional Neural Networks (CNNs) and Transformer-based models. CNNs, a type of feedforward neural network, utilise convolutional layers to process data, primarily grid-like in nature. They function by applying filters across input data to extract patterns, and then employ fully connected layers to interpret these patterns [3]. However, adapting CNNs for linear time-series data like stock prices can be challenging, and they may struggle with capturing long-range temporal dependencies. In contrast, RNNs are inherently designed for sequential data, adept at processing and retaining information across time steps. Transformers, offering a more complex structure, excel in handling long-range dependencies through self-attention mechanisms. These mechanisms, coupled with the ability to process sequential data in parallel, often make Transformers more effective for identifying long-term dependencies compared to RNN models [4]. However, the complexity and resource intensity of Transformers restrict their practicality, making them more suited for scenarios involving extensive sequential data analysis.

Method Description

The neural network architectures implemented in this report centres around the Recurrent Neural Network (RNN). RNNs are a type of neural network that specialise in processing sequential datasets. The main idea behind RNNs is that the neural network allows for previous outputs to be used as inputs while having hidden states.

The architecture of an RNN model consists of three parts, the input layer the hidden layers and the output layer. The first part of the RNN model architecture is the input layer. The primary function of the input layer is to convert the input data into a format suitable for use in the RNN. Unlike other neural networks, the RNN model requires that the input data be processed into chronological sequences. The chronological sequences are important as are used to create the memory of the model in the hidden layer.

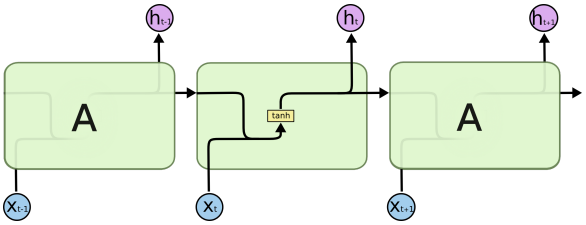


Figure 1: RNN Hidden State Architecture [5]

The hidden layer is the main mechanism that distinguishes the RNN model from other neural networks. The hidden layer serving as the model's memory for recognising patterns in sequential data. Within this layer, multiple nodes process input data sequentially. Each node updates its hidden state, the output reflecting learned information from the sequence, based on the current input and the previous node's state [6]. This hidden state output is then passed to the next node alongside the next input data in the sequence, enabling the RNN to maintain a continuous memory of the sequence's context. This flowing process through nodes allows the RNN to retain contextual information from previous events, crucial for its predictive and analytical capabilities.

$$h_t = tanh(x_t W_{ih} + b_{ih} + h_{t-1} W_{hh} + b_{hh})$$

Equation 1: RNN Update Function [7]

The RNN update function can be seen in Equation 1. The variable h_t represents the output for the current hidden state at time t. This vector is crucial as it encapsulates information learned from previous data sequences up to the current point, embodying the RNN's core functionality. This variable not only serves as the output at each time step, but it also used as a component in ongoing computations, where it is fed forward to influence subsequent states. This mechanism is denoted in the equation as h_{t-1}, which is the output of the previous state. A key aspect to state vectors is their dimensionality, referred to as the hidden size. The hidden size is independent of the dimensions of the input data, with the dimensions being set prior to model generation. For example, a hidden size of 32 results in the states vectors of length 32 at each time step. This ensures consistent processing capability across varying input sizes and highlighting the RNN's adaptable architecture for diverse sequential data. x_t is a vector containing the feature input data at time t in a sequence. The W_{ih} and b_{ih} represent the learnable weights and biases for the input data x_t. The W_{hh} and b_{hh} represent the learnable weights and biases for the input data h_{t-1}.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Equation 2: Tanh Activation Function [8]

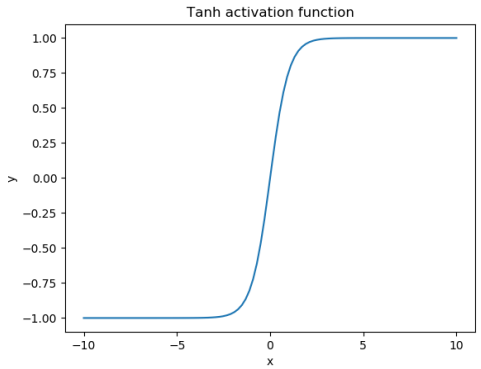


Figure 2: Tanh Activation Function [8]

The activation function used in the RNN is typically the hyperbolic tangent ($tanh$) function [7]. The tanh function's aim is to normalise the hidden state's output between -1 and 1 in a S-shaped curve. As seen in Equation 2, it represents the difference between the exponential growth (e^x) and decay (e^{-x}) of the input, divided by their sum. As x moves away from zero, the tanh function smoothly transitions, effectively normalising the RNN's output and introducing the non-linearity essential for learning complex patterns in the data.

$$y_t = h_t W_{hy} + b_y$$

Equation 3: Output Layer Linear Transformation [7]

The output layer is the final layer of in the RNN model. The output layer is a dense (fully connected) layer that is used to generate the output and can be seen in Equation 3. The variable y_t is equal to the output. Like with the hidden size, the output dimensions are set when generating the model. In the context of stock market price prediction, the dimensions of the output would be a single dimension and would be equal to the stock price for the next time $t + 1$ in the sequence. The variables W_{hy} and b_y represent the learnable weights and biases for the output of the hidden state h_t.

The main limitation of the RNN model is with its short term memory. The recurrent nature of the hidden state can cause vanishing and exploding gradients, effecting the impact of long range dependencies in the early elements of a sequence. If the sequence of data is too long, the multiplicative nature of the gradient updates can cause the vanishing gradients and information learned in earlier sequences can be lost. Vanishing gradients is an issue where the gradients shrink when calculated during backpropagation. This is an issue as it can cause the training process to stagnate with the model being unable to significantly update enough to escape local minimas. Exploding gradients are where the gradient values become too large, and the model cannot converge toward the minimas, resulting in volatile loss values during training. The solution to this problem comes from the Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) models. These models are variants to the RNN model that aim to overcome these issues. These models are variants in that they only change the architecture used in the hidden layer.

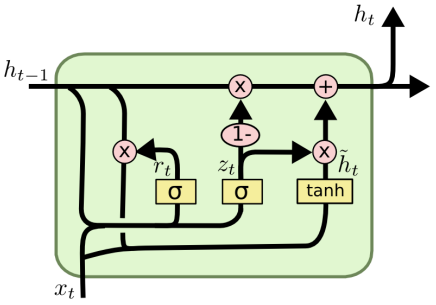


Figure 3: GRU Hidden Layer Node Architecture [5]

The GRU is a variant of the RNN model that aims to solve the gradient problem through introducing mechanisms called gates into the architecture [6].

$$r_t = \sigma(x_t W_{ir} + b_{ir} + h_{t-1} W_{hr} + b_{hr})$$
$$z_t = \sigma(x_t W_{iz} + b_{iz} + h_{t-1} W_{hz} + b_{hz})$$
$$\tilde{h_t} = tanh(x_t W_{in} + b_{in} + r_t(h_{t-1} W_{hn} + b_{hn}))$$
$$h_t = h_{t-1}(1 - z_t) + \tilde{h_t} z_t$$

Equation 4: GRU Equations [7]

The reset gate r_t and the update gate z_t are mechanisms that are used to regulate how much of the previous hidden states information (h_{t-1}) will be used in the current state and passed on to the future state. The gates achieve this through executing equations similar to the RNN update function (Equation 1) with the exception that a sigmoid activation function is used instead of the tanh activation function [7].

.

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Equation 5: Sigmoid Activation Function [8]

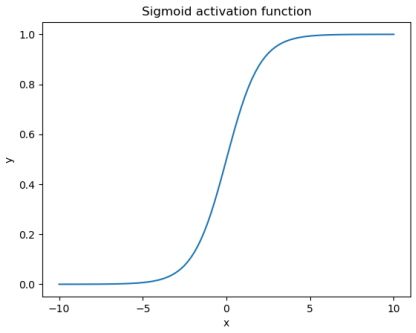


Figure 4: Sigmoid Activation Function [8]

The primary purpose of the sigmoid activation function is to introduce non-linearity into the neural network by mapping input values to a range between 0 and 1, following an S-shaped curve. The function is expressed in Equation 5 and is equal to 1 divided by 1 plus the exponential decay. Unlike the tanh function, which centres towards 0, the sigmoid function centres towards 0.5 with negative x values causing the function to converge towards 0 while positive values for x cause the function to converge towards 1.

The reset gate regulates how much past information needs to be used in the current hidden state while the update gate regulates how much past information needs to be recurrently forwarded to the next state. The sigmoid activation function achieves this as elements closer to 0 will be weighted less when calculating the current hidden state and candidate hidden state.

The candidate hidden state $\tilde{h_t}$ represents the proposed combination of the current sequence input and the output of the reset gate. The reset gate effects this state as resulting output can cause the previous state's values be weighted less if the value is closer to 0, or weighted more if the value is closer to 1. This means that a greater level of information is carried forward if the reset gate's value is closer to 1. In the equation for $\tilde{h_t}$, this is represented by multiplying the linear combination of the previous hidden state by reset gate's value. The current hidden state h_t represents the actual output of the GRU node. This state is equal to the sum of previous hidden state multiplied by 1 minus the update gate, and the candidate hidden state multiplied by the update gate. If the update gate has a value closer to 1 the candidate hidden state is weighted more but if the value is closer to 0, then the previous hidden state is weighted more in the current hidden state.

This weighting solves the short-term memory issue of the RNN as the adaptive weighting of previous hidden states allows for longer range dependencies to be maintained and be recurrently passed forward. The gating mechanism also helps to solve the gradient problems as it allows for a pathway for the gradients to flow further backwards in time without diminishing or exploding.

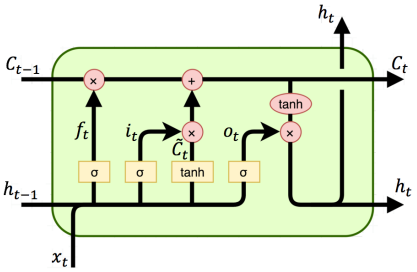


Figure 5: LSTM Hidden Layer Node Architecture [5]

LSTM is another variation to the hidden state node and is the most complex. Like with the GRU, the LSTM aims to improve the RNN model by introducing gates. Unlike the previous model's node architecture, the LSTM actually outputs two sets of data, the hidden state and the cell state [9].

$$f_t = \sigma(x_t W_{if} + b_{if} + h_{t-1} W_{hf} + b_{hf})$$
$$i_t = \sigma(x_t W_{ii} + b_{ii} + h_{t-1} W_{hi} + b_{hi})$$
$$\tilde{C_t} = tanh(x_t W_{ic} + b_{ic} + h_{t-1} W_{hc} + b_{hc})$$
$$o_t = \sigma(x_t W_{io} + b_{io} + h_{t-1} W_{ho} + b_{ho})$$
$$C_t = f_t C_{t-1} + i_t \tilde{C_t}$$
$$h_t = tanh(C_t) o_t$$

Equation 6: LSTM Equations [7]

The equations for the function used in the LSTM can be seen in Equation 6. Although LSTM contains multiple functions, the functions themselves are very similar to those used in the GRU and RNN. The gates used in the LSTM are the forget f_t, input i_t, and output o_t gates. These gates share a similar form to the reset and update gates used in the GRU. The cell state C_t acts as the LSTM's memory, carrying information through the hidden layer by only changing minimally as it passes through the node sequence. The forget gate makes the first alteration to the cell state by multiplying the state by the gate's sigmoid output. This functions to minimise the cell gate's information when the output is closer to 0 or retain the information when the output is closer to 1.

The next step in the process involved updating the cell state to include any information that we want to carry forward from this point sequence. The update cell state $\tilde{C_t}$ shares the same formula as the original hidden state update function seen in Equation 1. As with the forget gate, the magnitude of the update to the cell state is determined by the sigmoid output of the input gate. The update cell state is multiplied by the input gate, and then the product is added to the cell state.

The next step in the LSTM is to calculate the actual hidden state output h_t. This is achieved through multiplying the sigmoid output of the output gate by the tanh adjusted cell state. The LSTM model solves the memory issue of the RNN, as the hidden state value is not calculated based on the sequence data, but rather is calculated more-so based on the cell state. This is beneficial as the cell state serves as an aggregate of all the important information and dependencies identified in the sequenced data. The cell state and gating mechanisms also solve the vanishing and exploding gradient issues, as it allows for the gradients to flow backward in time more smoothly without diminishing or exploding.

The distinctive aspect shared across all RNN architectures is the use of Backpropagation Through Time (BPTT). BPTT is an adaptation of the standard backpropagation algorithm, which is commonly used in feedforward neural networks. In standard backpropagation, the chain rule is employed to calculate gradients with respect to weights and biases. However, RNNs are uniquely designed to handle sequential data, leading to dependencies between the nodes across different time steps. Unlike feedforward networks, where each input is processed independently, RNNs maintain a temporal state influenced by previous inputs. Consequently, BPTT takes into account the sequential dependencies by unfolding the RNN across time steps. This unfolding transforms the recurrent connections into a chain-like structure, allowing the algorithm to back propagate the error through time [10]. It considers how changes in previous states affect the current state, a critical aspect distinguishing BPTT from regular backpropagation used in networks without temporal dynamics, such as standard feedforward neural networks.

The ensemble method is based on the method of stacked generalisation [2]. Stacked generalisation is an ensemble method that learns how to best combine the predictions from multiple existing neural networks. The reason an ensemble method is used is related to the limitations of the RNN variants [11]. As explained earlier, the main issue with the RNN model is that it has difficulty establishing long-term dependencies due to its gradient calculation issues and multiplicative nature. The GRU helps to reduce this issue, but while it has a longer-term memory, it is still only effective up to a certain sequence length. The LSTM is far more effective at determining long-term dependencies, but its reliance on the cell state can cause overfitting when it is used on short sequence lengths. It is unknown at what sequence length dependencies can be identified in the input data. By creating multiple models that apply the RNN variants at different sequence lengths, models may be generated that identify short, medium, and long term dependencies. Ensembling these models into a new neural network could then provide a more effective model for predicting stock market prices. A Single-Layer Perceptron (SLP) will be used to ensemble the trained RNN models into a single model. The SLP model has been chosen as it can create linear combination of the RNN for its predictions..

Method Implementation

Full Python code implantation: https://github.com/Jmarkey11/DLF-Assignmet-3/blob/main/a3temp.ipynb

The training methodology for RNN, GRU, and LSTM models using PyTorch involves a comprehensive approach, starting with data importation and preprocessing. Data from the Yahoo Finance library, spanning from 2000-01-01 to 2023-01-01. The data gathered contains the share information for Apple. The initial dataset comprises six features: open, high, low, close, adjusted close, and volume. Through feature engineering, this is expanded to 14 features by adding 30 and 90-day moving averages, standard deviation, and Bollinger Bands, thereby enhancing the model's ability to capture intricate stock behaviours and patterns. The target variable for the model is the 'close' value.

The preprocessing stage involves min-max scaling of the data to the range of -1 to 1. This normalisation ensures that all features contribute equally to the model's learning process, stabilising and speeding up the optimisation. The data is then segmented into different time sequences (5, 10, 20, 40, and 80 days)

and split into distinct training, testing, and validation datasets. The split is date-based to maintain the chronological integrity of the time-series data. The validation dataset includes data from January 1, 2018, to December 31, 2021; the testing dataset covers January 1, 2022, to January 1, 2023; and the training dataset comprises data prior to January 1, 2018. Batching is employed during preprocessing to make the training process more manageable and efficient. It breaks the dataset into smaller batches, allowing the model to update parameters incrementally and leading to faster convergence and fewer training epochs. Batching is implemented using the DataLoader() and TensorDataset() functions from the torch.utils.data library. The training data is not shuffled during batching to preserve the time-series data's sequential nature.

The implementation of RNN, GRU, and LSTM models in PyTorch is structured to begin with class initialisation. This setup includes specifying parameters such as input dimension, hidden size, output dimension, the number of hidden layers, and the dropout rate. The hidden size determines the size of the hidden state vector, and the number of hidden layers indicates the depth of the model with stacked RNN, GRU, or LSTM cells. The dropout rate is introduced to prevent over-reliance on specific neurons and mitigate overfitting by randomly disabling a proportion of neurons during training. The RNN(), GRU(), and LSTM() functions from the torch.nn library are utilised to construct the respective hidden layer variations. The output layer is created using the Linear() function from the same library, responsible for producing the final output of the model. Each class has a forward method that dictates the operation flow. It starts by initialising the hidden state (and cell state for LSTM) as zero tensors. Then, the data is passed through the chosen RNN, GRU, or LSTM layer, followed by the output layer to generate the final predictions.

The training process, consistent across all three model types, begins with the initialisation of the loss function and optimiser.

$$L_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y_i})^2$$

Equation 7: MSE Loss Function [7]

The loss function used is the Mean-Squared Error (MSE) as expressed in Equation 7. The MSE is equal to the mean of the squared difference between the predicted value $\tilde{y_i}$ and the actual value y_i. The optimiser used in the training loop is the Stochastic Gradient Descent (SGD) optimiser, implemented using the SGD() function from the torch.optim library. Unlike conventional gradient descent, which uses the entire dataset for gradient calculation, SGD utilises mini-batches, providing diverse data samples per iteration and avoiding local minima. SGD is tasked with computing the gradients and updating the learnable parameters through the BPTT process, and is regulated through a set learning rate. In addition to the learning rate, SGD also utilised momentum and weight-decay. Additional features of SGD include momentum, which integrates previous updates to ensure consistency in gradient direction across mini-batches, thereby accelerating convergence, and weight decay, a regularisation technique that shrinks the learnable parameters slightly to prevent overfitting.

A learning rate scheduler, implemented using the ReduceLrOnPlateau() function from the torch.optim.lr_scheduler library, is incorporated to enhance the training process. This scheduler reduces the learning rate by a factor of 0.5 if there's no improvement in validation loss after 10 epochs, eliminating the need for manual adjustment of learning rates. Early stopping is also implemented, halting training when the learning rate falls below 3.1250e-04 or after 300 epochs. Validation is an integral part of the modelling process, running alongside training. It employs a separate dataset to evaluate the model's performance, check for overfitting, and guide iteration refinements. During each training epoch, additional evaluation metrics are calculated, which assist in making informed decisions about when to cease training and helping to prevent overfitting.

After the RNN models are trained and validated, they are utilised in an ensemble method using a SLP. The SLP is implemented in PyTorch and begins with the class initialisation, which involved defining the parameters for a linear layer from the torch.nn library. The input for the SLP comprises the predictions made by the RNN models. Given that there are three different RNN model variants, each applied to five distinct sequence lengths, the SLP receives a total of 15 input features (3 RNN variants × 5 sequence lengths). The SLP's architecture is structured with a single where the 15 input features are condensed into a single output value. The SLP achieves this through training. The training procedure is similar to the RNN model training procedure, where the loss function is MSE and the optimiser is SGD. Validation is also used as it provides a reference for comparison to the individual RNN models.

The method of evaluation for these models involves analysing and contrasting the 15 models with the ensemble model. This will involve analysing the training, validation, and testing MSE loss values in addition to additional evaluation metrics of Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE). MAE is similar to MSE, except that is equal to the absolute difference between the predicted and true values compared to the squared difference. RMSE

is also tied to MSE and is equal to the root of MSE. MSE, MAE, and RMSE are appropriate evaluation metrics for this report as stock market price prediction is a regression task and these metrics are apt in this scenario.

Experiments and Analysis

Analysing the performance of the RNN variants (Appendix, Figures 7-8) reveals unexpected results. Contrary to typical expectations, RNN models outperform the GRU and LSTM variants across most configurations. This is unusual because RNNs are generally considered less capable of capturing long-term dependencies compared to GRU and LSTM. The general expectation for these results was that the RNN models would have had the best performance for the short-term sequence lengths, while the GRU would have had the best performance for medium-term lengths, and that the LSTM would have excelled at the long-term lengths. The RNN model with a sequence length of 40 shows the best performance out of all models with the lowest validation loss of 0.00096, indicating it has learned to predict the sequences with the least error.

Another surprising aspect of the analysis is the performance of LSTMs. LSTMs are theoretically superior in learning and predicting long-term dependencies within sequences. Contrary to these expectations, LSTMs displayed the highest loss values across all sequence length configurations, lagging behind the performance of the other models by a factor of 10. This suggests difficulties in pattern recognition within the dataset. This is evident when analysing the graph in Figure 8 which shows that the LSTM models begin to deviate from the true value when the true value shows an upwards trend. This underperformance hints at the possibility that LSTMs might benefit from additional training epochs to fully capture and learn the sequence dependencies.

The ensemble SLP model's results are seen in Figure 9.The results of the ensemble method indicate that the method is performing very well with a validation loss of 0.00235, MAE of 0.03809 and RMSE of 0.04850. These metrics would indicate that the ensemble method is the 7th best performing model. After training the model the weights were identified as 0.1777, -0.0300, -0.0457, 0.3146, -0.0745, 0.0987, 0.3601, -0.0111, -0.0424, 0.0514, 0.0868, -0.1269, 0.3723, -0.0985, 0.0637 for the RNN 5, RNN 10, RNN 20, RNN 40, RNN 80, GRU 5, GRU 10, GRU 20, GRU 40, GRU 80, LSTM 5, LSTM 10, LSTM 20, LSTM 40, and LSTM 80 models respectively. The models that contribute the most to the ensemble are the RNN 40, GRU 20, and LSTM 20 models. Given the RNN and GRU excellent performance, it is unsurprising that these models have high weights, but the LSTM's model's weight is surprising given its poor performance. This suggests that the RNN models and GRU models may be overpredicting the stock market prices, leading to predictions that are too high, requiring the LSTM model to bring their performance inline.

Overall the results of the experiments are quite surprising with the RNN model showing better than expected performance and the LSTM showing worse than expected performance. The ensemble method shows encouraging results with very good performance. However, its performance is not high enough to support its use over the individual RNN and GRU models.

Conclusion

The aim of this report was to present an in-depth exploration of the various Recurrent Neural Network (RNN) variants and to determine whether a stacked generalisation ensemble approach could be used to harness the collective strengths of the individual variants and overcome their weaknesses. RNN models are a type of neural network architectures that specialise in using sequential data to make predictions. RNNs specialise in predicting sequential data due to their recurrent nature, where outputs are passed back into the model for use in future predictions, establishing a memory. While basic RNN architectures are typically efficient with short-term sequences, it suffers from short-term memory issues, usually resulting in poor performance when long sequences are analysed. This shortcoming is what the Gated Recurrent Unit (GRU) and Long Short Term Memory (LSTM) variants are designed to fill, albeit with their own sets of challenges.

RNN, GRU and LSTM models were generated and used tested on data made up of sequences of various lengths. Surprisingly, the performance of the standard RNNs surpassed the GRU and LSTM models, achieving the lowest validation loss in the 40-sequence length dataset. This performance contradicts the usual expectation of the LSTMs outperforming the other variants in long-term dependency tasks. Instead, the LSTM exhibited a significant gap in performance, with an apparent struggle when predicting upward trends.

The ensemble model, constructed using a Single-Layer Perceptron (SLP), showcased robust predictive capabilities, ranking seventh in performance. Surprisingly, the LSTM model received a substantial weight within the ensemble despite its poor standalone performance, hinting that it might play a corrective role to the RNN's and GRU's' tendency to overpredict. The ensemble method, while effective, did not outperform the individual RNN and GRU models to a degree that would justify its preference. The report underlines the need for empirical testing in developing effective predictive models for the stock market.

There are three main avenues for potential future work in this area. Additional feature engineering could significantly enhance the performance of the RNN models used for stock price prediction. By incorporating more diverse and potentially influential factors, the models could capture a broader range of dependencies and patterns that affect stock prices. For instance, including non-financial data like market sentiment, derived from news articles, social media trends, or economic indicators, could provide insights. The main assumption of the RNN models is that the stock market data alone can be used to predict prices. The non-financial data could reveal complex relationships and trends that are not immediately apparent from financial data alone, allowing the RNN models to potentially establish more comprehensive and nuanced dependencies, leading to more accurate predictions.

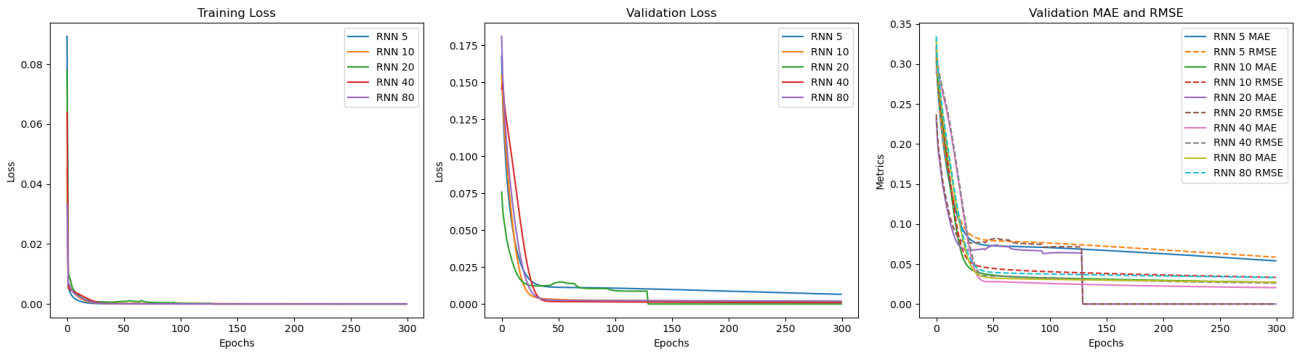
The second avenue for future work would be to implement hyperparameter tuning to the models. For the implemented method, only a limited number of hyperparameters were tested. Further tuning to explore the effect of different hidden sizes, number of hidden layers, number of companies tested, learning rates, training epochs and data sequences could reveal more dependencies in the data and potentially overcome the issues identified in the model, such as the underperformance of the LSTM.

The last avenue of future work would be to implement the competitor CNN and Transformer models and compare them with the implemented models. Testing the alternative models and potentially integrating them with the ensemble method could be beneficial as the CNN and Transformer architecture are both quite distinct from RNN and could potentially identify dependencies that the data that are non-sequential.

References

1. Sethia, A. and Raut, P. (2018) 'Application of LSTM, GRU and ICA for Stock Price Prediction', Smart Innovation, Systems and Technologies, 107. doi:https://doi.org/10.1007/978-981-13-1747-7_46.
2. Brownlee, J. (2020) Stacking Ensemble for Deep Learning Neural Networks in python, MachineLearningMastery.com. Available at: https://machinelearningmastery.com/stacking-ensemble-for-deep-learning-neural-networks/ (Accessed: 23 November 2023).
3. Gehring, J. *et al.* (2017) *Convolutional Sequence to Sequence Learning* [Preprint]. doi:https://doi.org/10.48550/arXiv.1705.03122.
4. Vaswani, A. et al. (2017) 'Attention Is All You Need', Advances in neural information processing systems, pp. 5998–6008. doi:https://arxiv.org/abs/1706.03762.
5. dprogrammer (2019) RNN, LSTM & Gru, Recurrent Neural Network (RNN), Long-Short Term Memory (LSTM) & Gated Recurrent Unit (GRU). Available at: http://dprogrammer.org/rnn-lstm-gru (Accessed: 23 November 2023).
6. Zarga, S.A. (2021) *Introduction to Sequence Learning Models: RNN, LSTM, GRU*. Available at: https://www.researchgate.net/profile/Sakib-Zargar-2/publication/350950396_Introduction_to_Sequence_Learning_Models_RNN_LSTM_GRU/links/607b41c0907dcf667ba83ade/Introduction-to-Sequence-Learning-Models-RNN-LSTM-GRU.pdf.
7. Pytorch (2023) Torch.nn Recurrent Layers, torch.nn - PyTorch 2.1 documentation. Available at: https://pytorch.org/docs/stable/nn.html#recurrent-layers (Accessed: 23 November 2023).
8. Antoniadis, P. (2023) *Activation functions: Sigmoid vs Tanh*, *Baeldung on Computer Science*. Available at: https://www.baeldung.com/cs/sigmoid-vs-tanh-functions (Accessed: 23 November 2023).
9. M, H. et al. (2018) 'NSE stock market prediction using deep-learning models', Procedia Computer Science, 132, pp. 1351–1362. doi:10.1016/j.procs.2018.05.050.
10. Zhang, A. et al. (2023) Dive into Deep Learning 1.0.3 9.7. backpropagation through time, Dive into Deep Learning 1.0.3. Available at: https://d2l.ai/chapter_recurrent-neural-networks/bptt.html (Accessed: 23 November 2023).
11. Nosouhian, S., Nosouhian, F. and Kazemi Khoshouei, A. (2021) *A review of recurrent neural network architecture for Sequence learning: Comparison between LSTM and gru* [Preprint]. doi:10.20944/preprints202107.0252.v1.

Appendix



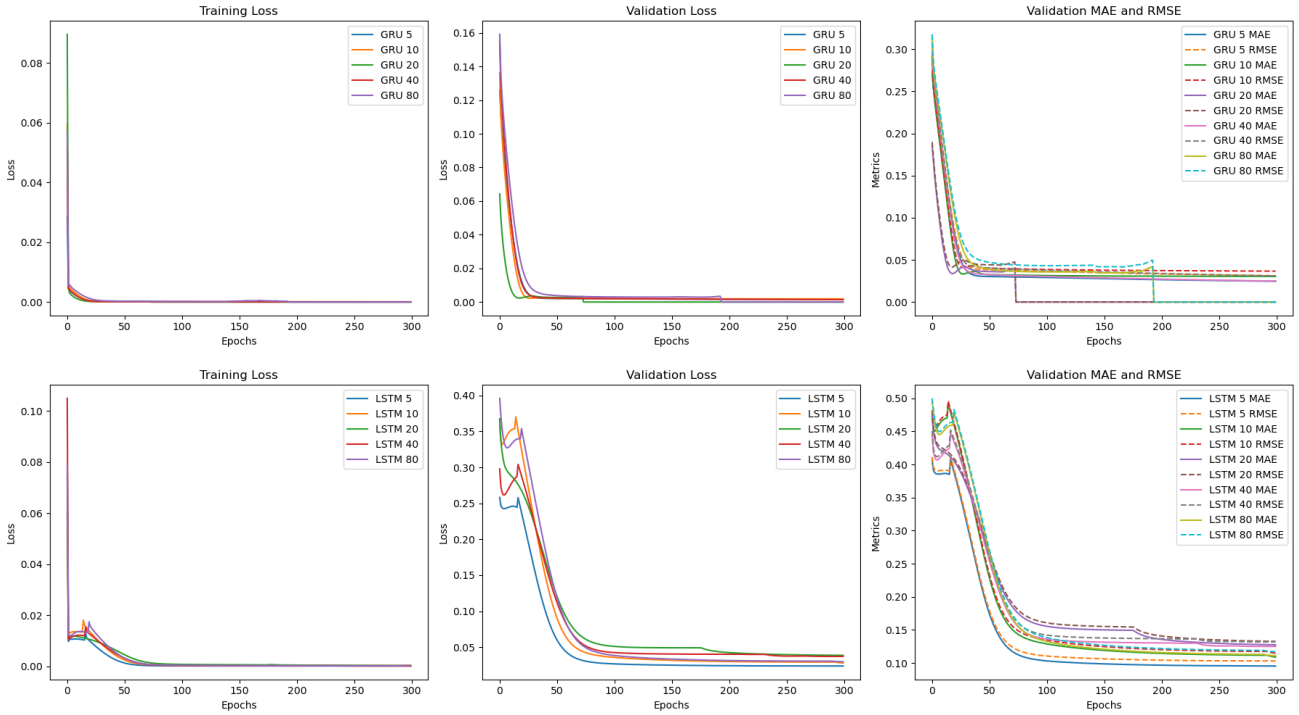


Figure 7: RNN, GRU and LSTM Training Results

```
Model: RNN 5, Final Model Epoch: 299, Train Loss: 0.00005, Val Loss: 0.00655, Val MAE: 0.05393, Val RMSE: 0.05852
Model: RNN 10, Final Model Epoch: 299, Train Loss: 0.00009, Val Loss: 0.00164, Val MAE: 0.02707, Val RMSE: 0.03336
Model: RNN 20, Final Model Epoch: 118, Train Loss: 0.00020, Val Loss: 0.00869, Val MAE: 0.06400, Val RMSE: 0.07147
Model: RNN 40, Final Model Epoch: 299, Train Loss: 0.00003, Val Loss: 0.00096, Val MAE: 0.02048, Val RMSE: 0.02622
Model: RNN 80, Final Model Epoch: 299, Train Loss: 0.00006, Val Loss: 0.00194, Val MAE: 0.02719, Val RMSE: 0.03311

Model: GRU 5, Final Model Epoch: 299, Train Loss: 0.00005, Val Loss: 0.00141, Val MAE: 0.02468, Val RMSE: 0.03116
Model: GRU 10, Final Model Epoch: 299, Train Loss: 0.00008, Val Loss: 0.00189, Val MAE: 0.03037, Val RMSE: 0.03662
Model: GRU 20, Final Model Epoch: 17, Train Loss: 0.00028, Val Loss: 0.00234, Val MAE: 0.03388, Val RMSE: 0.04135
Model: GRU 40, Final Model Epoch: 299, Train Loss: 0.00004, Val Loss: 0.00139, Val MAE: 0.02486, Val RMSE: 0.03110
Model: GRU 80, Final Model Epoch: 170, Train Loss: 0.00046, Val Loss: 0.00287, Val MAE: 0.03487, Val RMSE: 0.04219

Model: LSTM 5, Final Model Epoch: 298, Train Loss: 0.00011, Val Loss: 0.02380, Val MAE: 0.09544, Val RMSE: 0.10300
Model: LSTM 10, Final Model Epoch: 299, Train Loss: 0.00032, Val Loss: 0.02801, Val MAE: 0.10885, Val RMSE: 0.11479
Model: LSTM 20, Final Model Epoch: 299, Train Loss: 0.00024, Val Loss: 0.03868, Val MAE: 0.12754, Val RMSE: 0.13269
Model: LSTM 40, Final Model Epoch: 292, Train Loss: 0.00008, Val Loss: 0.03711, Val MAE: 0.12501, Val RMSE: 0.13185
Model: LSTM 80, Final Model Epoch: 299, Train Loss: 0.00022, Val Loss: 0.02973, Val MAE: 0.11092, Val RMSE: 0.11705
```

Figure 8: Model Training Performance

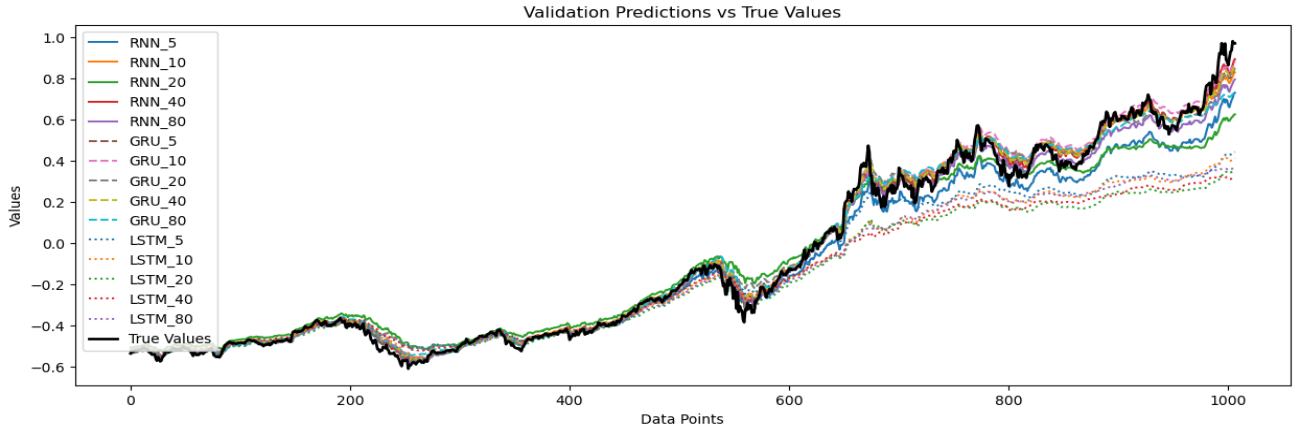


Figure 9: Models Training Performance Graph

```
Best Model Stats: Best Epoch: 999, Train Loss: 0.00005, Val Loss: 0.00235, Val MAE: 0.03809, Val RMSE: 0.04850
```

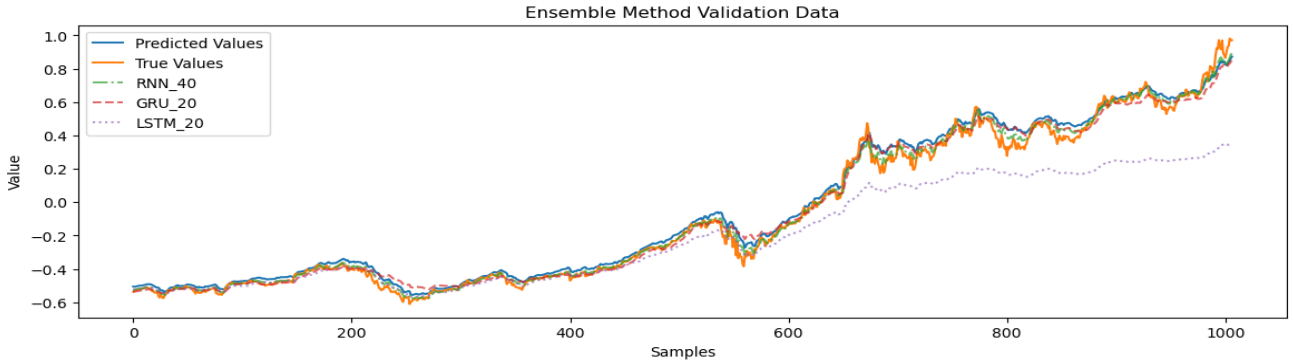


Figure 10: Ensemble Model Training Performance and Best Model Statistics