A basic exploration into Recurrent Neural Networks and their applications on time-series forecasting through comparing a Vanilla Recurrent Neural Network against advanced Long Short Term Memory and Gated Recurrent Unit Neural Networks

Anonymous CVPR submission

Paper ID *****

Abstract

Recurrent Neural Networks (RNN) in recent years have gained traction for their suitability in modeling sequential data (Sun et al. 2023, p.1), which makes them effective for tasks involving time-series prediction. This paper aims to delve into the fundamental architectures of RNNs and the advanced variants such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU). The intent of this paper is to understand what goes on under the hood of these neural networks while highlighting the differences in architecture. Using stock price prediction as a case study, the performance of each RNN will be compared to demonstrate how each model captures the temporal dependencies in time-series data. While the focus of this paper is understanding each architecture and the respective behaviors, their performances will also be evaluated in terms of the predictive accuracy, particularly in terms of the performance gains after hyperparameter tuning. The results will provide insights into the strengths and limitations of each architecture in regards to their suitability for time-series prediction tasks. This paper aims to serve as an educational start point for leveraging RNNs and the variants in practical applications.

1. Introduction

1.1. Background

Time-series data can be found in all sectors, for example in stock trading firms, weather forecast stations and even agriculture. It is to say, any industry that has data on time-series can benefit from harnessing the power of time-series predictions.

However, time-series data is inherently sequential, and the value of a given time step depends on the values from previous time steps (Sun et al. 2023, p.1). For tasks involving predicting future values, it is important that these temporal dependencies - the patterns that occur across time, can be understood by the model chosen for the task.

Traditional feed-forward machine learning models such as Perceptrons or Convolutional Neural Networks, while effective in other applications, fall short in this task as they are not able to capture these temporal relationships effectively, due to the lack of mechanisms to retain information from previous inputs (Bengio, Simard & Frasconi 1994, p.1).

On the other hand, Recurrent Neural Networks with hidden states and encoder/decoder architectures can overcome this limitation and act as powerful tools to bridge the gap for these tasks and have been widely adopted as the tool for other similar tasks, such as speech recognition (Graves et al. 2013), natural language processing and notably time-series prediction (Chung et al. 2014, p.1; Sun et al. 2023, p.1).

1.2. Objective

The primary aim of this paper will be to provide a thorough explanation and understanding of the architectures of Recurrent Neural Networks - Vanilla, LSTM and GRU - by highlighting the intricacies of the model and evaluating the performances of these model on a test case time-series data.

While the predictive performances of the model is evaluated, the main objective of this paper is to help understanding of the model architecture and their possible practical applications, particularly on how well RNNs can perform on time-series data.

As a note: the provided assigned training and test data is limited and differ in complexity to real life stock data which can be affected by many additional factors. The experiment conducted in this paper should act as a guideline or stepping stone to be improved on to be applied to more complex use cases.

2. Recurrent Neural Networks

2.1. Design Philosophy

Recurrent Neural Networks (RNNs) have grown in popularity due to their ability to maintain contextual information across time via hidden state vectors which act as memory mechanisms (İlhan et al. 2020, p.1). To put simply, RNNs are designed similar to a feed-forward neural network that is "unrolled" over time to process inputs sequentially (Géron 2019, pp. 537-542), hence the "recurrent" namesake.

The architecture of a Recurrent Neural Network comprises 4 main components (Cho et al. 2014, p.3, Das et al. 2023, pp.117-122; Géron 2019, pp. 537-542):

1. Input layer: at each time step t, the RNN takes in an input vector x_t (e.g. the stock prices for the day) of size $input_size$.

2. Hidden Layer: Each time step's input x_t is processed here and updated iteratively. It combines the current input h_t with the previous hidden state h_{t-1}. These hidden state vectors are the defining traits of a RNN. The hidden state vectors are updated at every time step as the network processes sequential inputs with the formula:

$$h_t = f(\mathrm{W}_h \cdot h_{t-1} + \mathrm{W}_x \cdot x_t + b)$$

where:

* h_t: The hidden state h_t is the RNN's memory, which is updated iteratively across each time step.

* h_{t-1}: The hidden state from the previous time step, representing the information accumulated.

* x_t: The input vector at time step t.

* W_h, W_x: Weight matrix for the hidden state, and weight matrix for the input, respectively.

* b: the bias term.

* f: A non-linear activation function, usually $tanh$ or $ReLU$.

The number of neurons in the hidden layer is called the hidden size. A larger hidden size allows more complex patterns to be modeled.

Multiple RNN layers can be stacked to learn deeper hierarchical representations. The output of one layer ($h_t{}^{(l)}$ serves as the input to the next layer:

$$h_t{}^{(l)} = f(W_h{}^{(l+1)} \cdot h_t{}^{(l)} + W_x{}^{(l+1)} \cdot x_t{}^{(l+1)} + b)$$

3. Output Layer: The output layer can produce outputs at either every time step y_1, y_2,...,y_r or only at the last time step. The output layer is a fully connected layer which maps the hidden state of the final time step to the desired output and the activation function can be a linear function for regression tasks or a softmax for classification tasks. In our case of predicting a stock price for the future, a linear activation function is used.

By updating the hidden state iteratively, the RNN 'learns' the temporal dependencies in the sequence by leveraging the encoder-decoder architectures (Cho et al. 2014, p.1) within itself to process input sequences and generate outputs (Das et al. 2023, pp.117-122) . The encoder processes input sequence step by step and updates the hidden state at each time step. The final hidden state after the sequence is processed entirely is then passed onto the decoder, which then generates the output sequence one time step at a time (Cho et al. 2014, p.3).

This architecture within a RNN allows information/temporal relationships from input sequence to be kept within the hidden state and allows the model to predict outputs with better context from history.

In the case of predicting stock prices where today's stock price could be affected by observations from previous time steps, having the ability to 'understand context' or 'remembering history' makes RNN a very effective architecture for the task.

2.1.1 Strengths and Challenges of Vanilla RNNs

The advantage of an RNN is the ability to retain temporal dependencies in sequential data through the usage of hidden states. This allows the network to share weights across time steps and effectively lowers the number of trainable parameters compared to traditional feed-forward neural networks.

However, the challenge that a RNN can face as the scale or size of the network increases is the vanishing and exploding gradient problem (Bengio, Simard & Frasconi 1994, p.1).

In back-propagation, the gradients of the loss in weight is computed as products of the weight matrix W_h at each time step, creating a chain of gradients:

$$\frac{\delta_L}{\delta_{W_h}} = \sum_{t=1}^{T} W_h \cdot \frac{\delta L}{\delta h_t}$$

As gradients are back-propagated through large time steps, the gradient diminishes excessively and makes it near impossible to learn long-term dependencies as basically no updates to the weights is done. On the other hand, exploding gradients grow exponentially large during back-propagation and can destabilize training as well by causing the weight to oscillate or diverge.

In order to overcome the exploding gradient problem, gradient clipping was introduced. The theory behind gradient clipping is to rescale the gradients to fit within a threshold if the gradient norm exceeds a defined threshold during back-propagation, effectively preventing excessively large updates that can cause the explosion (Zhang et al. 2019). This technique stabilizes the training with no change to the underlying RNN structure.

There are also other techniques such as proper weight initialization, layer normalization and shortening the sequences to mitigate these gradient problems. However, the more well-known methods are two variants of RNNs called Long Short Term Memory and Gated Recurrent Units.

2.2. Long Short-Term Memory and Gated Recurrent Units

Both Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are variants of Vanilla RNNs. These variants introduce gating mechanisms to implement control over how the information is propagated throughout the network. This helps to mitigate and prevent gradient problems previously mentioned from happening.

3. Long Short-Term Memory

LSTMs build on the Vanilla RNN architecture by now introducing cell states and gates which helps to control information flow (Das et al. 2023, pp.122-124, Hochreiter & Schmidhuber 1997). By introducing these gates, the network is able to retain important information while forgetting irrelevant information and decide the output at each state.

3.1. Architecture of Long Short-Term Memory

The architecture of LSTM builds on the existing RNN but introduces new components:

* **Cell State** (C_t): A separate memory component that flows through the network but remains unchanged. This cell carries information within it around the network and allows gradients to travel along the network without shrinking or exploding.

* **Gates**: Gates are additional neural network layers introduced to control the flow of information. It has a sigmoid (δ) activation function to output values between 0 (forget) and 1 (retain).

There are 3 gates:

1. **Input gate** (i_t) and **Candidate Values** $(\check{C}_t)$: The input gate decides what new information to store in the cell state

$$i_t = \delta(W_i \cdot [h_{t-1}, x_t] + b_i)$$

and the candidate values suggests candidate values to be added to the cell state:

$$\check{C}_t = tanh(W_C \cdot h_{t-1}, x_t] + b_C)$$

2. **Forget Gate** (f_t): Forget gates decides which information from previous cell state $C_t - 1$ should be forgotten:

$$f_t = \delta(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where:

h_{t-1} is the hidden state from the previous time step.

x_t is the current input

W_f, b_f are the forget gate's weights and biases.

The Cell State (C_t) is updated by combining the forget gate and input gate:

$$C_t = f_t \cdot C_{t-1} + i_t = \check{C}_t$$

3. **Output gate** (o_t) **and Hidden State** (h_t): The output gate finally determines which part of the cell state will be output as the hidden state:

$$o_t = \delta(W_o \cdot [h_{t-1}, x_t] + b_o)$$

and the output from the timestep is determined by the hidden state:

$$h_t = o_t \cdot tanh(C_t)$$

3.2. Advantages of Long Short-Term Memory

By using the cell state (C_t) to act as a "long-term" memory, the information can flow along the cell state with minimal linear modifications (forget and add), which mitigates the vanishing gradient problem previously mentioned. The forget gate is useful in allowing irrelevant information to be forgotten and preserves important long-term dependencies.

The gates in LSTMs control how much of the new information is added and also how much of past information is remembered. With the additional components, the network is able to focus on important temporal relationships while forgetting irrelevant noise.

4. Gated Recurrent Unit (GRU)

Gated Recurrent Units (GRUs) are a simpler version of a LSTM model. It is able to achieve similar benefits but with fewer trainable parameters. The difference between a GRU and a LSTM neural network lies largely in how a GRU combines both the cell state and the hidden state into one single hidden state, instead of separating the two (Das et al. 2023, pp.124-125). A GRU also has fewer gates (Chung et al. 2014; Sun et al. 2023): just an update gate and a reset gate, which helps it to be more computationally efficient compared to a LSTM.

4.1. Architecture of Gated Recurrent Unit

The GRU has 2 gates:

1. **Update Gate** (z_t): The update gate determines how much of the previous hidden state $h_t - 1$ to retain and how much to replace with newer information:

$$z_t = \delta(W_z \cdot [h_{t-1}, x_t] + b_z)$$

2. **Reset Gate** (r_t): The reset gate decides how much of the previous hidden state $h_t - 1$ to forget:

$$r_t = \delta(W_r \cdot [h_{t-1}, x_t] + b_r)$$

3. **Candidate Hidden State** $(\tilde{h}_t)$: A new hidden state which uses the reset gate r_t to control how much the past information from previous time step influences the current update:

$$\tilde{h}_t = tanh(W \cdot [r_t \cdot h_{t-1}, x_t] + b)$$

4. **Final Hidden State** (h_t): The final hidden state then combines the previous hidden state $h_t - 1$ and the candidate hidden state $\tilde{h}_t$ using the update gate:

$$h_t = z_t \cdot h_{t-1} + (1 - z - t) \cdot \tilde{h}_t$$

4.2. Advantages of Gated Recurrent Unit

The GRU is able to mitigate vanishing gradients by using its reset gate (r_t) to ensure that only relevant past information is remembered, stopping gradients from diminishing from unnecessary updates. The update gate (z_t) preserves long term dependencies by controlling how much of the past will be carried forward. A GRU functions much like a LSTM neural network, but a GRU can be much simpler and faster to implement.

4.3. Comparison of Vanilla, LSTM and GRU

The Vanilla RNN is able to retain temporal differences in time series data, which is an added advantage to traditional feed-forward neural networks. By using hidden state vectors to remember temporal dependencies of data fed into it. However, a Vanilla RNN suffers from the common gradient problems that arise as the network grows larger and the network has to deal with excessively small or large gradient updates.

LSTMs and GRUs were created in response to these problems. Both LSTMs and GRUs are excellent at handling long-term dependencies in sequential data. The model selection will depend on the use case. An LSTM is particularly useful when the time-series spans very long term and the model needs to be robust enough to capture these very long term dependencies. A GRU on the other hand is preferred when the training needs to be fast and there are less parameters required to be trained, and the sequence are shorter in length.

5. Experimental Setup and Methodology

The experiment will include setting up 3 different models: Vanilla, LSTM and GRU on a stock price dataset, which was pre-assigned and publicly available at https://www.kaggle.com/rahulsah06/gooogle-stock-price.

The dataset has 5 columns: **'Open'**: The opening price of the stock for the day.

'Close': The closing price of the stock for the day.

'High': The highest price recorded of the stock for the day.

'Low': The lowest price recorded of the stock for the day.

'Volume': The total volume of the stock traded for the day.

The target variable to predict will be the 'Close' price for the next day.

Data preparation was done by:

1. **Normalization**: rescaling all the numerical features of the dataset with Min-Max Scaling to fit the neural networks better.

2. **Sliding Windows**: Input-output pairs were also generated with a sliding window approach of 60 time steps, hence the input would be 60 days within the time step and the target would be the 'Close' price on the 61st day.

3. **Data Splitting**: The dataset was split chronologically into a 60-20-20 ratio with the first 60% for training, the next 20% for validation and the final 20% used for testing. The split was performed chronologically to ensure no data leakage and maintain integrity between dates.

Each models were initiated with consistent hyperparameters where possible with a default setting of: * **Hidden Size**: 50 * **Number of Layer**: 1 layer, * **Learning rate**: 0.001 * **Batch size**: 32 * Number of **Epochs**: 50

Hyperparameter tuning was then performed via a grid search method on all models before the final tuned models were compared against each other.

The evaluation metrics used included: **Mean Squared Error (MSE)**: To quantify the average squared differences between the predicted and actual values.

Mean Absolute Error (MAE): To measure the average magnitude of prediction errors, easier to interpret in original scale.

R-Squared (r^2) **values**: The proportion of variance in the target variable the model can explain

The experiment was conducted on an Apple MacBook Air with an M2 chip.

6. Experiment

6.1. Vanilla RNN

The Vanilla RNN will act as a baseline model for this paper, due to the simplicity and ease of implementation. The Vanilla RNN is a foundational model for modeling sequential data and acts as a good starting point. Given the simplicity of the model, Vanilla RNNs are limited in their ability to model long-term temporal dependencies given a large time frame.

The baseline Vanilla RNN was initiated with the following hyperparameters over 50 epochs:

and Figure 1 plots the performance during the testing. Table 2 shows the performance metrics.

Table 1. Default Hyperparameters

Hyperparameter	Value
Hidden Sizes	50
Number of Layers	1
Learning Rates	0.001
Batch Sizes	32

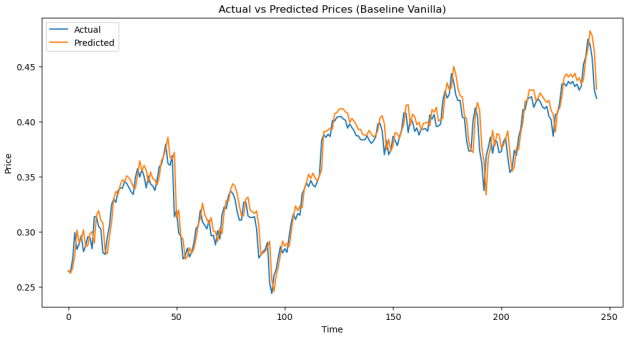

Figure 1. Vanilla RNN

Table 2. Performance Metrics for the Vanilla RNN

Metric	Value
Mean Squared Error (MSE)	0.000378
Mean Absolute Error (MAE)	0.016534
R^2	0.853757

6.2. Hyperparameter-tuned Vanilla RNN

A GridSearch method was implemented to loop through different hyperparameters:

Table 3. Hyperparameter Grid for Vanilla RNN Tuning

Hyperparameter	Search Parameter
Hidden Sizes	50, 100, 200
Number of Layers	1, 2
Learning Rates	0.01, 0.001, 0.0001
Batch Sizes	16, 32, 64

The tuning was done over 10 epochs for faster tuning and the final chosen hyperparameters were:

Table 4. Optimal Parameters for Vanilla RNN

Hyperparameter	Value
Hidden Sizes	50
Number of Layers	2
Learning Rates	0.001
Batch Sizes	16

The model was finally retrained with the tuned hyperparameter settings over 50 epochs. Figure 2 shows the plotted performance.

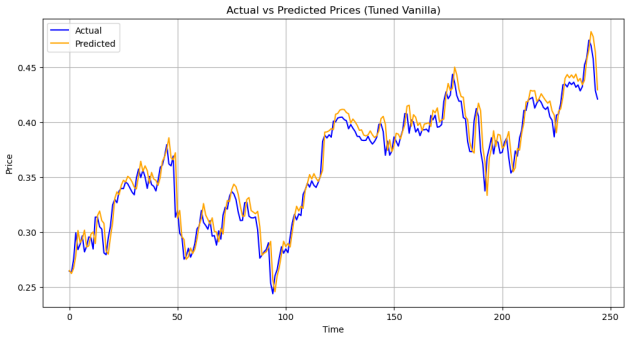

Figure 2. Tuned Vanilla RNN prediction

Final performance of Vanilla RNN (tuned):

Table 5. Performance Metrics for Tuned Vanilla RNN

Metric	Value
Mean Squared Error (MSE)	0.000301
Mean Absolute Error (MAE)	0.013826
R^2	0.883645

We can observe that the Vanilla RNN is performing decently well as per the metrics, with very small errors and high r-square value. However, the plot is showing a trend where the model is over and under predicting the price when there are bigger spikes in price change. It is likely that a Vanilla RNN is struggling to capture these relationships. The variants (LSTM and GRU) should theoretically perform better in this aspect.

6.2.1 Gradient Clipping in Vanilla RNN

Gradient clipping was tested during training and tuning to stabilize weight updates, however it was found unnecessary as the training and testing was stable without it. Removing the clipping actually improved the convergence and predictive performance. This was likely due to the simplicity of the dataset and the gradients updates were already stable, hence not including gradient clipping allowed appropriate weight updates.

7. Long Short-Term Memory RNN

The baseline Long Short-Term Memory RNN was designed to address the limitations that our tuned Vanilla RNN had. A LSTM is able to capture long-term dependencies more effectively but also comes at a cost of model complexity and computational power. This is due to the forget, input and output gates implemented which helps the model

retain important information over longer time sequences while forgetting irrelevant data.

Our baseline LSTM was initiated with similar hyperparameters to the baseline Vanilla RNN:

Table 6. Default Parameters for LSTM RNN

Hyperparameter	Value
Hidden Size	50
Number of Layers	1
Dropout	0.2
Learning Rate	0.001
Batch Size	32

The only notable difference being the introduction of a dropout layer. The dropout layer was omitted from the Vanilla RNN due to the inherent vanishing gradient problem and difficulty in remembering long term dependencies, and adding dropout would degrade Vanilla RNN performance further. However, adding a dropout layer in LSTM is beneficial due to the gating mechanism present which mitigates the problems a Vanilla RNN faces.

After initial training of the baseline LSTM, this was the performance:

Table 7. Performance Metrics for Baseline LSTM

Metric	Value
Mean Squared Error (MSE)	0.000400
Mean Absolute Error (MAE)	0.013800
R-Squared (R^2)	0.883600

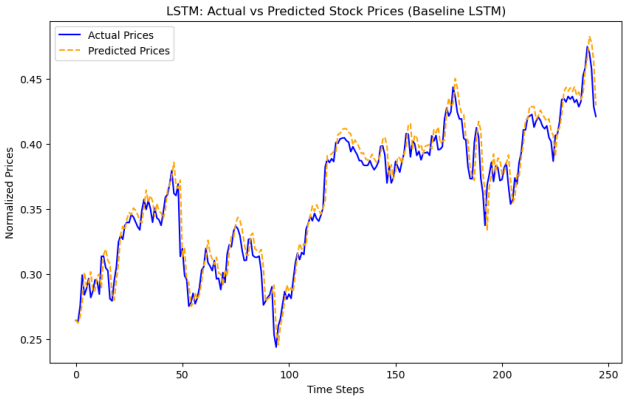

Figure 3. Baseline LSTM prediction

Overall performance in metrics were slightly better than the tuned Vanilla models. However, it can be observed from the graph that the model was consistently under-predicting prices. Further tuning will be focused on allowing the LSTM to capture the price range and relationships.

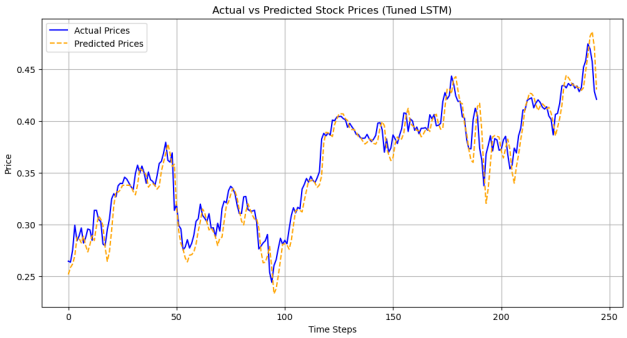

Figure 4. Tuned LSTM predictions

The grid search tuning was performed and the optimal parameters were:

Table 8. Best Hyperparameters for Tuned LSTM

Hyperparameter	Value
Hidden Size	200
Number of Layers	1
Dropout Rate	0.5
Learning Rate	0.001
Batch Size	16

The tuned LSTM model now has performance of:

Table 9. Performance Metrics for Tuned LSTM

Metric	Value
Mean Squared Error (MSE)	0.000208
Mean Absolute Error (MAE)	0.010641
R-Squared (R^2)	0.919493

which is excellent. The graph shows good predictions that are nearly identical with the actual prices and Table X shows very small errors and high r-square values.

The tuned LSTM model outperforms the tuned Vanilla RNN model by a small margin. It has learned the temporal dependencies effectively and is able to predict prices based on remembering history to predict future values.

8. Gated Recurrent Units

Gated Recurrent Units being a simpler version of a LSTM model are expected to work almost as well as an LSTM but with lower computational expense. We will try to build a baseline GRU model and then tune it to see if comparable performance can be observed.

The baseline GRU will be initiated similar to our LSTM:

The GRU reports a performance of:

Table 10. Default Hyperparameters for GRU Model

Hyperparameter	Value
Hidden Size	50
Number of Layers	1
Dropout Rate	0.2
Learning Rate	0.001
Batch Size	32

Table 11. Performance Metrics for Baseline GRU

Metric	Value
Mean Squared Error (MSE)	0.0003
Mean Absolute Error (MAE)	0.0152
R-Squared (R^2)	0.8723

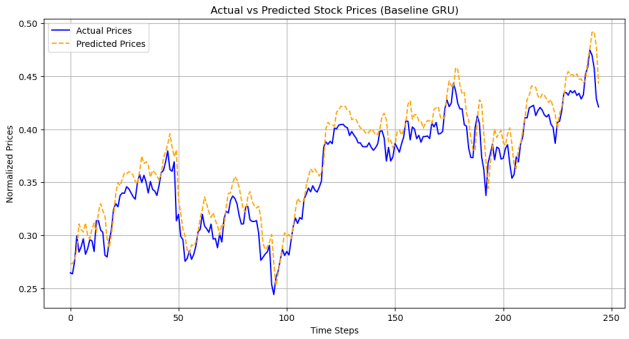

Figure 5. Baseline GRU predictions

The performance is similar to the LSTM in the previous section; the trend is captured slightly better compared to the Vanilla RNN but is lagging behind on spikes and troughs. Further hyperparameter tuning is required in order to compare it with the tuned LSTM model.

Hyperparameter tuning shows the best settings as:

Table 12. Best Hyperparameters for Tuned GRU Model

Hyperparameter	Value
Hidden Size	200
Number of Layers	1
Dropout Rate	0.5
Learning Rate	0.001
Batch Size	16

Final performance of GRU:

The Tuned GRU managed to learn both long-term trends and short-term fluctuations including the valleys and troughs. However, an interesting observation was that it fell short of the Vanilla RNN performance.

Table 13. Performance Metrics for Tuned GRU

Metric	Value
Mean Squared Error (MSE)	0.000438
Mean Absolute Error (MAE)	0.018681
R-Squared (R^2)	0.830558

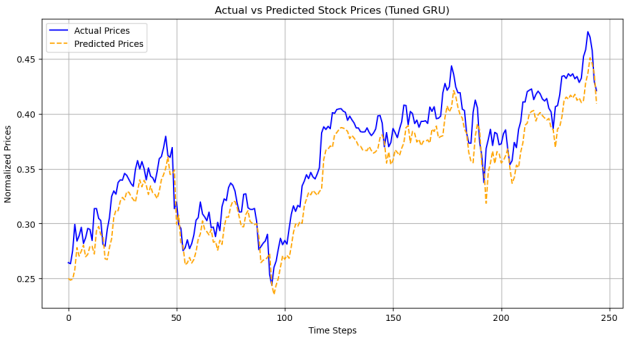

Figure 6. Tuned GRU predictions

9. Results and Analysis

9.1. Results

Table 14 shows the performances of each model before and after tuning.

Table 14. Final Model Performances on Stock Price Prediction

Model	Variant	MSE	MAE	R^2
Vanilla	Baseline	0.000226	0.01145	0.9126
Vanilla	Tuned	0.000301	0.01383	0.8836
LSTM	Baseline	0.000400	0.01380	0.8836
LSTM	Tuned	0.000208	0.01064	0.9195
GRU	Baseline	0.000300	0.01520	0.8723
GRU	Tuned	0.000438	0.01868	0.8306

9.2. Discussion

Each model has their strengths and weaknesses.

1. Vanilla RNN is simple and can be implemented the easiest among the three models. However, Vanilla RNN is unable to capture long-term temporal dependencies and the performance can be weak compared to the other two variants on long term time steps. Even after tuning the hyperparameters, its performance fell short of the other variants and struggled to predict the highs and lows effectively.

2. Long Short-Term Memory has the highest potential capabilities of all models due to its innate architecture and allows it to use its gating mechanisms to capture long-term temporal dependencies. The downside of a LSTM model is the computational cost and training time which can be limiting. The tuned LSTM model in this paper was able

to predict the stock price changes quite effectively. In this paper, the LSTM was able to perform the best out of all three models.

3. Gated Recurrent Units are a slightly more lightweight version of LSTM which has less parameters, states and gates. The lower number of parameters allows a GRU to be more computationally efficient and easier to implement. It is worth noting that a GRU usually does not perform as well as an LSTM on more complex datasets.

It was unexpected that the Vanilla RNN outperformed the GRU, but it could be attributed to the complexity of the dataset. Given that the focus of this paper was to explore the usage of RNN architectures on time-series data, no further experimentation will be conducted on the training of each RNN variant. But it is a hypothesis that more experiments and training performed will likely show the GRU outperforming the Vanilla model over the long run, when longer time steps and more complex datasets are given.

10. Conclusion

This paper demonstrated how effective RNN-based architectures can be in predicting time-series data. An RNN is able to capture temporal dependencies that traditional feed-forward neural networks architectures have difficulties in.

Through extensive training and experimentation, it was found that a LSTM-RNN architecture was best suited for the given task and dataset and is the recommended choice for the assigned task. For future studies that involve more complex datasets and extensive parameters, it is likely that an LSTM model will outperform the other variants in the long run.

11. Code

The codes executed in this paper can be found on my Github repository: https://github.com/EdwardQuah/DLF-Assignment-3-.

References

1. Bengio, Y, Simard, P & Frasconi, P 1994, 'Learning long-term dependencies with gradient descent is difficult', IEEE Transactions on Neural Networks, vol. 5, no. 2, pp. 157–166, viewed 27 November 2024, https://ieeexplore.ieee.org/document/279181.

2. Cho, K, van Merriënboer, B, Gulcehre, C, Bahdanau, D, Bougares, F, Schwenk, H & Bengio, Y 2014, 'Learning phrase representations using RNN encoder–decoder for statistical machine translation', Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, Doha, Qatar, pp. 1724–1734, viewed 26 November 2024, https://arxiv.org/abs/1406.1078.

3. Chung, J, Gulcehre, C, Cho, K & Bengio, Y 2014, 'Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling', arXiv, viewed 25 November 2024, https://arxiv.org/abs/1412.3555.

4. Das, S, Tariq, A, Santos, T, Sandeep, S.K. & Banerjee, I 2023, 'Recurrent neural networks (RNNs): architectures, training, and applications', in O Colliot (ed.), Machine learning for brain disorders, vol. 263, viewed 28 November 2024.

5. Géron, A 2019, Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: concepts, tools, and techniques to build intelligent systems, 2nd edn, O'Reilly Media, Sebastopol, CA.

6. Graves, A, Mohamed, A & Hinton, G 2013, 'Speech recognition with deep recurrent neural networks', Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6645–6649, viewed 28 November 2024, https://ieeexplore.ieee.org/document/6638947.

7. Hochreiter, S & Schmidhuber, J 1997, 'Long short-term memory', Neural Computation, vol. 9, no. 8, pp. 1735–1780, viewed 25 November 2024, https://doi.org/10.1162/neco.1997.9.8.1735.

8. İlhan, F, Karaahmetoğlu, O, Balaban, İ & Kozat, SS 2020, 'Markovian RNN: An Adaptive Time Series Prediction Network with HMM-based Switching for Nonstationary Environments', arXiv, viewed 25 November 2024, https://arxiv.org/abs/2006.10119.

9. Sun, P, Wu, J, Zhang, M, Devos, P & Botteldooren, D 2023, 'Delayed Memory Unit: Modelling Temporal Dependency Through Delay Gate', IEEE Transactions on Neural Networks and Learning Systems, viewed 25 November 2024, https://arxiv.org/abs/2310.14982.

10. Zhang, J, He, T, Sra, S & Jadbabaie, A 2019, 'Why gradient clipping accelerates training: a theoretical justification for adaptivity', arXiv, viewed 26 November 2024, https://arxiv.org/abs/1905.11881.