Stock price prediction using RNNs

Samridhi Gupta

The University of Adelaide

Email id: samridhi.gupta@adelaide.edu.au

Abstract

Stock prices are influenced by numerous factors, making their prediction highly valuable to stock market traders and analysts. Since stock prices rely heavily on historical data sequences, Recurrent Neural Networks are highly successful for analysis and prediction. As RNNs excel at processing sequential data, they are an excellent choice for forecasting stock prices. This study aims to evaluate and compare the performance of different RNN models, including Vanilla RNN, LSTM, and GRU, in predicting stock prices of the Google Stock Price dataset.

1. Introduction and Background

Stock prices are time-series data, meaning that the order of data points is crucial to understanding trends and patterns. Traditional Artificial Neural Networks (ANNs) are not well-suited for stock price prediction because they cannot handle sequential dependencies in data.

ANNs are designed for fixed-size inputs and outputs, and treat data points independently, which leads to the loss of temporal relationships. Moreover, ANNs are prone to overfitting and require extensive feature engineering to effectively model complex patterns in time-series data. In contrast, models like Recurrent Neural Networks, Long Short Term Memory, and Gated Recurrent Units are specifically designed to capture sequential and temporal dependencies, making them more effective for stock price forecasting.

RNNs are good for stock price prediction because they are designed to process sequential data, leveraging their feedback loop to retain information from previous time steps. This ability is crucial for capturing dependencies in stock prices, which are influenced by historical trends. However, traditional RNNs face limitations with long sequences due to issues like vanishing gradients, which hinder their ability to retain distant past information effectively.

Long Short-Term Memory and Gated Recurrent Unit networks address these issues by introducing mechanisms like forget gate, to selectively retain or forget information. This allows them to model both short-term and long-term dependencies more effectively, making them superior choices for complex time-series data such as stock prices [1].

2. Method description

2.1 Vanilla RNN

A Recurrent Neural Network (RNN) is a specialized neural network designed for sequential and time-series data analysis. Unlike feedforward neural networks, RNNs include feedback loops that connect their output from one step to the input of the next, enabling them to retain information about previous steps. A Feedforward Neural Network is one of the simplest forms of artificial neural networks. In an FNN, data moves in a single direction from input to output without feedback loops.

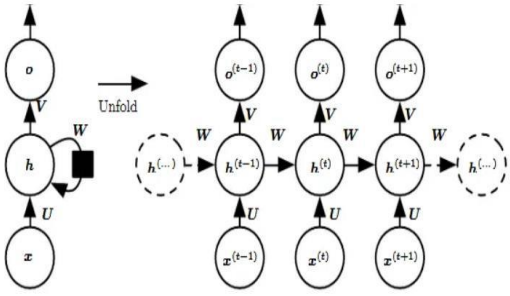


Fig 1. Architecture of an RNN [2]

Here is a short description of the figure:

Input: The input to the RNN at any given time step is denoted as x(t). In the figure, inputs at

three distinct time steps (t-1, t, and t+1), illustrate how the RNN processes sequences over time.

Hidden State: The hidden state, represented by h(t), is computed based on the input at the current time step and the hidden state from the previous time step. This mechanism enables the RNN to retain information from earlier elements in the sequence, effectively maintaining a "memory" of prior data while processing new inputs.

The hidden state at time step t is computed as,

$h(t) = f(U.x(t) + W.h(t-1))$, where function f is the activation function (e.g., tanh or ReLU). This equation combines the current input x(t) with the previous hidden state h(t−1) using weight matrices U and W, allowing the network to retain memory.

Forward Pass: The forward pass in an RNN is represented mathematically as:

$$h(t) = tanh(U.x(t) + W.h(t-1) + b),$$

where tanh is the activation function, and b is the bias term.

Output: The network's output at time step t is given by, $o(t) = V.h(t) + c$ where V is the output weight matrix, and c is the bias term for the output layer.

Loss Calculation: The total loss aggregates all individual losses across all time steps, ensuring that the network learns to optimize its performance for the entire sequence.

In Recurrent Neural Networks (RNNs), the forward pass flows from left to right, followed by the backward pass, which moves from right to left. The backward pass is also known as Backpropagation Through Time (BPTT), as the parameters of the network are shared across time steps. During BPTT, the gradient of an output at any given time step depends not only on the current time step but also on prior time steps. The weights in the network are updated using the formula:

$$W=W-Learning\ rate \times \partial W/\partial x$$

The same is done to weights U, V and biases b and c.

The vanishing gradient problem in RNNs happens when the gradient values diminish to zero during backpropagation. This is due to the nature of RNNs, where gradients are propagated backwards across time steps and multiplied by weight matrices. If these weights are small, the gradient becomes progressively smaller, making it difficult for the model to capture long-term dependencies.

Solutions include using advanced RNN variants like LSTMs and GRUs that are specifically designed to overcome this challenge. Additionally, techniques like gradient clipping can be used to prevent the gradient from becoming too small [4].

2.2 Long-Short Term Memory (LSTM)

The main problem is that it is too difficult for an RNN to preserve information over many timesteps.

The Long Short Term Memory network mitigates the vanishing gradient problem by utilizing mechanisms called gates: the Forget Gate, Input Gate, and Output Gate. These gates work together to manage two types of memory: long-term memory, stored in the cell state, and short-term memory, stored in the hidden state. This architecture enables LSTMs to retain important information over long sequences while discarding irrelevant data, ensuring effective learning of long-range dependencies.

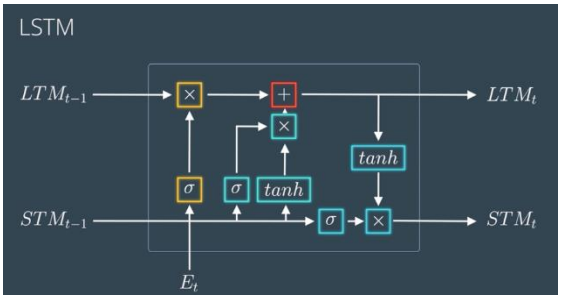


Fig 2. Architecture of the LSTM [1]

2.2.1 Learn gate

The model combines the previous short-term memory (STM_{t-1}) and the current event vector (E_t), applies a weight matrix (W_n), adds a bias,

and passes the result through a tanh activation to create a learn matrix (N_t). To filter insignificant information, it calculates an ignore factor by processing [STM_{t-1}, E_t] with a weight matrix (W_i) and sigmoid activation. Finally, N_t and i_t are multiplied to generate the learn gate output.

2.2.2 Forget gate

The previous hidden state and the current input are fed into a sigmoid-activated neural network, which outputs values between 0 and 1. These values act as weights, indicating the relevance of the input components: close to 0 for irrelevant components and close to 1 for relevant ones. The output is then pointwise multiplied with the previous cell state, reducing the influence of irrelevant components on subsequent steps.

2.2.3 The Remember gate

The outputs of the Forget Gate and the Learn Gate are combined to form the Remember Gate, which updates the long-term memory for the next cell in the sequence.

2.2.4 The Use gate

The model computes the Use Gate output by multiplying U_t and V_t. This result not only represents the output of the current cell but also serves as the short-term memory for the next cell in the sequence. The LSTM model architecture enables LSTMs to handle long-term dependencies effectively [1].

2.3 Gated Recurrent Unit (GRU)

GRUs again, are an improved version of the RNNs. To address the vanishing gradient problem in standard RNNs, GRUs use two gates: the update gate and the reset gate. These gates determine which information to retain or discard, enabling GRUs to preserve relevant long-term dependencies and discard irrelevant data efficiently.

The update gate determines how much past information should be retained for future use, allowing the model to preserve all necessary information. The reset gate determines how much past information to discard, addressing the vanishing gradient problem.

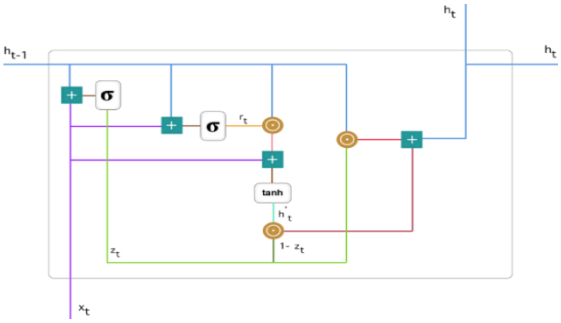


Fig 3. GRU architecture [3]

Input Layer: Processes sequential data like stock prices or sentence words over time. **Hidden State:** Acts as the network's memory, updated at each time step using the current input and the previous hidden state.

Update Gate: The input at the current time step (x_t) is multiplied by its weight $W^{(z)}$, and the hidden state from the previous time step (h_{t-1}) is multiplied by its weight $U^{(z)}$. The results are combined and passed through a sigmoid function, allowing the update gate to decide how much past information to retain.

Reset gate: The formula appears the same as in the previous gate, but it is to be noted that the weights are different.

Candidate Activation Vector: Calculated based on the current time step's input and the reset hidden state from the preceding time step.

Output Layer: Processes the final hidden state to generate the network's output.

GRUs can perform really well on complex problems, provided they are well-trained and they can overcome the vanishing gradient problem.

3. Method Implementation/Code analysis

The code for this study can be found at:

https://github.com/samridhi20-hub/A3_Deep-Learning/tree/main

This section describes the implementation of Vanilla RNN, LSTM, and GRU. The dataset chosen is the Google Stock Price dataset, taken from Kaggle. It's been divided into 2 parts: train and test sets. The train dataset comprises

of 1258 rows and 6 columns. The columns are ‘Date’, ‘Open’, ‘High’, ‘Low’, ‘Close’ and ‘Volume’. Close and Volume are stored as strings and may need conversion for numerical analysis. The test dataset contains 20 rows and the same columns as the train dataset. Here, the ‘volume’ column is stored as a string and needs conversion for numerical operations.

3.1 Data pre-processing and feature selection

The code preprocesses and prepares time-series data for an RNN-based model. The rows in the dataset are inherently ordered by time (chronologically). Even though the Date column is not explicitly utilized, the sequence generation relies on this order to create consecutive timesteps.

It first cleans the Open, Close, High, Low, and Volume columns in the training and testing datasets by removing commas, converting them to numeric types, and dropping any rows with missing values. It then scales the features and target variable (Open) separately using MinMaxScaler to ensure all values are normalized between 0 and 1. To create input sequences for the RNN, the create_sequences_multi function generates sequences of specified timesteps (10 in this case) from the scaled feature data, with corresponding target values. The resulting training and testing sequences are reshaped to a 3D format required by RNNs, with dimensions representing the number of samples, timesteps, and features. Finally, the shapes of the input and output arrays are printed to verify the data's suitability for model training.

Next, the train dataset is split into train and validation datasets. The code uses train_test_split to randomly allocate 80% of the sequences (X_train_seq) and their corresponding target values (y_train_seq) to the training set, while 20% are reserved for validation (X_val_seq and y_val_seq).

3.2 Vanilla RNN implementation

This code defines and compiles a simple RNN model for stock price prediction using the TensorFlow library. The model starts with an RNN layer (SimpleRNN) consisting of 50 units, processing sequential data with a specified input shape and returning only the final output. A Dropout layer with a rate of 0.2 is added to reduce overfitting by disabling some neurons during training at random. Finally, a Dense layer with a single unit outputs the predicted stock price, as it is a regression task. The model is compiled with the Adam optimizer (learning rate: 0.001) and mean squared error (MSE) loss function, suitable for continuous output predictions.

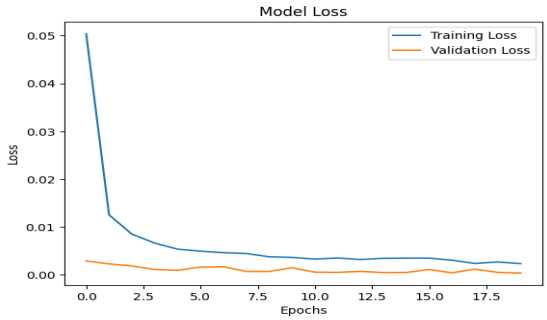
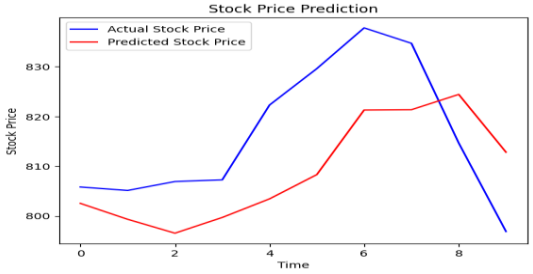


Fig 4. Training loss and validation loss for 20 epochs

The training loss decreases sharply in the initial epochs and then stabilizes, indicating the model is learning effectively during training. The validation loss starts lower than the training loss and decreases further, suggesting the model generalizes well to unseen data without overfitting. While the predicted values follow the general trend of the actual stock prices, there are noticeable deviations, particularly during peak and drop points, indicating room for model improvement. Next, the number of epochs is increased to 50 and the batch size is changed to 16.



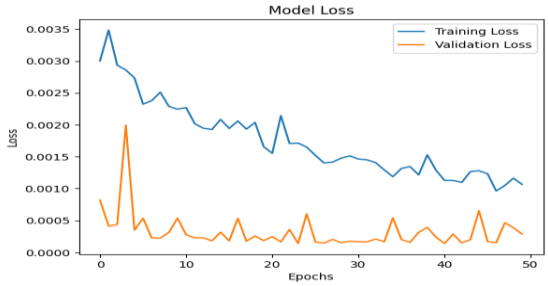


Fig 5. Prediction, training and validation loss, 50 epochs

While the training loss steadily decreases, the validation loss remains relatively stable with small fluctuations, indicating that the model is not overfitting and is generalizing well to unseen data.

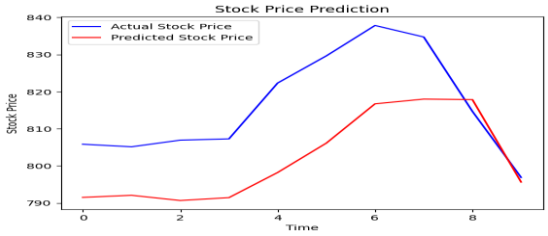


Fig 6. Stock price prediction for 50 epochs

3.2.1 Implementing early stopping

Early stopping is implemented to prevent overfitting by halting training when the model's performance on validation data stops improving, ensuring better generalization. The training stops after the 6th epoch. While the model captures the overall trend, it underestimates during peaks and overestimates during drops, indicating potential for further optimization.

3.2.2 Adding L2 regularization to RNN

Adding **L2 regularization** to the base RNN model helps improve generalization by penalizing large weights in the network. This prevents overfitting, as it discourages The model is prevented from becoming overly dependent on any single feature or pattern in the training data, making the model more robust on unseen data.

This graph indicates a decreasing trend in both, with validation loss consistently lower than training loss, suggesting a good fit without overfitting:

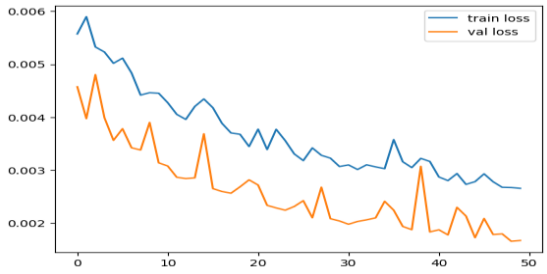


Fig 7. Train and val loss, 50 epochs, regularization

3.3. LSTM model implementation

The first layer is an LSTM layer with 50 units, using the tanh activation function, which processes sequential data and captures temporal dependencies. It does not return sequences; this is a regression task with a single output. The output layer is a Dense layer with 1 unit, suitable for predicting a single continuous value. The model is compiled using the Adam optimizer with a learning rate of 0.001 and a mean squared error loss function, which is common for regression problems. First, the number of epochs is 50 and the batch size is 32. As the number of epochs increases from 20 to 30, 40, 50 and 80, the validation loss also fluctuates, it doesn't show a clear increasing trend. This suggests that the model might be learning generalizable features and could potentially improve with further training or adjustments to the model architecture.

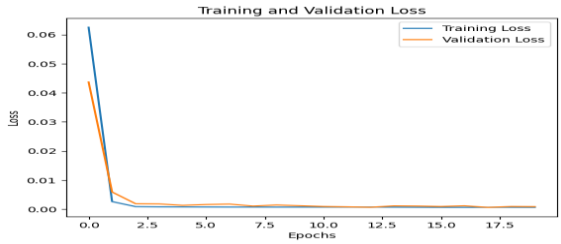


Fig 8. Train and val loss, 20 epochs, LSTM model

Based on these observations, it's difficult to definitively declare one model as the best. However, 50 epochs seem to offer a good balance between performance and overfitting. It achieves a reasonable level of accuracy while avoiding overfitting for a longer period.

Fig 9. 30 epochs train and val loss

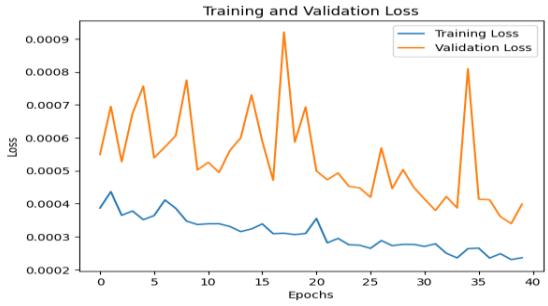


Fig 10. 40 epochs train and val loss

Fig 11. 50 epochs train and val loss

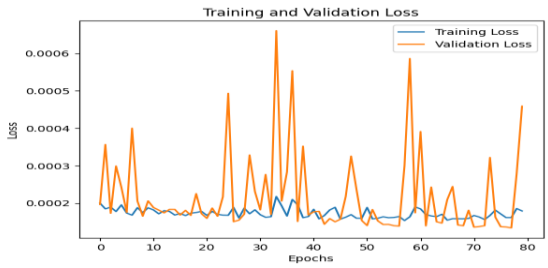


Fig 12. 80 epochs Training and validation loss

3.3.1 Adding L2 regularization to LSTM base model

The previous observation suggests that there is room for improvement and thus, regularization was added to the LSTM base model. Increasing the number of epochs can result in overfitting and struggle with generalizing to new data.

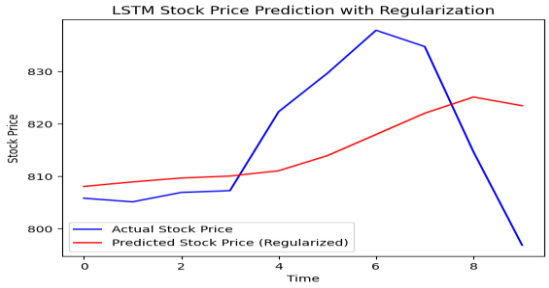


Fig 13. 20 epochs with L2 regularization

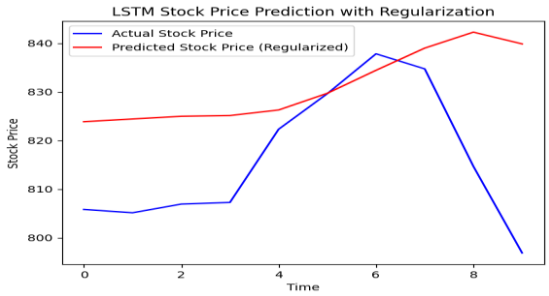


Fig 14. Predicting with 40 epochs

3.4. GRU implementation

A GRU layer with 50 units, using the tanh activation function, which captures temporal dependencies in the input data. It includes recurrent dropout (0.2) for regularization, which randomly drops connections within the GRU's recurrent state to prevent overfitting. A Dropout layer (0.3) is added after the GRU to further regularize the model by randomly dropping neurons during training.

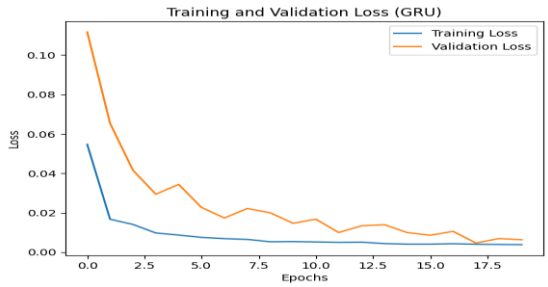


Fig 15. Training and val loss, 20 epochs, GRU

While the training loss decreases steadily, the validation loss initially decreases and then increases. This indicates that the model is overfitting. The predicted price generally follows the trend of the actual price but underestimates the peaks and troughs.

3.4.1 Varying epochs and batch size

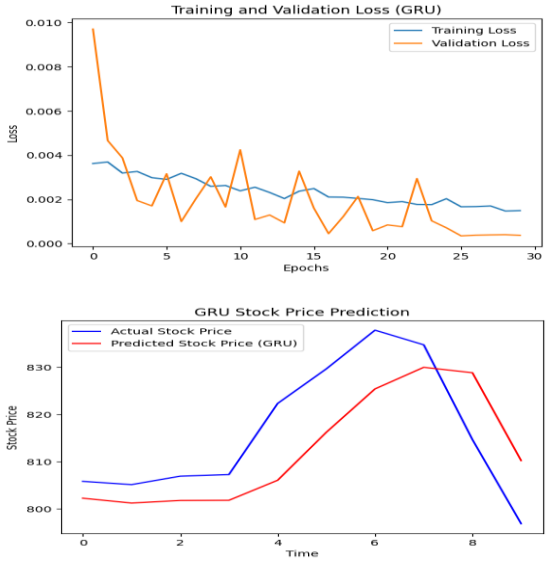


Fig 16. Train, val loss and prediction for 30 epochs

Training for 40 epochs exacerbates the overfitting problem, as evident from the increasing validation loss. Stopping the training at around 20 epochs might have resulted in a better-generalizing model.

Fig 17. Train and val loss, 40 epochs

Training the model on 50 epochs also indicates overfitting. Hyperparameter tuning and adding L2 regularization to the model might be a good solution to prevent this from happening. Also, it would be better to further pre-process the dataset.

Fig 18. Train and val loss for 50 epochs, GRU

4. Experiments and analysis

To evaluate the models, MAE and RMSE were chosen as the evaluation metrics. MAE measures the average absolute difference between predicted and actual values, providing a straightforward understanding of prediction accuracy. RMSE, on the other hand, penalizes larger errors more heavily, making it sensitive to outliers and providing a more rigorous assessment of model performance.

RNN:- The following observations can be made from the RNN implementation:

Epoch	MAE	RMSE	Test loss
20	15.650031	16.831060	0.0009803
50	9.752916	10.521962	0.0009803

Table 1. RNN model

The MAE and RMSE are much lesser with 50 epochs.

MAE	RMSE	Test loss
8.892705	9.739896	0.00032828

Table 2. RNN model with early stopping

The MAE and RMSE have further reduced, indicating the best RNN model among the experiments done so far.

MAE	RMSE	Test loss
14.939406	16.621457	0.00095605

Table 3. RNN model with L2 regularization

LSTM:- The LSTM model was run with various epochs, with batch size 32 and the following was observed:

Epoch	MAE	RMSE	Test loss
20	16.138035	18.931608	0.0009803
30	10.203874	12.520586	0.0009803
40	8.2000332	10.725044	0.0009803
50	11.152916	12.454449	0.0009803
80	10.584914	11.513947	0.0009803

Table 4. LSTM model with batch size 32

It is observed that 40 epochs give the least errors. After this, L2 regularization was added to the base LSTM model and the batch size was changed to 16. Then, the model was run with 20 and 40 epochs respectively.

Epoch	MAE	RMSE
20	10.80874	13.33515
40	15.55274	19.994308

Table 5. LSTM model L2 regularization, batch size = 16

GRU:- This model was tested with varying epochs and batch sizes. The following observations were made:

Epoch	MAE	RMSE	Batch size
20	41.199195	42.635219	32
30	9.2409536	10.410411	16
40	8.908057	11.824903	16
50	10.766444	11.814992	16

Table 6. GRU model with varying epochs and batch size

The error is very high with 20 epochs and decreases sharply with 30 epochs and remains in the same range for 40 and 50 epochs. It's also observed that by the error reduced drastically by changing the batch size from 32 to 16.

5. Reflection/Conclusion and further work

This study aimed to experiment with parameters like the number of epochs and batch size. We also saw how the regularization and early stopping techniques can help increase the accuracy of prediction. Although in this case, we saw an increase in error after regularization, it suggests that the model was overfitting to unseen data and future work can include better data pre-processing techniques like performing Principal Component Analysis or better cleaning techniques. The stock market price dataset has ups and downs and the market is sometimes unstable, having sharp decreases.

The study utilizes the 'Adam' optimizer, but experimenting with alternatives like RMSprop or SGD could provide insights into their effects on the model's performance.

To modify the model's complexity, extra RNN, LSTM, or GRU layers can be included. Additionally, applying dropout layers for regularization can be tested to assess their influence on the model's performance.

6. References

[1] Gourav 2024, *Understanding Architecture of LSTM,* Analytics Vidhya, viewed 5 December 2024, <https://www.analyticsvidhya.com/blog/2021/01/understanding-architecture-of-lstm/>

[2] Nabi, J 2024, *Recurrent Neural Networks (RNNs),* Towards Data Science, viewed 5 December 2024, <https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85>

[3] Kostadinov, S 2017, *Understanding GRU Networks*, Medium, viewed 5 December 2024, <https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be>

[4] Otten, NV 2023, *The Vanishing Gradient Problem, How To Detect & Overcome It*, Spot Intelligence, viewed 5 December 2024, <https://spotintelligence.com/2023/02/06/vanishing-gradient-problem/>