Stock Price Prediction Using Neural Network Architectures : A Comparative Analysis

Sai Akshay Suresh
Faculty of Science, Engineering and Technology
University of Adelaide, North Terrace, Adelaide, South Australia
a1906525@adelaide.edu.au

Abstract

Stock price prediction remains challenging as financial data is filled with complex and inherent non-linear dependencies. In this paper, we investigate the competency of the various neural network architectures, predominantly Recurrent Neural Network (RNN) architectures, based on the progression in terms of complexity for stock prediction. We took a systematic approach containing preprocessing and feature engineering, and the robust dataset obtained from these two processes was tested upon 5 neural network architectures namely: Vanilla RNN, Gated Recurrent Unit (GRU), Long Short-Term Memory (LSTM), Temporal Convolutional Networks (TCN), and a hybrid LSTM-GRU model. Each model was validated with the variation of hyperparameter configurations such as weight initializations, activation functions, and epochs, and advanced regularization techniques such as dropout and gradient clipping were integrated to avoid overfitting. The results of these experiments were analyzed thoroughly with performance indicators such as Mean Absolute Percentage Error (MAPE), Root Mean Squared Error (RMSE), and Directional Accuracy, etc, and autoregression was performed to check if predicted values matched with actual values which highlighted the strengths and vulnerabilities of each architecture. Our findings indicated that specific configuration settings yielded performances of models with superior prediction power for forecasting stock prices, underscoring the need for careful architecture selection and thoughtful preprocessing strategies.

1. Introduction

1.1. Background

Financial markets are taken as an essential driver of all financial activities. The dynamics in the markets often render the precise prediction of stock prices challenging, which is once again a critical requirement for analysts as well as investors. In this research, the Google Stock price dataset employed provides an all-rounded historical record of activity related to the stock markets for four years (2012 through 2016), reflecting the opening and closing prices, along with daily high and low values, and trading volumes. There is a substantial increase in the volume of data. The introduction of deep learning and machine learning methodologies has facilitated the proliferation and efficacy of advanced algorithms that have been devised for modeling and forecasting stock market behavior. Of these methodologies, the Neural Network, especially the Recurrent Neural Network is of specific importance; from the performance metrics of RNN models, patterns can be identified, trends anticipated, and informed decisions can be developed that can influence investment strategies [1].

1.2. Motivation

Conventional techniques used to forecast the stock price are often linear models, which do not reflect the true relationships and dependencies existing in stock market data. Sometimes these models may predict that patterns from the past will persist, leading to incorrect projections. This sometimes leads to poor investment decisions. As computational algorithms grow highly complex in terms of the structure and augmentative elements intended to enhance predictive accuracy through deep learning and machine learning methodologies, remedies for these implications may emerge. The impetus behind this research paper is to identify the most robust model capable of capturing historical patterns among five architectures—predominantly Recurrent Neural Networks (RNNs) such as Vanilla RNN, Gated Recurrent Unit (GRU), Long Short-Term Memory (LSTM), an LSTM-GRU hybrid—alongside a Temporal Convolutional Network for comparison. The paper therefore aims at making the actualization of stock prices by recognizing patterns along with several conditional factors, which then becomes a reliable model for predicting the stock price, thereby bringing together traditional financial analysis with the latest data-driven techniques [2].

1.3. Goals

The objective of this study is to analyze historical stock market data to find trends, patterns, and correlations useful in investment strategies and policy decisions. This will be achieved by evaluating existing models and methodologies

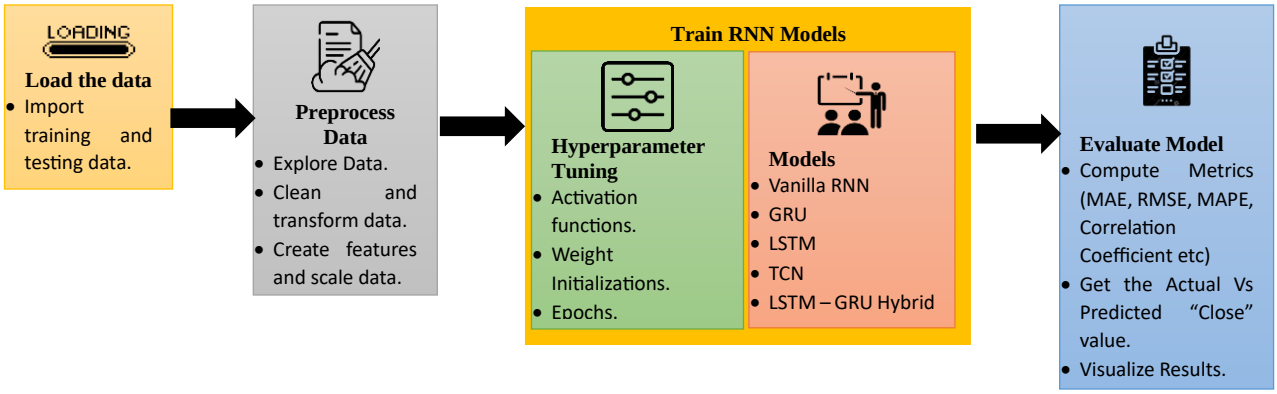

Figure 1, **Flowchart** of the Proposed Workflow

used in stock market analysis with hyperparameter tuning in several combinations while assessing their strengths and weaknesses [3,4]. Training and testing procedures will be quite rigorous and entail the use of historical stock price data for deriving insightful meaning from the stock market while studying the preprocessing methods and feature engineering on the accuracy of the models. The research conducted looks forward to providing solid insights into the novel area of financial data analytics, enabling stakeholders to be informed enough to navigate the complexities of the stock market better. It includes a critical review of relevant literature followed by an in-depth analytical methodology concerning the use of machine learning in financial markets as a feasible tool for decision-making.

2. Methodological Framework

2.1. Overview

The training and testing datasets were first loaded from their designated directories and a short exploratory data analysis was done to verify that their shapes were appropriate to prepare the data for model training. The datasets are then explored to understand their characteristics, checking data types, and summary statistics are computed with an examination of correlations. With the consideration of the exploratory analysis highlights, the data has been pre-processed with data-cleaning, formatting strings, interpreting the accurate dates, and adding lagged features to capture time-based trends. Additional variables, such as day and month were created as an aid to the model's recognition of seasonal and cyclical patterns. The dataset was normalized using a "MinMaxScaler" for the improvement of consistency and stable model training. Furthermore, the training data was enhanced by the addition of Gaussian noise to increase the sample size and diversity for the model to generalize more effectively. Once the data was prepared, it was used for training and validation on the models and subsequently tested on the test data with various recurrent neural network models namely, Vanilla RNN, Gated Recurrent Unit (GRU), Long Short-Term Memory (LSTM), Temporal Convolutional Networks (TCNs), and a hybrid model of GRU-LSTM respectively using different hyperparameters and weight initialization strategies such as epochs, 3 kinds of activations ('relu', 'tanh', 'Leaky relu'), and 'random_normal', 'zeros', 'he_normal' initialization techniques. The prediction capacity of the models was studied with metrics like Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), Mean Absolute Percentage Error (MAPE), and Pearson's correlation coefficient correlation. Additionally, the model's prediction for the response variable (close) was compared to the actual values, and the outcomes were recorded for future reference as given in Figure 1. Visual tools, such as heatmaps, and graphs were effective in assessing the influence of different hyperparameters, thus, guiding us ultimately toward more informed decisions.

2.2. Recurrent Neural Network Architecture

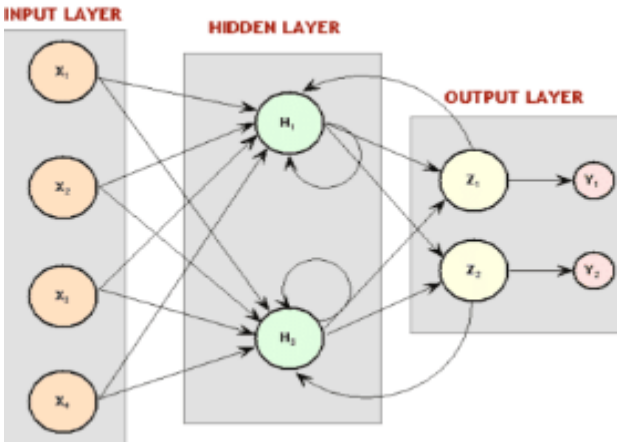

Figure 2, **Recurrent Neural Network Architecture** with its Layers.

The Recurrent Neural Networks (RNNs) are primarily designed to process sequential data by maintaining a hidden state that captures information from previous inputs/past experiences. This flexible architecture, as given in Figure 2, allows RNNs to handle tasks effectively with accurate predictions where the context is essential, such as language modeling or predictions based on time series [5]. The essential components of a general RNN architecture are given below:

a) Input Layer
The input layer receives the input data to be processed, for example, a series of words, time-series data, or any ordered information. Each of the inputs is usually represented in vector format as it allows the model to process complex features of the data. The order of the sequences is maintained as the inputs are fed into the model one at a time, which is important for learning dependencies [5].

b) Hidden Layer
The hidden layer contains the recurrent units to maintain a hidden state, which will effectively act as memory. The hidden state is updated at every time step depending on the present input and the previous hidden state. A feedback loop is present, which is useful when the output from the hidden layer at each time step is fed back into itself for the next time step, allowing the network to retain information over time. Parameter sharing is done, unlike traditional neural networks, as the RNN shares the parameters across all the time steps, thereby reducing complexity and improving generalization across sequences of data [5].

c) Output Layer
The output layer predicts based on the hidden state at every time step. It can generate outputs at every step or just at the final step based on the architectural complexity. The output layer structure can vary based on the task, for example, for tasks like image captioning or a single output for sentiment analysis, the output layer may produce a sequence of outputs. Commonly, activation functions like "SoftMax" are used in the output layer for classification tasks or linear functions for regression tasks [5].

d) RNN Unfolding

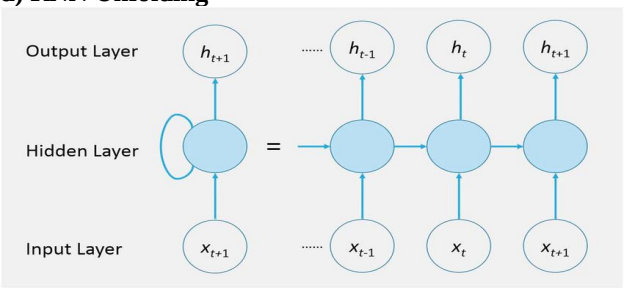

Figure 3, An **Unfolded RNN** with the layers.

Unfolding here refers to the representation of RNN across multiple time steps as an intense computational graph, and each time step is treated as an individual layer as shown in Figure 3. The Back Propagation Through Time (BPTT) technique allows the gradients to flow backward through all time steps which enables effective weight updates across the whole sequence and improvement of temporal dependencies [5]. The unrolling helps in visualization of the information flow through the network over time which will illustrate dependencies between inputs and outputs through different time steps.

2.3. Models

2.3.1 Vanilla Recurrent Neural Network Model

The Vanilla RNN employs a simple recurrent unit in which the next hidden state is a direct function of the previous hidden state and the current input. It basically consists of a single hidden layer where the same weights are applied across all the time steps. The input sequences are processed for one element at a time which maintains a hidden state that captures information from the previous inputs for the accumulation of insights into the temporal dependencies i.e. the relationships between previous and current data points [6].

Vanilla RNN serves as a baseline model for comparison of more sophisticated recurrent architectures and shows initially how temporal dynamics can influence predictions. The RNN architecture is found to have the program of Vanishing Gradient, where it is challenging for the model to learn long sequences as the gradients diminish during backpropagation through time [6]. Tasks like simple sequence prediction and basic language modeling are suitable to be performed with Vanilla RNNs but, this architecture struggles with more complex tasks that demand long-term memory. Vanilla RNNs are considered for this research to establish a foundational benchmark against which we can track the improvements provided by more complex architectures.

2.3.2 Gated Recurrent Unit (GRU) Model

GRU introduces new updates and reset gates in better control of the flow of information, thus, making training more stable than the Vanilla RNNs. Unlike the LSTM RNN, which has cell states along with hidden states, it is based on a combined hidden state. The design also makes it more computationally efficient yet effective. The GRU also reduces both complexity and the number of total parameters than LSTMs, which often exhibit faster convergence and still handle long dependencies fine [7]. The update gate determines the amount of past information to retain and the amount of new information that needs to

be included. The reset gate calculates the amount of past state information to forget. Thus, GRUs will perform in a balanced way with efficiency since they are a middle point between simple RNNs and RNNs with the complex gating technique.

GRUs are preferably used when there is data containing long-term dependencies, such as natural language processing and time series prediction. This paper makes use of GRUs in exploring a more powerful yet efficient architecture to preserve context for long sequences without adding complexity similar to that of LSTMs [7].

2.3.3 Long Short-Term Memory (LSTM) Model

The Long-Short-Term Memory (LSTM) includes three gates: input, forget, and output gates with a state cell that controls the inflow and outflow of information into memory cells; this architecture allows selecting information that LSTMs remember as well as forgetting it so it better holds the long-term dependents while having fewer difficulties in the issue of vanishing gradients as in Vanilla RNNs [8].

LSTM has been effective because it can capture subtle patterns in sequential data with improved long-horizon prediction accuracy, and it has been applied in many sequence-based tasks such as language modeling, time series-based forecasting, and speech recognition. Often, LSTM is referred to as a "go-to" advanced RNN model owing to its flexibility and wide acceptance by researchers [8]. The choice of LSTM architecture in this study is based on its established reputation as a commonly used framework that effectively manages extended sequences, thus providing a reliable performance standard.

2.3.4 Temporal Convolutional Network (TCN) Model

The Temporal Convolutional Network uses convolutional layers that support parallel processing on time steps and thus encourage more efficient gradient flow. This is achieved through causal convolutions where predictions at every time step t are dependent on only the previous and current inputs. Additionally, the use of dilated convolutions increases the receptive field without multiplying the parameters, making the TCNs able to well capture the long-range dependency without needing a gated recurrent connection [9,10].

Recently, TCNs have been shown to outperform traditional RNN models for many sequential and time-series tasks in terms of the ability to design complex temporal dependencies without suffering from the vanishing gradient problem [10]. Besides, TCNs have higher stability and improved parameter efficiency over traditional architectures of RNNs. TCNs are added in this paper to explore a non-recurrent approach that could naturally seek long-term dependencies and help improve training efficiency.

2.3.5 LSTM-GRU Hybrid Model

The LSTM-GRU hybrid model architecture is like the gating structure of LSTM and GRU, in that it combines both qualities with the potential to do better on complex sequence tasks. The hybrid model is primarily designed to balance the merits of the LSTM's performance on a long sequence with the training efficiencies of the GRUs. Hybrid models often use mechanisms of attention, allowing the network to dynamically focus on relevant parts of the input sequences, thereby making them also interpretable especially for applications such as machine translation [11]. The hybrid architecture of the LSTM-GRU is thus modified according to specific application needs, modifying functional units of LSTM and GRU while retaining complexity and computational efficiency with better efficiency and generalization.

The hybrid architecture of the LSTM-GRU has shown that in cases when the short time to react and long-memory storage are both in need, it will do better than individual architecture. It looked at a hybrid architecture to see whether the components from both LSTM and GRU may be integrated to bring better efficacy and feasibility into our forecasting methodology [11].

2.4 Hyperparameter Setting

The following hyperparameters, as given in Table 1, were used before training each of the architecture to control the training process,

Hyperparameter	**Varieties**
i) Activation Function	relu, tanh, Leaky relu.
ii) Weight Initializations	random_normal, zeros, he_normal.
iii) Epochs	10,20,30,40,50,100

Table 1, A **Description of Hyperparameters** with their variations.

The hyperparameter variations were done in a fashion, where, for each of the activation functions, every weight initialization technique was tested, and for each of these varieties, every epoch variety was tested.

2.4.1 Activation Functions

Activation functions are used in the introduction of non-linearity to neural networks to enable them to learn

complex patterns [12]. The neural networks would behave like a linear regression model irrespective of the number of layers without the activation functions. The impact on training stability, and convergence speed, along with the model's ability to capture the underlying temporal dependencies can be evaluated with the usage of various activation functions. The experimented activation functions are discussed as follows:

i) ReLU - The Rectified Linear Unit (ReLU) function outputs negative values to zero and leaves the positive values unchanged. It accelerates the training speed and also helps in removing vanishing gradient problems to a large extent. The ReLU is a simple technique and has been found to perform well in an enormous machine-learning domain, serving as a baseline activation for comparison against more complex variants [12]. **ii) tanh (Hyperbolic Tangent)** - tanh takes both positives and negatives and rescales the input to stay in the interval [-1,1]. tanh is generally more popular than the sigmoid function due to its decreased likelihood of saturation and reduced issues of vanishing gradients [12]. By contrasting tanh with ReLU, it will be shown whether centering around zero benefits learning in a problem of our interest- here being stock price prediction. **iii) Leaky ReLU** - A variation of ReLU, where it gives a very small, not zero gradient for the negative value. This function prevents dead neurons by making possible gradients flow even if negative input is given to neurons. The performance of leaky ReLU often surpasses standard ReLU in many applications. It is claimed that Leaky ReLU addresses the dead "ReLU" neurons by providing a small but consistent gradient for negative inputs. This ensures that more neurons remain active during training, improving the robustness and performance of the networks, especially when input distributions or learned features change significantly over time [12].

2.4.2 Weight Initializations

Weight initialization is giving initial values to the model's weights before the actual training begins. A good initialization will ensure good gradient-based optimization and will also allow stable signal transmission in the network. Bad weight initialization can cause many problems: slow convergence, vanishing or exploding gradients, in general, poor performance. An appropriate choice of good weight initialization is a very important player in fostering stable training and convergence. The following discusses the various weight initialization techniques tested [13].

i) Random Normal - It takes its weights from a normal distribution. This is to break the symmetry among neurons because of the randomness. Random normal initialization was used to induce intrinsic noise in the network right from the beginning, thereby breaking the symmetry among the neurons. This is well-known and well-established and therefore is a good control to compare against [13]. **ii) Zero initialization (with slight noise deviation)** - All the weights are initialized with zero or values close to zero. Traditional zero initialization leads to uniform outputs and is against good learning. Introducing a small amount of noise tries to overcome this problem and allows some randomness and potentially better initial search. Pure zero initialization has been known to be problematic where no learning occurs. Now introducing minimal noise around the zero allowed us to determine how extremely small initial differences affect convergence. This would give us an idea of how sensitive the initialization is and whether small perturbations were able to guide the model toward reasonable solutions [13]. **iii) He Normal (He et al. Initialization**) – It samples weights from a scaled normal distribution conditioned on the number of input units. This should ensure that the variance of signals remains constant across layers. Typically used in combination with ReLU-based activations, this tends to result in faster and more stable training. He initialization was chosen because it has been particularly designed to perform well when combined with ReLU-like activations. It ensures appropriate signal variance across layers, thereby reducing the likelihood of vanishing gradients significantly and enabling smooth training, especially in deep models [13].

2.4.3 Epochs

An epoch is taken to be the complete pass of the training dataset, which enables the model to improve its parameters. A low number of epochs will result in underfitting because the model does not get enough exposure to the data, while too many epochs result in overfitting since the model over-trains on the training patterns. We chose epoch values of 10, 20, 30, 50, and 100 to test performance across different training times and to identify a balance [14].

2.5 Performance Metrics

Performance metrics such as MAE, RMSE, and MAPE are needed to evaluate the predictions from the model [15]. These translate the output into easily interpretable measures of quality. Such analysis helps in choosing the models and in tuning such a model, ensuring the patterns are being captured by it. The following metrics were computed for the evaluation:

i) Mean Absolute Error (MAE) – MAE means to measure the average absolute difference between actual values and predicted values

$$\mathbf{MAE} = \frac{1}{n}\sum_{i=1}^{n} |\mathrm{yi} - \hat{y}i|$$

ii) Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) - This is known as the mean

squared error, and the square root of the mean squared error is called the root mean squared error, so it matches the unit of the error to the original data.

$$\mathbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\mathrm{yi} - \hat{y}i)^2$$

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

iii) Mean Absolute Percentage Error (MAPE): Indicates the average percentage error relative to actual values, providing insight into relative performance.

$$\mathbf{MAPE} = \frac{100\%}{n}\sum_{i=1}^{n}\frac{|\mathrm{yi} - \hat{y}i|}{yi}$$

iv) Correlation Coefficient (Pearson's r): Evaluates the linear relationship between predictions and actual values.

$$\mathbf{r} = \frac{\sum_{i=1}^{n}(\mathrm{yi} - \hat{y})(\hat{y}\mathrm{i} - \hat{y})}{\sqrt{\sum_{i=1}^{n}(\mathrm{yi} - \hat{y})^2 \sum_{i=1}^{n}(\hat{y}\mathrm{i} - \hat{y})^2}}$$

v) Directional Accuracy: Assesses how often the model correctly predicts the direction of changes.

$$\mathbf{DA} = \frac{\text{Number of Correct Direction Predictions}}{\text{Total Number of Observations}}$$

vi) Generalization Gap (MAE): The difference between testing and training MAE, indicating how well the model generalizes to unseen data.

Generalization Gap = Testing MAE−Training MAE

The code uses Keras's model. evaluate() function to calculate MAE over both the training and the test datasets:

vii) Training MAE: Following the training, model. evaluate(x_train, y_train, verbose=0) yields the MAE on the training set [16]. The output values contain both loss and MAE metrics since the model was compiled with loss='mean_absolute_error' and metrics=['mae'].

viii) Testing MAE: Similarly, model. evaluate(x_test_full, y_test_full, verbose=2) computes the MAE on the test set [16]. Again, the returned values include the MAE loss and metric, so that one can make a direct comparison with the training MAE to determine if the model generalizes.

3. Experimental Evaluations

3.1 Experimental Setup

A Google stock price dataset which contained 1258 observations with four columns (open, high, low, close, and volume) was used for training and validation. This training dataset contained prices for four consecutive years (2012 – 2016). Test data with 10 observations for the year was used in testing the model for 10 different dates of a particular month. The "close" variable was considered the response variable. All the architectures were tested on all the hyperparameter variants. The experiments were performed in Python via Google Colab.

3.2 Data Preparation

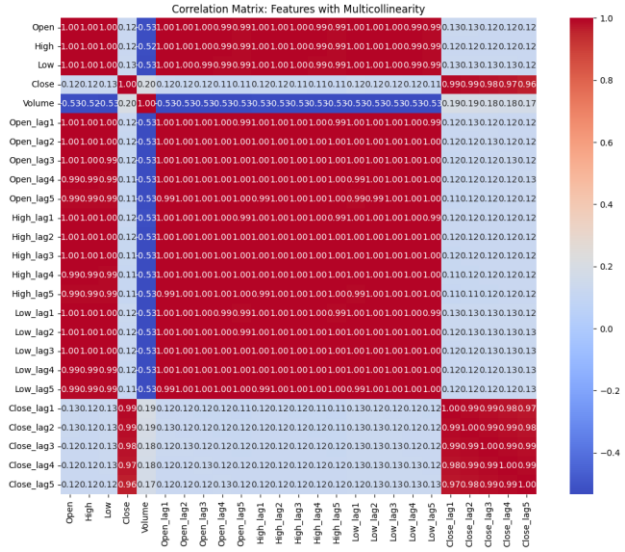

Figure 4, **Multicollinearity Matrix Before** choosing two features.

Figure 5, **Multicollinearity Matrix After** choosing two features.

The raw dataset was imported and reviewed for dimensions, columns, and basic statistics to catch any errors such as missing values or wrong types of data. The numerical features "Close" and "Volume" were first encoded as comma-separated strings; it was converted to a float type for correct numerical computations. The "Date" column was converted into "DateTime" format to get date-based features. Time lags were included, since, the lead target variable is "Close," including its lagging values-for example, "Close_lag1" and "Close_lag2"-gave the model some historical context in terms of time, and hence, lagged versions of significant predictors are developed [17]. Focusing solely on "Close" and its lags, along with "Volume, while the less useful features like "Open," "High," and "Low" as shown in Figure 4, are very highly

correlated with "Close", which were also dropped along the way that reduces the dimensionality, and multicollinearity also got reduced as shown in Figure 5. This resulted in a more understandable correlation structure and better ordering of the input space.

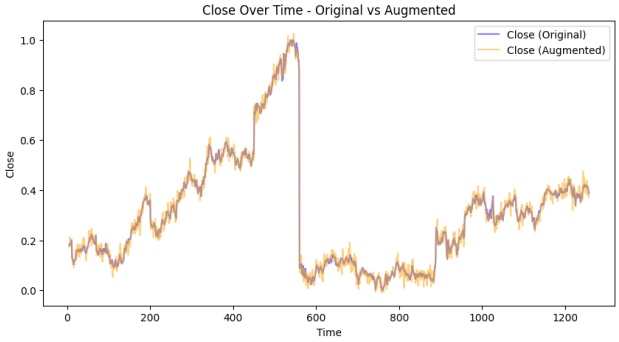

Figure 6, **Original Vs Augmented** Data for Close Feature.

After these preprocessing steps, all numeric features were scaled with MinMaxScaler, which provided a uniform scale for features and stability in the optimization process. For better robustness, as shown in Figure 6, Gaussian noise was added to the training set as a data augmentation technique [18]. The dataset was divided into two parts: training (75%), and validation (25%), which ensured an unbiased hyperparameter optimization process and a reliable estimate of the model's performance on unseen data.

3.3 Results and Analysis

All the layers, activation functions, and weight initializations were done using the TensorFlow Keras framework. Modules tensorflow.keras.models and tensorflow.keras.layers were used for layer development and activation functions. Custom weight initializations and pre-existing initializers were also taken from tensorflow. keras.initializers [19].

The Vanilla RNN results show that using the tanh activation function consistently yields superior performance, with Testing MAEs as low as 0.0006 and perfect Directional Accuracy (1.0000). Weight initializations of random_normal or zeros with tanh provide excellent outcomes, reflected in nearly perfect Correlation Coefficients and minimal RMSE. While ReLU and Leaky ReLU configurations occasionally perform decently, they do not surpass tanh-based setups. Overall, tanh with proper initialization stands out as the best choice, delivering both high predictive accuracy and almost a stable generalization, as indicated by negative or minimal Generalization Gaps.

GRU models also have a great advantage when using tanh activation, paired with either random_normal or zeros initialization, almost optimal Correlation Coefficients and Directional Accuracy, and Testing MAE reaches as low as 0.0012. While the ReLU variants are good enough, they hardly achieve this level of accuracy and robustness with configurations from tanh. One configuration of Leaky ReLU, using zeros initialization had a Testing MAE of 0.0029 and a significant Correlation Coefficient of 0.9660. GRUs with the tanh activation function provide the best trade-off between reduced error metrics, strong directional forecasting, and minimal generalization differences.

For LSTMs, tanh activation with either zeros or random_normal initialization is working quite well, producing Testing MAEs as low as 0.0014 and a Directional Accuracy of 1.0000. The remaining LSTMs-those that use ReLU for activation more than adequate but still fragile, providing a tendency of higher MAE and looser correlations. Sometimes leaky variants of ReLUs outperform but otherwise do not surpass tanh-dominated settings. In a word, tanh activation invariably allows LSTMs to approach near-ideal predictive performance metrics, thus further vindicating its use on complex temporal patterns and generalizing well.

TCNs are remarkable when tanh and some of the initializations are used; this gives very low Testing MAE (even as low as 0.0005) with perfect Directional Accuracy and nearly perfect Correlation Coefficients. The ReLU-based TCNs are generally from fair to good in performance and, when using Leaky ReLU with zeros or random_normal initialization, they perform pretty well. Most importantly, the tanh TCNs can match or even do better than the best recurrent models in both accuracy and directionality. The activation of tanh for TCNs leads to very high precision and strong generalization.

The LSTM-GRU hybrids again behave according to the trend from the previous architectures-that is, the winner turns out to be tanh. Random_normal, zeros, or he_normal initialization with tanh activation turns out to have almost perfect correlation and directional accuracy. Some of the ReLU or Leaky ReLU variants perform well, though never better than the best tanh-based configurations. The hybrid technique effectively taps the benefits of both LSTM and GRU cells; judiciously combined with some proper activation functions and initialization techniques, it will deliver excellent predictive accuracy with no error rates and generalized gaps.

Arc	Act	Init	Ep	Train MAE	Test MAE	Corr. Coef	Dir. Acc	RMSE	MAPE	Gen. Gap (MAE)
Vanilla	tanh	random_normal	50	0.0021	0.0006	0.9989	1.0000	0.0008	0%	-0.0015
GRU	tanh	he_normal	100	0.0038	0.0044	0.9985	1.0000	0.0046	1%	0.0006
LSTM	Leaky relu	zeros	100	0.0058	0.0053	0.9445	0.9286	0.0064	1%	-0.0005
TCN	Leaky relu	random_normal	40	0.0022	0.0018	0.9994	1.0000	0.0020	0%	-0.0004
LSTM-GRU	tanh	he_normal	100	0.0022	0.0010	0.9962	1.0000	0.0014	0%	-0.0012

Table 2, **Peak of Peaks Table for** All the Architectures

The "peak of peaks" table as shown in Table 2, is exemplary on multiple architectures and configurations, with the peak of peaks highlighted in yellow. The LSTM-GRU hybrid RNN with tanh and he_normal initialization, achieves a Testing MAE of 0.0010, implying its correlation coefficient is almost perfect with 0.9962 and a perfect Directional Accuracy with 1.0000. Other top performers are Vanilla RNN with tanh and random_normal initialization, which achieve a high correlation and directional accuracy. All of the above configurations display very low errors, very low MAPE, and negligible Generalization Gaps.

Conceptually, GRUs and LSTMs are variations over vanilla RNNs where the ability to capture long-term dependencies is achieved due to gating mechanisms in those architectures [20]. From this experiment, it follows that, under appropriately chosen conditions of activation functions and weight initialization, the results of a simple Vanilla RNN are nearly ideal at predicting accuracies. However, in practice, GRUs and LSTMs usually yield more stable and scalable performance with the increasing complexity of time-series patterns, which indeed confirms their theoretical advantage in handling extended temporal dependencies.

The autoregressive testing is implemented as shown in Table 3, to affirm the consistency between observed prices and forecasted prices, thus strengthening the conclusions that can be derived from our earlier analyses. Various architectures including Vanilla RNN, GRU, LSTM, TCN, and LSTM-GRU show that configurations using the tanh activation function with proper weight initialization, often random_normal or zeros, consistently have the lowest Testing MAE, near-optimal Directional Accuracy, and minimal MSE. Vanilla RNNs, with tanh and proper initialization, present differences between "Close" actual and predicted values to be extremely small, often almost to zero error. Models of GRU and LSTM follow the trend; tanh-based architectures far outperform ReLU or Leaky ReLU-based setups. TCN models also under certain conditions are particularly effective and stability in many situations match or even beats the best recurrent architectures. Finally, the hybrids of LSTM-GRU optimized by tanh

Date	Act	Init	Ep	Close (Act)	Close (Pred)	Dif	MSE
10-01-2017	tanh	he_normal	10	0.4322	0.4300	-0.0022	0.0000
11-01-2017	tanh	he_normal	20	0.4365	0.4422	0.0057	0.0000
12-01-2017	tanh	he_normal	30	0.4343	0.4339	-0.0005	0.0000
13-01-2017	tanh	he_normal	40	0.4364	0.4362	-0.0002	0.0000
17-01-2017	tanh	he_normal	50	0.4319	0.4346	0.0026	0.0000
18-01-2017	tanh	he_normal	100	0.4339	0.4327	-0.0012	0.0000

Table 3, A **Sample Testing Result** for LSTM-GRU Hybrid

initialization confirms that focusing on an appropriate activation function and weight initialization strategy leads to robust, precise, and stable predictions in stock price forecasting.

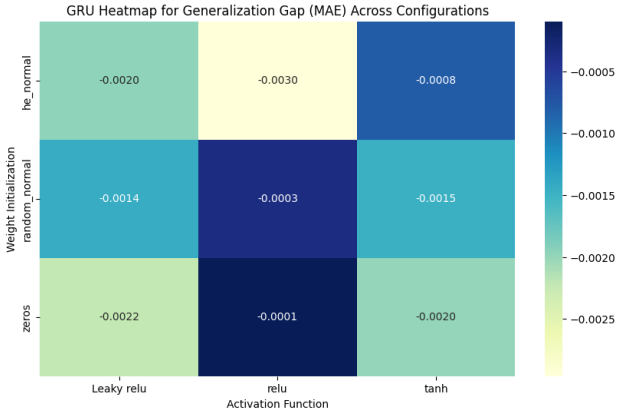

Figure 7, **Heatmap of Generalization** Gap for GRU across configurations

Heatmaps were visualized as shown in Figure 7, for all the architectures across configurations. The graphical analysis was done for i) Line Graphs for comparing Epochs vs. metrics according to weight Initialization techniques for Architectures across different activation functions as displayed in Figure 8, ii) Peaks comparison of "weight initialization-activation function" Vs metrics with the variation of Epochs for Architectures, iii) Peak of Peaks

comparison of "Architectures-Activation-Weight Initialization" Vs Metrics with bar charts as represented in Figure 9, iv) Comparison of Actual and Predicted Values for Vanilla RNN architecture with a Line Plot as presented in Figure 10.

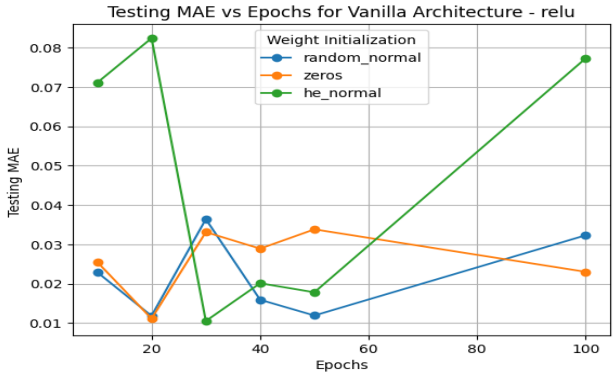

Figure 8, **Line Graph for Epochs Vs Testing MAE** across Configurations of Vanilla Architecture

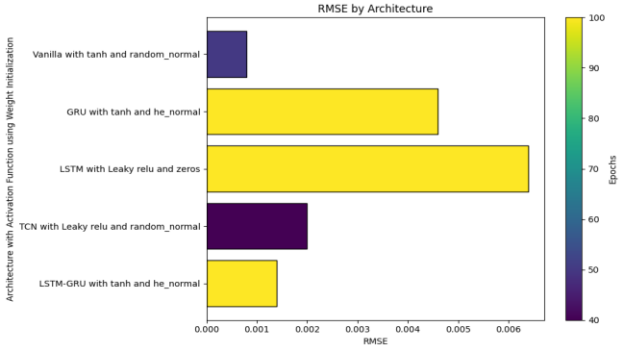

Figure 9, **Comparison of Peak of Peaks RMSE** by Architecture according to the variation of Epochs.

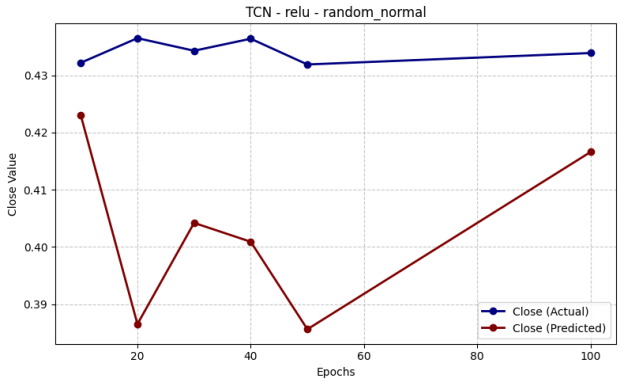

Figure 10, **Comparison of Actual and Predicted** Values for TCN architecture.

4. Codes and Folders

The codes, folders, results, and proofs for implementation can be viewed at https://github.com/a1906525/RNN.git.

5. Conclusion

To conclude, all our careful experiments, from the baseline Vanilla RNN to the sophisticated GRU, LSTM, TCN, and LSTM-GRU configuration-reveal that a good choice of activation functions and weight initialization procedures are quite decisive for the correctness of the stock price predictive model. The integrated findings validate the earlier works: models that include tanh activation with random_normal or zeros initialization have always obtained the lowest Testing MAE, almost perfect Directional Accuracy, and less MSE. These effects consistently pervaded throughout the full design hierarchy as Vanilla RNNs well-optimized would approximate very nearly zero errors. Many complex techniques like GRU, LSTM, and TCN had favor under the tanh, setups and LSTM/GRU hybrids proved improved Activation and initializations, choice result in robust Accurate and ultra-stable predictors.

Despite all this, there are many problems left. Financial time series of real-world applications often contain complicated non-stationarity and the optimization procedure for hyperparameters is still very computationally intensive. Further directions could be towards adaptive frameworks that learn to adapt to changes in the market environment, can use external economic variables for better contextual understanding, and can use more efficient search algorithms for model selection. This pathway may lead us to robust, interpretable, and scalable frameworks for forecasting toward the ultimate development of a suite of informative, data-driven tools within the financial sector.

6. References

[1] G. R. Kanagachidambaresan, A. Ruwali, D. Banerjee, and K. B. Prakash, "Recurrent Neural Network," Programming with TensorFlow, pp. 53–61, 2021, doi: https://doi.org/10.1007/978-3-030-57077-4_7. 1

[2] H. Song and H. Choi, "Forecasting Stock Market Indices Using the Recurrent Neural Network Based Hybrid Models: CNN-LSTM, GRU-CNN, and Ensemble Models," Applied Sciences, vol. 13, no. 7, p. 4644, Apr. 2023, doi: https://doi.org/10.3390/app13074644. 1

[3] K. E. Hoque and H. Aljamaan, "Impact of Hyperparameter Tuning on Machine Learning Models in Stock Price Forecasting," *IEEE Access*, pp. 1–1, 2021, doi: https://doi.org/10.1109/access.2021.3134138. 2

[4] B. Dinda, "Gated recurrent neural network with TPE Bayesian optimization for enhancing stock index

prediction accuracy," arXiv.org, 2024. https://arxiv.org/abs/2406.02604 (accessed Dec. 05, 2024). 2

[5] S. Das, A. Tariq, T. Santos, Sai Sandeep Kantareddy, and I. Banerjee, "Recurrent Neural Networks (RNNs): Architectures, Training Tricks, and Introduction to Influential Research," Neuromethods, pp. 117–138, Jan. 2023, doi: https://doi.org/10.1007/978-1-0716-3195-9_4. 3

[6] S. M. Al-Selwi et al., "RNN-LSTM: From applications to modeling techniques and beyond—Systematic review," Journal of King Saud University - Computer and Information Sciences, vol. 36, no. 5, p. 102068, Jun. 2024, doi: https://doi.org/10.1016/j.jksuci.2024.102068. 3

[7] M. Fairuzabadi, K. Kusrini, E. Utami, and A. Setyanto, "Advancements and Challenges in Gated Recurrent Units (GRU) for Text Classification: A Systematic Literature Review," 2024 7th International Conference of Computer and Informatics Engineering (IC2IE), pp. 1–7, Sep. 2024, doi: https://doi.org/10.1109/ic2ie63342.2024.10748229. 3,4

[8] H. N. Bhandari, B. Rimal, N. R. Pokhrel, R. Rimal, K. R. Dahal, and R. K. C. Khatri, "Predicting stock market index using LSTM," Machine Learning with Applications, vol. 9, no. 100320, p. 100320, May 2022, doi: https://doi.org/10.1016/j.mlwa.2022.100320. 4

[9] P. Lara-Benítez, M. Carranza-García, J. M. Luna-Romera, and J. C. Riquelme, "Temporal Convolutional Networks Applied to Energy-Related Time Series Forecasting," Applied Sciences, vol. 10, no. 7, p. 2322, Mar. 2020, doi: https://doi.org/10.3390/app10072322. 4

[10] P. Janssen, "Attention based Temporal Convolutional Network for stock price prediction," 2022. Accessed: Dec. 05, 2024. [Online]. Available: https://studenttheses.uu.nl/bitstream/handle/20.500.12932/41588/Master_Thesis_definitive.pdf? 4

[11] N. S. Chauhan and N. Kumar, "Traffic Flow Forecasting Using Attention Enabled Bi-LSTM and GRU Hybrid Model," Communications in computer and information science, pp. 505–517, Jan. 2023, doi: https://doi.org/10.1007/978-981-99-1648-1_42. 4

[12] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," Neurocomputing, vol. 503, pp. 92–108, Sep. 2022, doi: https://doi.org/10.1016/j.neucom.2022.06.111. 5

[13] M. V. Narkhede, P. P. Bartakke, and M. S. Sutaone, "A review on weight initialization strategies for neural networks," Artificial Intelligence Review, Jun. 2021, doi: https://doi.org/10.1007/s10462-021-10033-z. 5

[14] Maswan Pratama Putra and Yuliant Sibaroni, "Effect of Epoch Value on the Performance of the RNN-LSTM Algorithm in Classifying Lazada App Review Sentiments," *Sinkron*, vol. 8, no. 2, pp. 918–928, Mar. 2024, doi: https://doi.org/10.33395/sinkron.v8i2.13368. 5

[15] A. Kumar, "MSE vs RMSE vs MAE vs MAPE vs R-Squared: When to Use?," Analytics Yogi, Dec. 29, 2023. https://vitalflux.com/mse-vs-rmse-vs-mae-vs-mape-vs-r-squared-when-to-use/ 5

[16] E. R. Collins, "5 Best Ways to Evaluate Your Model with Keras in Python – Be on the Right Side of Change," Finxter.com, Mar. 08, 2024. https://blog.finxter.com/5-best-ways-to-evaluate-your-model-with-keras-in-python/ 6

[17] O. Surakhi et al., "Time-Lag Selection for Time-Series Forecasting Using Neural Network and Heuristic Algorithm," Electronics, vol. 10, no. 20, p. 2518, Oct. 2021, doi: https://doi.org/10.3390/electronics10202518. 7

[18] H.-X. Dou, X.-S. Lu, C. Wang, H.-Z. Shen, Y.-W. Zhuo, and L.-J. Deng, "PatchMask: A Data Augmentation Strategy with Gaussian Noise in Hyperspectral Images," Remote Sensing, vol. 14, no. 24, pp. 6308–6308, Dec. 2022, doi: https://doi.org/10.3390/rs14246308. 7

[19] "Module: tf.keras | TensorFlow Core v2.4.1," TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras. 7

[20] R. Cahuantzi, X. Chen, and S. Güttel, "A comparison of LSTM and GRU networks for learning symbolic sequences," arXiv:2107.02248 [cs], Jul. 2021, Available: https://arxiv.org/abs/2107.02248. 8