# Report on LSTM-RNN Networks and Using Them to Predict Stock Prices

Ben Tennyson
Undergraduate of Computer Science
University of Texas at Dallas
Plano, Texas
bwt220000@utdallas.edu

***Abstract*— This report aims to demonstrate that I have gained a basic grasp of how Long-Short Term Memory Recurrent Neural Networks (LSTM-RNNs as they will be referred to throughout most of the report) work and how they can be applied to predict the stocks given real stock market data. LSTM-RNNs are highly powerful machine-learning models that can be used for both regression and classification tasks. Their power lies in their unparalleled ability to enhance the long-term memory of AI models through their use of cell state in addition to hidden state, which is used in typical RNNs. In the first part of this paper, I will explain what I have learned about the theory and concepts behind LSTM-RNNs. We will explore the basic RNN architecture to understand the LSTM-RNN better. I will briefly cover why the LSTM unit was created, which was to deal with the vanishing/exploding gradient problem that was an inherent issue of vanilla RNNs. Then we will cover LSTM units and how they fix the vanishing/exploding gradient problem inherent within RNNs. We will also cover a learning algorithm associated with LSTM-RNNs, called Backpropagation Through Time (BPTT). In the second part of this report, I will discuss the LSTM-RNN model that I made using TensorFlow and Keras libraries and how I used it to make predictions of stock prices using real stock market data. Finally, I will discuss the results of my model.**

***Keywords*— *Recurrent Neural Network (RNN), Long-Short Term Memory (LSTM), Activation function, Sigmoid, Tanh, Forget Gate, Input Gate, Output gate, Cell State, Hidden State, Backpropagation Through Time (BPTT), gradient, TensorFlow, Keras, Sci-kit learn, Stocks***

## I. Introduction

This report covers what I have learned about LSTM-RNNs, the background required to thoroughly understand them, and how they can be applied to stock market data. The first three sections of this report (II, III, IV, V) will cover the theory and concepts of both RNNs and LSTM-RNNs. The final two sections will be about how (VI, VII) will be about the LSTM network model that I created, how I used it to predict stock prices, and the results that I attained.

The first topic I will cover is the Recurrent Neural Network (RNN), which is necessary to understand if one wants to know why LSTM networks were developed in the first place. RNNs are a type of neural network architecture used to detect patterns within sequential data [2]. Much of the data encountered in the world is sequential, therefore, RNNs have found many applications over the years, such as natural language processing (NLP), text generation, time-series analysis and forecasting (e.g. stock market data, weather), speech recognition, and image description generation. We will cover how RNNs differ from Feedforward Neural Networks, which stem from their use of feedback loops that create cycles that transmit information back into itself [2].

The second topic covered in this report is the problem inherent in vanilla RNNs, the vanishing/exploding gradient problem. The vanishing/exploding gradient problem exists because every previous step is considered in the calculation during backpropagation through time (BPTT) due to the chain rule engraved in the hidden state calculation of RNNs. The hidden state of an RNN is the representation of the network's memory at a given time step. It is a vector that encapsulates information from the previous time steps and is used to influence the output and the next hidden state. To better understand the vanishing/exploding gradient problem, it is helpful to understand what it means to unroll an RNN, which will be discussed later. Then it will be easier to understand the issue of drastic gradient change within vanilla RNNs.

The third topic covered is one of the most popular ways of dealing with the vanishing/exploding gradient problem, the LSTM cell. The LSTM cell is the fundamental unit of an LSTM-RNN, which is a network consisting of numerous LSTM cells. There are some similarities between RNNs and LSTM-RNNs, for instance, both use the same set of weights and biases at each time step. This weight sharing is what allows these models to handle data sequences of different lengths. However, where they differ is that LSTM units have some additional architecture that allows them to maintain both a cell state and a hidden state. The cell state is the long-term memory component that carries information across many time steps with minimal modification. The hidden state is similar to the hidden state of a RNN, it is the short-term memory component used for output, but the cell state directly influences it. LSTM cells use three different gates that allow them to maintain and update the cell state and hidden state of the cell (see Fig. 2). The forget gate decides what information to discard from the cell state. The input gate decides what new information to add to the cell state. The output gate decides what part of the cell state to output as the hidden state. In the section on the LSTM networks, I will briefly cover some of the math behind how an LSTM unit works.

The fourth topic is about making and training an LSTM-RNN model using libraries such as TensorFlow, Keras, and Scikit-learn. Training LSTM-RNNs involves many of the same

fundamental steps as training other neural networks, with some additional considerations due to their recurrent nature. The steps involved in training an LSTM-RNN include data preparation, network initialization, the forward pass, loss calculations, the backward pass, weight updates, and hyperparameter tuning. All of these steps are modified to work with sequential data. Each of these aspects of training an LSTM-RNN is discussed in section four.

Finally, I will cover the results I achieved using an LSTM-RNN to make predictions of stock prices using real stock market data. The dataset I have used to train my LSTM-RNN on and make predictions has about 5 years of Apple stock data. I have displayed the performance metrics of my model in a tabular format and given an analysis of how well I think the model did and where it might be improved.

## II. RECURRENT NEURAL NETWORKS

RNNs are specialized neural networks that are used for processing sequential data, such as time-series data $X_t = X_1, \dots, X_n$ with time step t ranging from 0 to n. RNNs are termed recurrent because they repeatedly execute the same task for each element in a sequence, with the output depending on the preceding computations [4]. This is why RNNs are said to have "memory" which captures information that has been calculated already. This is the major difference between RNNs and regular Feedforward networks. Feedforward Networks transmit information through the network without forming cycles, whereas RNNs form cycles using feedback loops that allow information to be fed back into the network. This allows them to extend the functionality of Feedforward Networks by considering previous inputs $X_{0:n-1}$ and not only the current input $X_n$ [1].

Fig. 1 [4] shows a high-level graph representation of an RNN. The left side of the diagram depicts a compact representation of an RNN, while the right side shows the RNN unrolled over time. Unrolling refers to a way of expressing the network for the entire sequence, representing each time step as a separate layer. This is useful for understanding how information flows through the network over time.

At each time step $t$, the input $x(t)$ is fed into the network. The hidden state $h(t)$ acts as the memory of the network. It captures information from the previous inputs and propagates it forward, which can easily be seen from the unrolled RNN on the right side of Fig. 1 [4]. The hidden state at time step $t$ is computed using the current input $x(t)$ and the hidden state from the previous time step $h(t-1)$:

$$h(t) = f(Ux(t) + Wh(t-1) + b), \qquad (1)$$

where $U$ is the weights matrix for input-to-hidden connections, $W$ is the weights matrix for hidden-to-hidden (recurrent) connections, and b is the bias term [4]. These weights are shared across all time steps, which helps in reducing the number of parameters and making the model efficient. The output $o(t)$ at each time step is derived from the hidden state. This output can be subjected to further non-linear transformations, especially if there are additional layers in the network. The loss $L$ is computed by comparing the output $o(t)$ (or its processed form) with the actual target $y(t)$.

Unrolling the RNN, as shown on the right side of the diagram, helps to visualize how the network handles sequences. Each layer corresponds to a different time step. At time $t-1$ the input is $x(t-1)$, the hidden state is $h(t-1) = f\big(Ux(t-1) + Wh(t-2)\big)$, the output is $o(t-1)$, and so on for the other time steps. The loss $L$ measures the difference between the predicted output $o(t)$ and the actual target $y(t)$. Internally, the loss function computes the predicted probabilities $\hat{y}$ from the outputs $o$ and compares them with the true labels $y$.

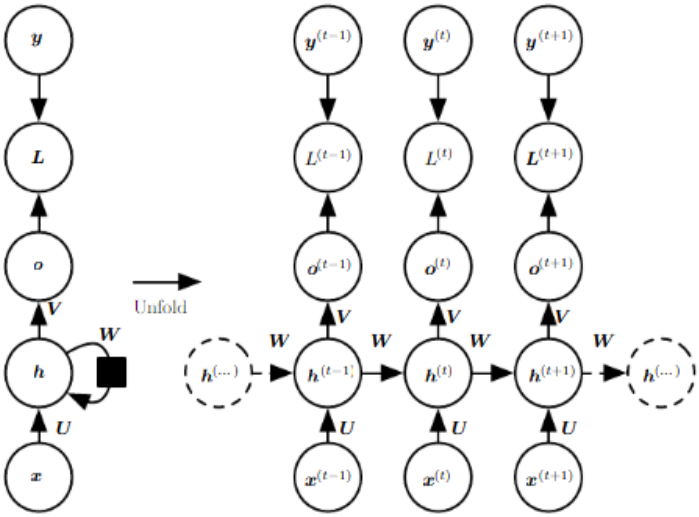


Figure 1 (adapted from [4]): The computational graph illustrating the training loss calculation for a recurrent neural network (RNN) that maps an input sequence $x$ to a corresponding output sequence $o$. The loss $L$ quantifies the discrepancy between each output $o$ and the corresponding target $y$. The RNN is defined by input-to-hidden connections parameterized by weight matrix $U$, hidden-to-hidden recurrent connections parameterized by weight matrix $W$, and hidden-to-output connections parameterized by weight matrix $V$. (Left) The RNN and its loss are represented with recurrent connections. (Right) The same RNN is shown as a time-unfolded computational graph, where each node corresponds to a specific time instance.

To effectively train RNNs and adjust the weights $U$, $W$, and $V$ to minimize the loss $L$, we use a specialized version of the backpropagation algorithm called Backpropagation Through Time (BPTT).

## III. BACKPROPAGATION THROUGH TIME (BPTT)

BPTT takes into account the temporal dependencies and the cyclic nature of RNNs by unrolling the network over time and computing gradients for each time step. This allows the network to learn from the sequential data and update its parameters accordingly, improving its performance on tasks involving temporal sequences. The steps of BPTT include unrolling the network, processing the input sequence in the forward pass, calculating the loss, the backward pass, and updating the gradients.

The RNN is unrolled for however many time steps there are in the sequence, where each time step is represented as a separate layer in a feedforward network. During the forward pass, the input sequence is processed through the unrolled network from the first to the last time step. When the input sequence $X_t$ is forward passed through the RNN, we compute

the hidden state $H_t$ and the output state $O_t$ one step at a time [1]. The loss function $L(O, Y)$ describes the difference between the outputs $O_t$ and the target values $Y_t$ [1]. The total loss $L$ is the sum of the individual loss terms $\ell t$ for each time step $t$:

$$L(O, Y) = \sum_{t=1}^{T} \mathrm{l}_t(O_t, Y_t), \qquad (2)\ [1].$$

The backward pass involves calculating and propagating the gradients of the loss backward through time from the last time step to the first. This is done using the chain rule, which multiplies the gradients of the current time step with those from the previous time steps. To compute the gradients of the loss $L$ we need to compute the partial derivatives of the loss $L$ w.r.t each of the weight matrices $U$ (input-to-hidden), $W$ (hidden-to-hidden), and $V$ (hidden-to-output) of an RNN using the chain rule. The partial derivative of the loss $L$ w.r.t. $V$ is computed as follows:

$$\frac{\partial L}{\partial V} = \sum_{t=1}^{T} \frac{\partial \mathrm{l}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial V}, \quad (2)\ [1].$$

Equation 2 can be simplified and written in the form of equation 3.

$$\frac{\partial L}{\partial V} = \sum_{t=1}^{T} \frac{\partial \mathrm{l}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot H_t, \qquad (3)\ [1]$$

Here, $\mathrm{l}_t$ is the loss at time step $t$, $O_t$ is the output at time step $t$, $\phi_o$ is the activation function for the output layer, and $H_t$ is the hidden state at time step $t$.

The partial derivative of the loss $L$ w.r.t.. $W$ is more complex due to the recurrent nature of the RNN. It is computed as follows.

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathrm{l}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot \frac{\partial H_t}{\partial \phi_o} \cdot \frac{\partial H_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial W}, \qquad (4)\ [1]$$

Simplifying equation 4 we get equation 5.

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathrm{l}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot V \cdot \frac{\partial H_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial W}, \qquad (5)\ [1]$$

Considering the dependency of $H_t$ on previous time steps we can substitute the last part from equation 5.

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathrm{l}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot V \sum_{k=1}^{t} \frac{\partial H_t}{\partial H_k} \cdot \frac{\partial H_k}{\partial W}, \quad (6)\ [1]$$

This can be expanded as shown in equation 7.

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathrm{l}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot V \sum_{k=1}^{t} W^{t-k} \cdot H_k, \ (7)\ [1]$$

In the equations 4, 5, 6, and 7: $\mathrm{l}_t$ is the loss at time step $t$, $O_t$ is the output at time step $t$, $\phi_o$ is the activation function for the output layer, $\phi_h$ activation function for the hidden layer, $H_t$ is the hidden state at time step $t$, $H_k$ is the hidden state at time step $k$, $[\sum_{t=1}^{T}]$ is the summation over all time steps, $[\sum_{k=1}^{t}]$ is the summation over all previous time steps up to $t$, and $W^{t-k}$ represents the propagation of the weights through $t-k$ time steps.

The partial derivative of the loss $L$ w.r.t $U$ is computed similarly to the partial derivative w.r.t. $W$.

$$\frac{\partial L}{\partial U} = \sum_{t=1}^{T} \frac{\partial \mathrm{l}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial H_t} \cdot \frac{\partial H_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial U}, \qquad (8)\ [1]$$

$$\frac{\partial L}{\partial U} = \sum_{t=1}^{T} \frac{\partial \mathrm{l}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot V \cdot \sum_{k=1}^{t} W^{t-k} \cdot X_k, \qquad (9)\ [1]$$

In equation 9, $X_k$ is the input vector at time step $k$.

The gradients computed during the backward pass are used to update the network parameters. These updates are performed using optimization algorithms like Gradient Descent or its variants. In practice, computing these gradients can lead to instability due to the vanishing and exploding gradient problems.

## IV. The Vanishing/Exploding Gradient Problem

The vanishing/exploding gradient problem led to gradient-based RNN methods requiring an extremely long amount of time to train and made RNNs produce inaccurate results. These problems occur due to the nature of gradient propagation through many layers (or time steps) in the network. As described in the equations for computing gradients w.r.t. weight matrices $U$, $W$, and $V$, the gradients involve products of many terms, which can lead to gradient instability.

When calculating the gradients using Backpropagation Through Time (BPTT), we see terms like $\frac{\partial H_t}{\partial H_k}$ appearing in the equations. This term introduces matrix multiplication over the sequence. If the values in the weight matrix are less than 1, the gradients can diminish exponentially as we go back in time, leading to the vanishing gradient problem. Specifically, the gradient $\frac{\partial H_t}{\partial H_{t-n}}$ decreases with each time step $n$, eventually becoming negligibly small [1]. This means that the contributions of earlier time steps to the current state are effectively lost, preventing the network from learning long-term dependencies.

Conversely, if the values in the weight matrix are greater than 1, the gradients can grow exponentially, this is called the exploding gradient problem. In this case, the gradient $\frac{\partial H_t}{\partial H_{t-n}}$ increases with each time step $n$, becoming excessively large [1]. This causes the weights to be updated too much, resulting in unstable training and large changes in the loss function.

These problems are critical because they disallow RNNs from learning from long sequences of data. To address the vanishing gradient problem, Long Short-Term Memory (LSTM) cells were introduced.

## V. LSTM-RNN

An LSTM-RNN is a special type of RNN architecture that was developed to combat the vanishing gradient error that vanilla RNNs suffer from. LSTM structures have changed as they have been developed, but the following is a description of the most common architectures [2]. An LSTM cell structure attempts to capture information in two states, the cell state and the hidden state. Within an LSTM cell, these states are controlled by three gates. The three gates are called the forget gate, the input gate, and the output gate. Each gate serves a unique purpose within an LSTM cell. Fig. 2 [2] shows the architecture of an LSTM cell.

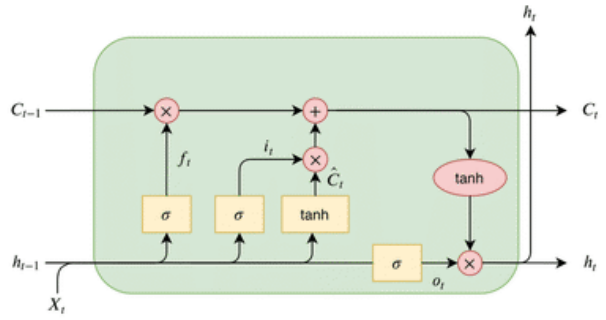


Figure 2 (adapted from [2]): LSTM cell architecture, $X_t$is the input time step, $h_t$is the output, $C_t$ is the cell state, $f_t$ is the forget gate, $i_t$ is the input gate, $o_t$ is the output gate, $\hat{C}_t$ is the internal cell state. Operations inside the light red circle are pointwise (an operation that is applied independently to each element of an vector, matrix, or tensor).

The cell state ($C_t$) carries information across many time steps and acts as the long-term memory (the cell state can be seen running horizontally at the top of the LSTM cell in Fig. 2). The cell state is updated at every time step, but it tends to retain information longer due to its linear transformations. The cell state is modified by the forget and input gates and combines information from these two gates to determine what information is discarded and what new information is added.

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \widetilde{C_t}, \quad (10)\ [1]$$

In equation 10, $C_t$ is the cell state, $f_t$ is the forget gate, $i_t$ is the input gate, $C_{t-1}$ is the cell state from the previous time step, and $\widetilde{C_t}$ is the candidate cell state. Candidate cell state is used to determine the internal cell state of the LSTM cell.

$$\widetilde{C_t} = \tanh(x_t U^g + h_{t-1} W^g), \quad (11)\ [1]$$

In equations 15, $\widetilde{C_t}$ is the candidate cell state, $U^g$ and $W^g$ are the weight matrices for generating the candidate cell state, $x_t$ is the input at time step t, and $h_{t-1}$ is the hidden state from the previous time step.

The hidden state ($h_t$) is used at each time step to produce the output and is the short-term memory (the hidden state is output vertically from the LSTM cell in Fig. 2).

$$h_t = o_t \cdot \tanh(C_t), \quad (12)\ [1]$$

In equation 11, $h_t$ ,$o_t$ is the output gate, and $C_t$ is the current cell state.

The computations for the LSTM cell gates are shown in the following equations (12, 13, and 14).

$$f_t = \sigma(x_t U^f + h_{t-1} W^f), \quad (13)\ [1]$$

$$i_t = \sigma(x_t U^i + h_{t-1} W^i), \quad (14)\ [1]$$

$$o_t = \sigma(x_t U^o + h_{t-1} W^o), \quad (15)\ [1]$$

Here, σ is the sigmoid activation function, $x_t$ is the input at time step t, $h_{t-1}$ is the hidden state from the previous time step, $U^f$ and $W^f$ are the weight matrices for the forget, $U^i$ and $W^i$ are the weight matrices for the input gate, $U^o$ and $W^o$ are the weight matrices for the output gate.

The forget gate determines how much of the previous cell state $C_{t-1}$ to retain when we update the cell state. The input gate determines how much new information to add to the cell state when we update the cell state. The output gate determines what to output based on the updated cell state $C_t$. As can be seen from the equations that govern each of these gates, weights are connected to each gate. These weights are used along with gradient-based optimization to train the LSTM cell.

The LSTM cells described above are chained together allowing LSTM-RNNs to retain information from past time steps and make time-series predictions. Using the LSTM cell architecture, the network can solve the vanishing gradient problem.

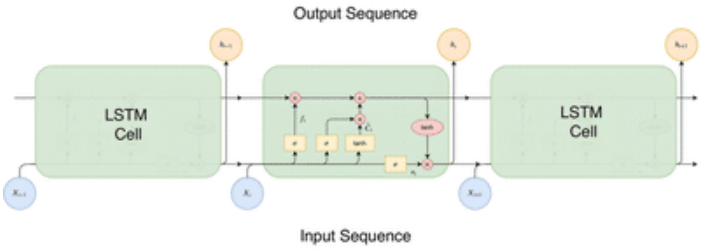


Figure 3 (adapted from [2]): LSTM cells chained together, with input sequence and output sequences shown.

## VI. Creating and Training my LSTM-RNN Model

I used Keras and TensorFlow to build a multi-layer LSTM-RNN model. The model is built using the Sequential class from the tensorflow.keras.models module. I have experimented with my LSTM model architecture, but have landed on the following model as it has been my most successful attempt. The architecture of the model is as follows: first, there is a layer of 50 LSTM units (with return sequences set to true), then there is a dropout layer for regularization (0.3 dropout rate), then there is another LSTM layer with 50 units(not returning sequences). Next, there is a second dropout layer (0.2 dropout rate), after that, there is a dense layer with 25 units. Finally, there is a dense layer with 1 unit. The model is compiled with the Adam optimizer, which automatically sets the learning rate at 0.001, and mean squared error (MSE) loss function. Early stopping is used to prevent overfitting, monitor the validation loss with a patience of 5 epochs, and ensure that the model weights are restored to the state corresponding to the epoch with the best validation loss.

To train my LSTM network I had to begin by preparing the data, which involves loading the dataset, cleaning it, and normalizing it. I loaded the dataset from a URL to the dataset that I hosted on my Git Hub and read the CSV file into a Pandas DataFrame. Cleaning the data involved converting the date column to a datetime format, which is standardized way to represent date and time information in Python. I was then able to ensure that the data was sorted in chronological order using the "sort_values" method from the Pandas library. I set the date column as the DataFrame index using "df.set_index", and removed symbols, such as "$", commas, and semicolons from the price columns. Finally, I converted the columns to floats. To normalize the data I used MinMaxScaler from sklearn, to scale the features to a range of [0, 1], which helps improve the convergence and stability of the neural network training process

and makes the data have a mean of 1 and a standard deviation of zero.

When preparing data for LSTM-RNNs, it is important to preserve the temporal order of the data. One way to ensure that the split of the dataset into training and validation sets preserves the temporal order is to use rolling-window validation. Rolling-window validation is a method that uses a window (a subset of data points from the dataset) of a fixed-sized length that contains a fixed number of data points. Since the data moves forward sequentially in time, the window is simply shifted forward in time. In my model, the dataset is divided into multiple overlapping training and validation sets to simulate how the model performs on unseen data over time. Specifically, I use a fixed window size of 200 data points for training and a validation window size of 10 data points. For each iteration, the model is trained on the data within the training window, and its performance is validated on the subsequent validation window.

Another method for splitting time-series data into training and validation sets is time-series validation. In time-series validation, we start with a window of some length, and then continually add a data point to the set until we have gone through the entire dataset. The last data point of the sequence is the actual value we are trying to predict. I decided to use rolling-window validation to split my dataset due to this method requiring less computation. I tried time-series validation but realized that it would take a long time to run on my computer. Therefore, I decided to use only rolling-window cross-validation.

As stated earlier, I used early stopping to halt training when the validation loss stops improving, with a patience of 5 epochs to ensure the training process does not prematurely terminate. This technique helps prevent overfitting and ensures that the model generalizes well to new data. The rolling window approach, combined with the validation window, provides an effective framework for evaluating the model's performance over time.

After training, the model's performance is evaluated using several metrics, including mean squared error (MSE), root mean squared error (RMSE), mean absolute error (MAE), R-squared ($R^2$), and explained variance score. These metrics provide a comprehensive assessment of the model's ability to predict stock prices. Finally, the trained model is used to make predictions on future stock prices, averaging the predictions from multiple validation windows to enhance reliability. The results discussed in the next section indicate that the LSTM-RNN model effectively captures the patterns in the temporal dependencies in stock prices, making it a useful tool for financial forecasting.

## VII. RESULTS FROM MY LSTM-RNN MODEL

The model that performed the best had the architecture described in the previous section and had the hyperparameters of batch size of 10 and 10 epochs. It predicted that the close stock price would be $217.29 on 07/26/2024. The actual close price for Apple stock on 07/26/2024 was $217.96. Table 1 shows metrics calculated on the entire training and validation sets after the model has been trained. Table 2 shows the metrics that I used on the training sets per split to determine the performance of my model. Table 3 shows the metrics that I used on the validation sets per split. Both of these tables show aggregated metrics across all training and validation windows.

| Metric | Value |
|---|---|
| Training MSE | 0.000228 |
| Validation MSE | 0.000312 |
| Training RMSE | 0.015099 |
| Validation RMSE | 0.017653 |
| Training MAE | 0.011256 |
| Validation MAE | 0.015108 |
| Training $R^2$ | 0.959285 |
| Validation $R^2$ | 0.657239 |
| Training Explained Variance Score | 0.962397 |
| Validation Explained Variance Score | 0.663785 |

Table 1:Metrics calculated on the entire training and validation sets

| | Train MSE | Train RSME | Train MAE | Train $R^2$ | Train EVS |
|---|---|---|---|---|---|
| count | 988.000 | 988.000 | 988.000 | 988.000 | 988.000 |
| mean | 0.00025 | 0.01551 | 0.01207 | 0.94086 | 0.94924 |
| std | 0.00010 | 0.00305 | 0.00247 | 0.02939 | 0.02395 |
| min | 0.00010 | 0.01000 | 0.00777 | 0.80233 | 0.88142 |
| 25% | 0.00017 | 0.01288 | 0.00996 | 0.92241 | 0.93281 |
| 50% | 0.00024 | 0.01545 | 0.01201 | 0.94263 | 0.94942 |
| 75% | 0.00031 | 0.01770 | 0.01383 | 0.96583 | 0.97119 |
| max | 0.00092 | 0.03033 | 0.02280 | 0.98576 | 0.98629 |

Table 2: Train Metrics per Split

| | Val MSE | Val RSME | Val MAE | Val $R^2$ | Val EVS |
|---|---|---|---|---|---|
| count | 988.000 | 988.000 | 988.000 | 988.000 | 988.000 |
| mean | 0.00042 | 0.01821 | 0.01514 | -0.7441 | 0.02609 |
| std | 0.00069 | 0.00939 | 0.00840 | 2.10490 | 0.84053 |
| min | 0.00002 | 0.00422 | 0.00308 | -22.363 | -7.4521 |
| 25% | 0.00016 | 0.01253 | 0.01018 | -0.9936 | -0.2316 |
| 50% | 0.00027 | 0.01631 | 0.01337 | -0.1744 | 0.22226 |
| 75% | 0.00047 | 0.02164 | 0.01805 | 0.31715 | 0.55399 |
| max | 0.00995 | 0.09978 | 0.08710 | 0.87861 | 0.97429 |

Table 3: Validation Metrics per Split

I also made some graphs that show how the model's performance evolves during training. Fig. 4 and 5 plot the MSE loss over every epoch and the RSME loss over every epoch respectively.

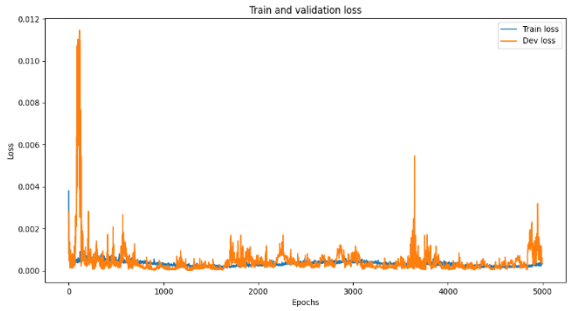


Figure 4: shows the training (blue) and validation (orange) MSE losses over all epochs.

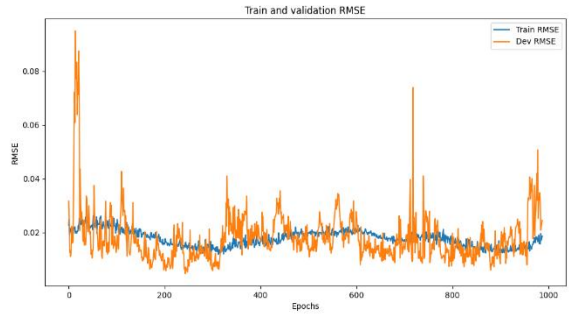


Figure 5: shows the training (blue) and validation (orange) RSME losses over all epochs.

These graphs both show that at the beginning of training, there is a high validation loss (sometimes termed dev loss for development loss) because the model is just starting to learn and hasn't captured any of the patterns in the data yet. As training progresses, both the training and validation losses tend to decrease. This indicates that the model is learning and improving its performance on both the training and validation datasets. The validation loss might fluctuate more than the training loss. This can be due to numerous reasons, such as the inherent variability in the validation data or overfitting if the model starts to memorize the training data rather than generalizing well. However, it is noticeable that while the training loss is significantly more stable in both graphs, the training and validation loss both follow the same trends as the learning progresses, just with the validation loss having more variability. Ideally, the training and validation losses should converge to similar values because a significant gap between them might indicate overfitting (training loss is much lower than validation loss) or underfitting (both losses are high).

## VIII. CONCLUSION

Based on the performance of my model, there is still room for improvement. I could continue to tune the hyperparameters of the network, such as experimenting with batch size, learning rate, and dropout rate to try to find a combination of hyperparameters that produces even better results. I could increase the training time with more epochs, even though when I tried this it did not improve the performance over 10 epochs. I could try to create additional features from the data, given that my data set did not have many features. I could also try to increase the complexity of my model's architecture by adding another LSTM layer or using a different form of cross-validation, such as time-series cross-validation. In conclusion, I have learned much about sequential data processing using LSTM-RNN architecture, and plan to continue to further improve my model using some of the methods described above or any other methods that I come across as I continue to learn this fascinating topic.

## REFERENCES


[1] R. M. Schmidt, "Recurrent Neural Networks (RNNs): A gentle Introduction and Overview," arXiv preprint arXiv:1912.05911, 2019. [Online]. Available: https://arxiv.org/abs/1912.05911

[2] T. M. Ingolfsson, "Insights into LSTM architecture: Finding the total number of multiply and accumulate operations in a LSTM layer," Thorir Mar Ingolfsson Blog, Nov. 10, 2021. [Online]. Available: https://thorirmar.com/post/insight_into_lstm/

[3] J. Nabi, "Recurrent Neural Networks (RNNs): Implementing an RNN from scratch in Python," *Towards Data Science*, Jul. 11, 2019. [Online]. Available: https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org