Recurrent Neural Networks on Time Series Prediction

Matthew Chimitt
University of Texas at Dallas
2200 Waterview Parkway, Richardson,
Texas 75080, United States
mmc200005@utdallas.edu

***Abstract*—Recurrent Neural Networks are a form of Artificial Neural Network that reuse outputs as inputs to interpret and train on temporal or time series data. These Recurrent Neural Networks work similarly to Artificial Neural Networks in that they have a forward pass and a backward pass, known as an epoch. Recurrent Neural Networks, however, face the vanishing gradient problem, which can lead to poor performance as gradients in backward propagation become too small to properly optimize the model. To combat against this, we can use a Long Short Term Memory Network, which uses various gate mechanisms to control the gradients during backward propagation. This paper will dive into understanding how the Long Short Term Memory Network is able to make predictions on time series data. In addition, an implementation of the LSTM in Python will explore how the different hyper parameters affect the performance of the model.**

Keywords—Long Short-Term Memory, Recurrent Neural Network, Neural Network, Time Series, Stock Prediction

I. Introduction

This report provides research into Recurrent Neural Networks and their applications on time series datasets. A Recurrent Neural Network, or RNN, is a form of an Artificial Neural Network that makes use of previous inputs in addition to the current input to influence the output [1]. RNNs are applied to numerous problem domains, such as language translation, natural language processing, prediction of future events, and more [1]. In this report, I will be using an RNN to predict stock prices for Dell Technologies Inc., as stock prices are influenced by previous stock prices, leading to a temporal problem domain.

II. Background Work

To fully understand what a Recurrent Neural Network is and how it functions, it is crucial to understand Artificial Neural Networks and how they operate, including the idea behind forward and backward propagation. In addition to this, it is important to understand the idea of gradient descent found in an Artificial Neural Network's backward propagation functionality.

A. Artificial Neural Networks

Artificial Neural Networks, or ANNs, are a deep learning technique that operates by mimicking the interconnected neurons found within the human brain. The neurons are represented by a unit called a perceptron, which takes multiple input values which are multiplied by a corresponding weight value and sums them up. Following this, the sum is passed into an activation function in order to introduce non-linearity. The output from the activation function is the output of the perceptron. An ANN is essentially a directed graph of these perceptrons, where output from one perceptron becomes the input of the next until reaching the output of the network [2]. During the training process of an ANN, the model goes through forward propagation, where input data moves through the network, producing a prediction. Following this, an error is calculated based on the predicted value and the actual value, which then triggers backward propagation. This phase involves adjusting the weights of the perceptrons through a process known as gradient descent [3]. This process fine tunes the model, and over various iterations of forward and backward propagation, a model is able to learn to effectively make predictions. Because RNNs are a form of artificial neural networks, it is very important to understand how ANNs work as RNNs also contain a forward and backward pass to optimize the weights found within the network.

B. Gradient Descent

Gradient descent occurs during the backward propagation phase within an ANN and allows for fine tuning of the weights within each perceptron of the network. Gradient descent works to minimize the error of the model and does so by iteratively adjusting the weights by calculating the gradient of the error with respect to each weight [3]. When plotting the error vs. the weights, the graph is a convex function [4]. When performing gradient descent, we are ultimately attempting to find the minima of the convex function, as that will result in the best weights for our model since the minimum will yield the smallest error given the weights [4]. Thus, during gradient descent, we start at a point on the curve, and gradually travel down the curve until finding a local minimum [5]. When looking at weight optimization, we can use the following formula to get the new weights for our model:

$$w_{new} = w_{old} - \frac{\partial L}{\partial w_{old}} \cdot \alpha \tag{1}$$

Understanding how gradient descent works within backward propagation is crucial in understanding the implementation of an RNN as a similar process is used in the RNN backward propagation.

III. Theoretical and Conceptual Study

A. What is a Recurrent Neural Network

Recurrent Neural Networks, or RNNs, are a form of ANN used to model sequential or temporal data [6]. Much like ANNs, RNNs use forward propagation and backward propagation to train and optimize their weights and biases. However, what makes RNNs special is that the inputs are

not independent of each other. For example, a common application of an RNN is predicting the next word of a phrase, which required knowledge of the previous words to properly make a prediction [6]. To do this, the RNN uses what is called a hidden state, which allows the network to in a sense "remember" specific pieces of information from previous inputs or sequences [6].

B. Inputs of Recurrent Neural Networks

The input of an RNN is very important in understanding how they operate. Traditionally, networks take in an input X and output y. In the case of a RNN, input X holds previous outputs while output y is the current output. For example, when looking at stocks, if predicting the closing price of day i, then the input X will hold the closing price of day i-1, i-2, …, i-n, where n is the specified number of outputs that you want to hold in memory. Each previous output in X is called a timestep. The shape of our input X consists of (samples, timesteps, features), where samples is the number of data points or inputs that we have in X, timesteps are the before mentioned previous outputs we want to remember, and features are the features we want to use in our model. In the case of my model, I only use 1 feature, the adjusted closing price of the Dell Technologies Inc. stocks each day, while providing the option of adjusting the number of timesteps in my preprocessing function.

C. Functionality of Recurrent Neural Networks

RNNs have various layers, much like an ANN. These layers include the input layer, where inputs are passed in, the hidden layers, which are layers of perceptrons that process the data by applying non-linear activation functions to learn complex tasks, and the output layer, which provides the network with the result. RNNs work much like an ANN, however, during the forward propagation process, an ANN will analyze the current input, and has no comprehension of temporal order [6]. RNNs however, will utilize a loop where before making any sort of judgement, the current input is evaluated in addition to the output from each of the past inputs, or timesteps as we saw when looking at the inputs of the RNN [6]. Each timestep is passed into the network, and the outputs from previous timesteps as well as the data from the current timestep is utilized to make the prediction. The diagram in Fig. 1 shows the architecture of a basic RNN. The left side of Fig. 1 gives us a simplified view of an RNN. The node with the "x" in it is our input layer, where we feed the network our data. The data is then moved to the hidden layer labeled "h". The data that is then processed and sent to the output layer, labeled "o", but is also returned to the hidden layer to be used as an input. The variables U, W, and V in the diagram refer to the weights of the model. U is the input to hidden layer weight, W is the hidden layer to hidden layer weight, and V is the hidden layer to output layer weight [7]. These weights are shared across time, meaning the weights are the same at each timestep [7]. The figure uses an idea of "unfolding" the network, sometimes referred to as "unrolling". This idea allows us to view the network in a complete sequence [7]. The unrolled version of the network shows the same layers as the left side; however, we can see exactly what is happening at each step and can get an idea

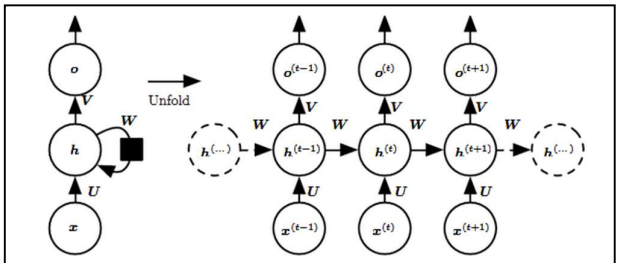

Fig. 1. Recurrent Neural Network Architecture.
Source: Adapted from [7]

of what goes into the hidden layer in order to obtain an output. For example, when looking at node $h^{(t)}$, or the hidden state at timestep t, we can see that it is receiving inputs from node $x^{(t)}$, which is from the input layer, but it is also receiving input from node $h^{(t-1)}$, which is the previous hidden state. These inputs are then used to provide our output, $o^{(t)}$, but in addition, $h^{(t)}$ is then used to as an input for $h^{(t+1)}$. This visualization allows us to get a good understanding of how the forward propagation works in an RNN, and we can use the ideas in the figure to establish the equations needed to implement this. For implementation, we will let W be our hidden layer to hidden layer weights, U be our input to hidden layer weights, V be our hidden layer to output weights, b and c be our bias terms, $h^{(t)}$ be our hidden state at timestep t, $o^{(t)}$ be our output at timestep t, $x^{(t)}$ be our input at timestep t, and $\hat{y}^{(t)}$ be our predicted output. Given these variables, we can make the calculations needed for our forward propagation. Thus, the following calculations can be used [7]:

$$h^{(t)} = tanh\,(b + W h^{(t-1)} + U x^{(t)}) \tag{2}$$

$$o^{(t)} = c + V h^{(t)} \tag{3}$$

$$\hat{y}^{(t)} = o^{(t)} \tag{4}$$

Equation (2) calculates our hidden state at the current timestep, given our inputs $x^{(t)}$ and the previous hidden state $h^{(t-1)}$, where we use tanh as our activation function. Then in (3) we use the calculated hidden state to determine our output at the timestep. In (4), we set our predicted output to be our output. In a classification task, we would use an activation function such as the softmax function to properly predict our data. However, in the context of stock prediction, which is the focus of my study and implementation, no activation function is required, as stock prediction is a regression task. This set of equations is our forward propagation step for our RNN. We can use the predicted output $\hat{y}$ to compute our error, which is our predicted output, $\hat{y}$, minus our actual output, y. We can use our error when looking at the loss function during backward propagation, which follows immediately after our forward propagation process in order to optimize our weights and biases. Backward propagation in an RNN works similarly to an ANN, and our weights optimization equation is the same as in (1). Thus, in backpropagation we are tasked with determining our gradients. To do this, we need to find the gradient, ∇, of each node as we iterate backward through time in our network, starting at the end of the sequence, which is a process known as backward propagation through time, or BPTT [7].

$$(\nabla_o^{(t)} L)_i = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} \quad (5)$$

$$\nabla_h^{(\tau)} L = V^\top \nabla_o^{(\tau)} L \quad (6)$$

$$\nabla_h^{(t)} L = W^\top diag(1 - (h^{(t+1)})^2)(\nabla_h^{(t+1)} L) + V^\top (\nabla_o^{(t)} L) \quad (7)$$

Equation (5) refers to the gradient with respect to the output node while (6) and (7) refer to the gradient with respect to the hidden state. Equation (6) is the gradient when we are beginning from the end of the sequence, which we denote as timestep τ, while (7) refers to the gradient of the previous timesteps as we iterate backwards in time, where timestep t is less than τ [7]. These equations provide us with the gradients of the nodes, or perceptrons, of the network, and we can use them to find the gradients of our weights, which are calculated by applying the chain rule . This is seen in (8), (9), and (10), as the gradients for the weights V, W and U are calculated respectively using the equations from (5), (6), and (7). These equations use a summation to take the sum at each timestep, which allows for the backward propagation to flow in the reverse direction, iterating through each timestep. Once we have obtained these gradients, we can use the formula in (1) to update each of the weights. Similarly, we can apply the same idea to our biases, b and c, whose gradients are calculated in (11) and (12) respectively.

$$\nabla_V L = \sum_t (\nabla_o^{(t)} L)\, h^{(t)\top} \quad (8)$$

$$\nabla_W L = \sum_t diag(1 - (h^{(t)})^2)(\nabla_h^{(t)} L)\, h^{(t-1)\top} \quad (9)$$

$$\nabla_U L = \sum_t diag(1 - (h^{(t)})^2)(\nabla_h^{(t)} L)\, x^{(t)\top} \quad (10)$$

$$\nabla_b L = \sum_t diag(1 - (h^{(t)})^2)\, \nabla_h^{(t)} L \quad (11)$$

$$\nabla_c L = \sum_t \nabla_o^{(t)} L \quad (12)$$

Following the optimization of the weights and biases, we repeat the forward and backward process in our training stage, ultimately creating our optimized RNN model.

D. Vanishing and Exploding Gradient Problems

While Recurrent Neural Networks do a great job allowing for sequential data to be processed to create a prediction, they suffer from a problem known as the vanishing gradient.

The usage of nodes that "loop back on themselves" to allow for utilization of previous timesteps comes with a problem for RNNs, a problem known as the vanishing gradient [8]. In the vanishing gradient problem, gradients get "weaker as they traverse through layers", which can lead to performance issues such as slowdowns or halts in learning [8]. Within Recurrent Neural Networks, this problem arises because of the loop at every timestep, because during backpropagation, we compute the gradients by applying the chain rule across all timesteps iteratively and in reverse. Gradients can become very small due to the "repeated multiplication" in each timesteps because of the application of the chain rule [8]. The closer a gradient is to zero, the less it will change the weights. Looking at (1) allows us to understand this, as the gradient multiplied by the learning rate is subtracted from the old weights to obtain the new weights. Thus, if the gradient is close to zero, the new weights will be virtually unchanged from the old weights, possibly leading to no improvement to the model performance. On the opposite end of this is another problem faced by RNNs known as the exploding gradient problem. This problem is similar to the vanishing gradient problem, however, rather than weights becoming too small, the weights become too bit as a result of the repetitive multiplication from the chain rule. While we can't completely solve these two problems, there are various methods that serve to minimize the problems. To minimize the exploding gradient problem in my model, I used a technique known as gradient clipping, which consists of limiting the gradients to a particular minimum or maximum value, which in my case I limited the gradients to be in between -1 and 1. To help combat against the vanishing gradient problem, I decided to implement a type of Recurrent Neural Network known as the Long Short-Term Memory network, or LSTM. This type of RNN "leverages gating mechanisms to control the flow of information and gradients" [8]. This control allows the LSTM to decide what information it wants to remember, what information it wants to forget, and what information it should output. This ultimately ensures that "gradients stay relevant," as the LSTM will let go of data that it doesn't need, while keeping the important data [8].

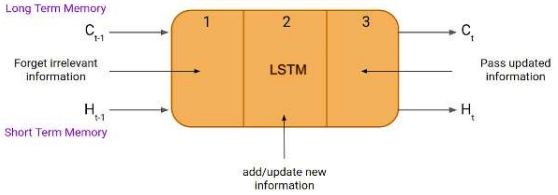

Fig. 2. Long Short-Term Memory Unit Architecture.
Source: Adapted from [9]

E. Long Short-Term Memory Network

As previously mentioned, the Long Short-Term Memory network, or LSTM, is a form of Recurrent Neural Network that helps prevent the vanishing gradient problem. Because it is a form of RNN, the LSTM works in the same way when regarding the output looping back to become an input for each timestep. The difference, however, comes when looking at the architecture of an LSTM unit. There are three parts that make up an LSTM unit, the input gate, the forget gate, and the output gate. In addition to these gates, the LSTM unit has a hidden state, much like in the simple RNN, but the LSTM also has a cell state for the current and previous timesteps [9]. The hidden state can be considered the short-term memory, while the cell state can be considered the long-term memory, which can be seen in Fig. 2, which shows the architecture of an LSTM cell [9]. The three sections of the cell in Fig. 2 labeled "1", "2", and "3" are referring to the forget gate, input gate, and output gate respectively. Also in Fig.2 are H_{t-1} and H_t , which refer to the previous and current hidden states, and C_{t-1} and C_t , which refer to the previous and current cell states at timestep t. During forward propagation of the LSTM, the input gate, forget gate, output gate, hidden state, cell state,

and output are all calculated in order to create a prediction. Following this, backward propagation adjusts the weights and biases found at each gate to optimize and improve the model's performance, much like in a simple RNN.

F. Forward Propagation

To forward propagate in an LSTM, you must calculate the value of all the gates, which are then used to determine the new hidden state, new cell state, and predicted value. When making calculations for each of the gates, we have weights and biases. In each gate equation, the hidden state at the previous timestep and the input at the current timestep are concatenated together and the dot product is taken of the concatenation and the weight, followed by the addition of the bias term [10]. This is then fed into a sigmoid activation function, where we get a value between zero and one. This value is used to ultimately determine whether information at the gate is relevant, or if we should drop it entirely. If the output of the sigmoid function is zero, then this means that nothing is passed through the gate, and we "forget" the data, but if the output is one, then "everything would be passed" through the gate [10]. The first gate that is referenced in an LSTM is the forget gate. This gate decides whether information is "relevant to context" [10]. For example, if we are using an LSTM to predict what word comes next in a sentence, context becomes very important in making an accurate prediction. If the current sentence or phrase is discussing food, as an example, and then switches its focus to a person later, then the LSTM's forget gate will forget the context related to food and focus on the context relating to the person. The equation for the forget gate is found in (13). Next we look at the input gate, which decides which information should be stored in the cell state, or c^t [10]. Here we introduce the idea of candidate values, denoted by $\tilde{c}$. Candidate values are values that can be added to the cell state, c^t. The equation for the input gate is found in (14), and the candidate values in (16). The final gate is the output gate, which we use to decide what we want to send as the output data [10]. The equation for the output gate is found in (15).

$$f^{(t)} = \sigma\left(W_f \cdot [h^{(t-1)}, x^{(t)}] \right) + b_f \tag{13}$$

$$i^{(t)} = \sigma\left(W_i \cdot [h^{(t-1)}, x^{(t)}] \right) + b_i \tag{14}$$

$$o^{(t)} = \sigma\left(W_o \cdot [h^{(t-1)}, x^{(t)}] \right) + b_o \tag{15}$$

$$\tilde{c}^{(t)} = \tanh(W_C \cdot [h^{(t-1)}, x^{(t)}] + b_C) \tag{16}$$

After calculating all of the gates, we can use them to find our new cell state, new hidden state, and predicted output. The new cell state is calculated in (17) and is done by multiplying the forget gate by the previous cell state, to determine whether we want to keep the data from the previous cell state, or if we want to forget it [10]. The product is then added to the product of the input gate and the candidate values, to determine what we want to add to the cell state [10]. We then use the new cell state in the calculation of our hidden state, as seen in (18). Here we are multiplying our output gate by the cell state after it undergoes the tanh activation function, which is done to "push values between -1 and 1" [10]. Finally, we can get our predicted output, which is calculated by taking the dot product of our output weight W_y and our hidden state, as seen in (19).

$$C^{(t)} = f^{(t)} * C^{(t-1)} + i^{(t)} * \tilde{c}^{(t)} \tag{17}$$

$$h^{(t)} = o^{(t)} * \tanh(C^{(t)}) \tag{18}$$

$$\hat{y}^{(t)} = W_y \cdot h^{(t)} \tag{19}$$

After we made our prediction, we successfully completed our forward pass. The error can be calculated by subtracting the predicted output by the actual output, much like the ANN and simple RNN. With the error computed, the LSTM moves on to its backward propagation phase.

G. Backward Propagation

The process of backward propagation in LSTMs is very similar to the process in the simple RNN. We start from the output at the last timestep and apply the chain rule iteratively on each timestep in the reverse direction [11]. For the LSTM specifically, we begin by finding the partial derivative of the output y with respect to the error L, as seen in (20). This is then used to find the partial derivative of the hidden state and the cell state, which is seen in (21) and (22) [11].

$$\frac{\partial L}{\partial y^{(t)}} = \hat{y}^{(t)} - y^{(t)} \tag{20}$$

$$\frac{\partial L}{\partial h^{(t)}} = W_y^{\top} * \frac{\partial L}{\partial y^{(t)}} + dh_{next} \tag{21}$$

$$\frac{\partial L}{\partial C^{(t)}} = dc_{next} + \frac{\partial L}{\partial h^{(t)}} * o^{(t)} * (1 - \tanh^2(C^{(t)})) \tag{22}$$

After finding the partial derivatives of the cell state and the hidden state, we can continue down the chain and find the partial derivatives with respect to the error of each of the gates and the candidate values. The candidate value partial derivative is displayed in (23), while the input, forget, and output gates partial derivatives are calculated in (24), (25), and (26) respectively [11].

$$\frac{\partial L}{\partial \tilde{c}^{(t)}} = \frac{\partial L}{\partial C^{(t)}} * i^{(t)} * (1 - (\tilde{c}^{(t)})^2) \tag{23}$$

$$\frac{\partial L}{\partial i^{(t)}} = \frac{\partial L}{\partial C^{(t)}} * \tilde{c}^{(t)} * i^{(t)} * (1 - i^{(t)}) \tag{24}$$

$$\frac{\partial L}{\partial f^{(t)}} = \frac{\partial L}{\partial C^{(t)}} * \tilde{c}^{(t)} * f^{(t)} * (1 - f^{(t)}) \tag{25}$$

$$\frac{\partial L}{\partial o^{(t)}} = \frac{\partial L}{\partial h^{(t)}} * \tanh(C^{(t)}) * o^{(t)} * (1 - o^{(t)}) \tag{26}$$

Following this, we can once again continue through the network, using the partial derivatives of the gates and candidate value to get the gradients of the weights, as seen in (27) through (31) [11].

$$\frac{\partial L}{\partial W_C} = \frac{\partial L}{\partial \tilde{c}^{(t)}} * [h^{(t-1)}, x^{(t)}]^{\top} \tag{27}$$

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial i^{(t)}} * [h^{(t-1)}, x^{(t)}]^T \quad (28)$$

$$\frac{\partial L}{\partial W_f} = \frac{\partial L}{\partial f^{(t)}} * [h^{(t-1)}, x^{(t)}]^T \quad (29)$$

$$\frac{\partial L}{\partial W_o} = \frac{\partial L}{\partial o^{(t)}} * [h^{(t-1)}, x^{(t)}]^T \quad (30)$$

$$\frac{\partial L}{\partial W_y} = \frac{\partial L}{\partial y^{(t)}} * h^{(t)^T} \quad (31)$$

Equations (32) and (33) refer to the derivative of the next hidden and cell states for the next iteration, which are used in the calculations of the gradients of the gates [11].

$$dh_{next} = W_C{}^T * \frac{\partial L}{\partial \tilde{c}^{(t)}} + W_i{}^T * \frac{\partial L}{\partial i^{(t)}} + W_f{}^T * \frac{\partial L}{\partial f^{(t)}} + W_o{}^T * \frac{\partial L}{\partial o^{(t)}} \quad (32)$$

$$dc_{next} = \frac{\partial L}{\partial c^{(t)}} * f^{(t)} \quad (33)$$

We can apply the same gradient descent weight formula from (1) to each of the weights to optimize our weights. For optimizing the biases, we can sum all the gradients in the corresponding gate gradients in order to get the gradient of the bias [11]. For example, for the forget gate bias, the summation of (25) will yield the partial derivative of the forget gate bias with respect to the loss. The weights equation in (1) applies the same to the biases as it does to the weights. This process makes up the back propagation for the LSTM, and combined with forward propagation, the LSTM can be trained to make efficient time series predictions.

IV. Results and Analysis

To see how an RNN can be used to predict time series data, I created a Long Short-Term Memory Network in Python, and fed in data from Dell Technologies Inc. stock data up until January 1, 2023. My LSTM model takes in hyperparameters that consist of the number of epochs, number of hidden units, number of timesteps, and the learning rate.

A. Timestep

I set up my data by allowing for a variable number of timesteps, so that I could determine whether the number of timesteps affects performance in any way. My original prediction was that a larger number of timesteps would yield a better performing model, however, this was not the case. I tested this by holding all of the hyperparameters constant, while changing the number of timesteps. Table I shows the training and testing mean squared error that resulted as a I slowly decreased the number of timesteps. During this test, I immediately noticed a difference in the duration of the training phase for each model. The models with the higher number of timesteps took noticeably longer at each epoch than those with a lower number of timesteps. This makes sense, as more timesteps means longer forward propagation and longer backward propagation, since the model processes data sequentially. The next thing that I noticed was that as the timesteps decreased, the training MSE got increasingly better. This goes against my hypothesis, as I originally believed that a higher number of timesteps would yields a better model, however, this proves that idea incorrect. Looking at Table I we can see that the training MSE is good for each of the models, however, the testing MSE is incredibly high when the timesteps are 25. This gives the idea that too many timesteps can lead to overfitting, which occurs when a model is able to efficiently predict, or "memorize", its training data, but underperforms on the testing data. Thus, because smaller timesteps not only yield faster training times, but also prevent overfitting, we can conclude that for the best predictions we should select a small timestep. For the remainder of my experiments, I will select a timestep of 3.

TABLE I. Results of the Timestep Test

Hyper Parameters				Mean Squared Error (MSE)	
Epochs	*Hidden Units*	*Timesteps*	*Learn Rate*	*Training*	*Testing*
50	50	25	0.0005	1.14965	151.29815
50	50	15	0.0005	0.54578	61.38370
50	50	10	0.0005	0.69323	21.02685
50	50	3	0.0005	0.45962	2.63978

B. Learn Rate

The next hyperparameter I looked at was the learn rate. I knew from previous experience that small learning rates tend to perform the best, however, I wanted to ensure the same results applied to the LSTM. Thus, like the timestep experiment, I held my learning rate constant while changing the other hyper parameters to determine how the learning rate affects my model's performance. Doing this allowed me to confirm my suspicions, the larger learning rate, the rate closer to one, leads to overfitting as seen in Table II. When the learning rate is 0.1, the training error is very good, yet the testing error is poor, much like with the larger number of timesteps. Yet we can see that as we scale down the learning rate, we have less overfitting as our testing MSE improves drastically to match that of the training MSE. The data in the table ultimately guides us to the conclusion that as we decrease the learning rate, we can improve the accuracy of our model and reduce overfitting.

TABLE II. Results of the Learn Rate Test

Hyper Parameters				Mean Squared Error (MSE)	
Epochs	*Hidden Units*	*Timesteps*	*Learn Rate*	*Training*	*Testing*
50	50	3	0.1	1.84364	228.32686
50	50	3	0.05	1.99723	231.29976
50	50	3	0.005	0.66623	34.44771
50	50	3	0.0005	0.65502	31.96565
50	50	3	0.00005	1.12402	17.53649

C. Hidden Units

We conduct the same test on the number of hidden units in order to once against determine how this hyper parameter affects the performance of our model. Holding the remaining hyper parameters constant, we change the number of hidden units to determine its effect. I noticed

during this test, that much like the timestep test, the more hidden units there are, the longer the model takes to train. This, much like the timestep test, makes sense, as the more hidden units there are, the more calculations the LSTM cell is required to make, leading to longer times. Looking at our data from Table 3, we can see that the lower number of hidden units leads to overfitting. Yet at the same time, when the number of hidden units gets too high, we also experience overfitting. From Table III, we can see that when the number of hidden units decreases from 50, our testing error begins to rise rapidly. In addition to this, when we increase from 100 hidden units to 200 hidden units, we can see a similar phenomenon. Thus, for my data, around 50 to 100 hidden units will result in the best performance and the least amount of overfitting.

TABLE III. RESULTS OF THE HIDDEN UNITS TEST

Hyper Parameters				Mean Squared Error (MSE)	
Epochs	*Hidden Units*	*Timesteps*	*Learn Rate*	*Training*	*Testing*
50	200	3	0.0005	0.39452	12.11142
50	100	3	0.0005	0.41648	4.21285
50	50	3	0.0005	0.52686	4.20113
50	25	3	0.0005	0.43833	9.89657
50	10	3	0.0005	1.38584	46.54433
50	5	3	0.0005	0.51934	44.25601

D. Epochs

The last hyper parameter to test is the number of epochs. My initial prediction is that as the number of epochs increases, the data will begin to perform better. However, I am also predicting that too many epochs will yield to overfitting as that may lead to the model "memorizing" the training data. Holding the other hyper parameters constant while changing the number of epochs yields me the data found in Table IV. The results in this table show that a low number of epochs will yield poor testing MSE's, which is mirrored as we increase the number of epochs to be too high. The table does follow my hypothesis, because as we increase epochs, the model begins to overfit slightly, but if we keep a low number of epochs, we don't have enough iterations to fully optimize our model. Thus, I can gather from my data that setting the epochs to be roughly 200 to 400 iterations would best support my data.

TABLE IV. RESULTS OF THE EPOCH TEST

Hyper Parameters				Mean Squared Error (MSE)	
Epochs	*Hidden Units*	*Timesteps*	*Learn Rate*	*Training*	*Testing*
500	25	3	0.0005	0.63030	9.05691
400	25	3	0.0005	0.77874	5.99203
200	25	3	0.0005	1.80252	6.21089
100	25	3	0.0005	0.48385	5.69767
50	25	3	0.0005	0.83826	30.92083

E. Most Accurate Model

Based upon the data from the experiments that I conducted, I was able to find a model that had great training and testing mean squared errors, while avoiding overfitting. Basing it off the experiments, this model uses 3 timesteps, 50 hidden units, a learning rate of 0.00005, and 200 epochs. After the training and testing of the model, I obtained a training MSE of 0.65359 and a testing MSE of 3.14821. We can see the outcome of our model by plotting the actual stock data and the predicted stock data on the same plot. The plot for the training prediction against the training data is shown on Fig. 3 while Fig. 4 shows the testing prediction against the testing data. These figures give us a visual representation of the accuracy of our model, as we can side by side compare the predicted data with the actual data for our model.

V. CONCLUSION AND FUTURE WORK

A. Conclusion

Using the Long Short Term Memory model allowed me to make predictions on my data while avoiding common issues of simple Recurrent Neural Networks such as the vanishing gradient problem. As a result of the data that I found while creating my LSTM, I can understand how the Long Short Term Memory Network can be used to predict time series data. I was able to learn that too many timesteps in my dataset will lead to overfitting of the data, and that lowering the number of timesteps allows for more accurate predictions while keeping the training of the data fast. For

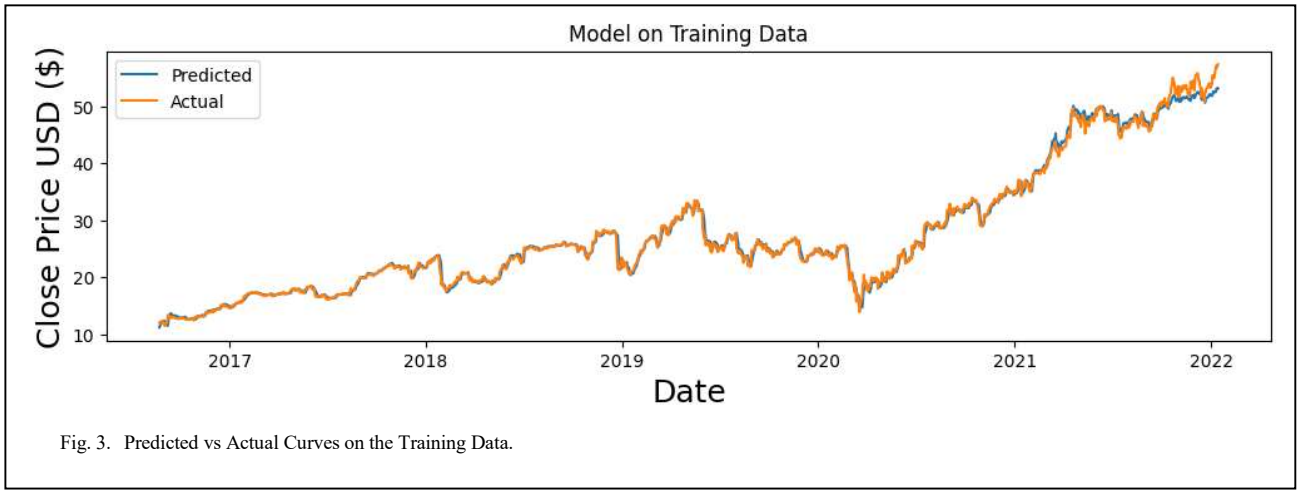

Fig. 3. Predicted vs Actual Curves on the Training Data.

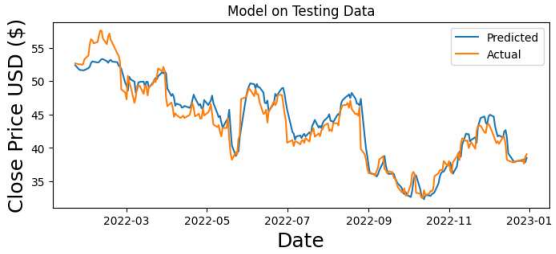

Fig. 4. Predicted vs Actual Curves on the Testing Data.

my specific model and dataset, the Dell Technologies Inc. stock data, I was able to find that three timesteps was able to sufficiently predict the closing price of the stocks with little overfitting. I was also able to make the determination that a higher learning rate in an LSTM network leads to overfitting, meaning I kept my rate around 0.0005 – 0.00005 in order to achieve strong performance and results. I was also able to understand how hidden units impact the performance, as there is an upper bound and lower bound to the number of units used, where crossing those bounds leads to overfitting. I ultimately found that for my model, and hidden unit size of around 50 gave me the best results. I was finally able to understand how the number of epochs has a similar effect that the number of hidden states has. With too many epochs, overfitting can occur, yet with too little, the model doesn't have enough iterations to properly optimize the model, leading to poor performance. With all these findings, I was able to create an accurate model that predicts my dataset with a low mean squared error and high accuracy.

B. Future Work

Although I used the Long Short-Term Memory Network, there have been advances in the processing of sequence-based data that can improve my results and expand upon my current model. For example, Transformers, sometimes referred to by an encoder-decoder architecture, is a model that can pass an input sequence parallelly allowing for an increase in speed of the training process [12]. For example, if we are trying to predict the next word in a sentence, rather than evaluating each word sequentially like in an RNN, the Transformer can evaluate every word at the same time, increasing the speed of the entire process. In addition to the increase in speed, Transformers overcome the vanishing gradient issue by using layers that are known as multi headed attention layers. These layers focus on certain parts of the input while giving the rest of the input less emphasis [12]. Ultimately, the parallelism and the ability to combat the vanishing gradient problem makes this type of model ideal in comparison to an RNN, as it is faster and more efficient. Thus, to continue my research, I would look more into the implementation of the Transformer, comparing the results with that of my LSTM model.

REFERENCES

[1] International Business Machines Corporation, "What are recurrent neural networks?," ibm.com. https://www.ibm.com/topics/recurrent-neural-networks. Accessed: Nov. 28, 2023.

[2] E. Guresen and G. Kayakutlu, "Definition of artificial neural networks with comparison to other networks," *Procedia Computer Science*, vol. 3, pp. 426–433, 2011. doi:10.1016/j.procs.2010.12.071. [Online]. Available: https://doi.org/10.1016/j.procs.2010.12.071.

[3] D. Kalita, "A Brief Overview of Recurrent Neural Networks (RNN)," analyticsvidhya.com. https://www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn/. Accessed Nov. 23, 2023.

[4] Y. Bengio, I. Goodfellow, A. Courville, "Sequence Modeling: Recurrent and Recursive Nets" in *Deep Learning*. Massachusetts, MIT Press, 2017, ch 10., pp. 368-415. Accessed: Nov. 23, 2023. [Online]. Available: https://www.deeplearningbook.org/contents/rnn.html.

[5] Baeldung, "Prevent the Vanishing Gradient Problem with LSTM," baeldung.com. https://www.baeldung.com/cs/lstm-vanishing-gradient-prevention. Accessed: Nov. 23, 2023.

[6] S. Saxena, "What is LSTM? Introduction to Long Short Term-Memory," analyticsvidhya.com. https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/. Accessed Nov. 23, 2023.

[7] C. Olah, et al., "Understanding LSTM Networks," Colah's Blog, 2015 [Online]. https://colah.github.io/posts/2015-08-Understanding-LSTMs. Accessed: Nov. 24, 2023.

[8] K. Shenoy, "LSTM Back-Propagation – The Math Behind the Scenes," medium.com. https://kartik2112.medium.com/lstm-back-propagation-behind-the-scenes-andrew-ng-style-notations-7207b8606cb2. Accessed: Nov. 25, 2023.

[9] U. Ankit, "Transformer Neural Networks: A Step-by-Step Breakdown," builtin.com. https://builtin.com/artificial-intelligence/transformer-neural-network. Accessed: Nov. 28, 2023.

[10] Y. Bengio, I. Goodfellow, A. Courville, "Deep Feedforward Networks" in *Deep Learning*. Massachusetts, MIT Press, 2017, ch 6., pp. 164-223. Accessed: Nov. 28, 2023. [Online]. Available: https://www.deeplearningbook.org/contents/mlp.html.

[11] Y. Bengio, I. Goodfellow, A. Courville, "Optimization for Training Deep Models" in *Deep Learning*. Massachusetts, MIT Press, 2017, ch 8., pp. 271-325. Accessed: Nov. 28, 2023. [Online]. Available: https://www.deeplearningbook.org/contents/optimization.html.

[12] R. Kwiatkowski, "Gradient Descent Algorithm – a deep dive," Towards Data Science. https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21. Accessed: Nov. 28, 2023