# Math Theory of Long Short-Term Recurrent Neural Networks and Their Role in Machine Learning

Cade Kennedy
*Department of Computer Science*
*Tennessee Technological University*
Cookeville, USA
chkennedy42@tntech.edu

***Abstract*—Long Short-Term Memory (LSTM) Networks have become a key piece in machine learning for modeling sequential data, speech recognition, and time-series analysis. Despite their widespread use, the underlying mathematical theory of Recurrent Neural Networks (RNNs) and LSTMs is often overlooked or misunderstood. This paper hopes to provide a full mathematical analysis of RNNs and LSTMs, giving both intuitive insights and detailed breakdowns of the mathematics.**

**This paper will begin by explaining the foundational structure of RNNs, showing their sequential nature and mathematical framework, including the propagation of hidden states and the application of activation functions. Then, we explore the limitations of RNNs and how LSTM has been adapted to combat these issues. Using LSTMs core mechanisms, including memory cells, forget gates, input gates, output gates, and the candidate layer, I show how they all work together to update and retain information.**

**Through theoretical explanations and practical examples, this paper demonstrates how the architecture of LSTMs allows efficient handling of long-term dependencies, making them a useful tool in machine learning.**

## I. Introduction

Long Short-Term Memory Recurrent Neural Networks (LSTM RNN) have begun to develop a significant impact on many different fields of machine learning that utilize sequential data such as language modeling, speech recognition, machine translation[1], and time-series analysis. This has led to the widespread use of LSTM in these respective fields. Unfortunately, many researchers seem to not only struggle to understand the mathematical theory behind Recurrent Neural Networks and LSTMS, but they also seem to completely neglect it in their reports of methodology.

In this paper, I aim to take a deeper dive into the inner workings of Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks. My goal is to provide the reader with a better understanding of the intuitive mathematics behind these models, offer a complete mathematical breakdown, and demonstrate how these mathematical foundations have led to the development of one of the most influential models for sequential data analysis.

## II. Mathematical Theory Behind RNN

Before getting into the mathematical theory of LSTM networks themselves its important to understand the workings of its foundational structure, RNNs. In this portion of the paper, I will give a high level explanation of how RNNs operate and then build that basic understanding to provide proven mathematical theory of these RNNs.

### A. Use-Case of RNN

For the sake of understanding the basics of an RNN let's first consider a real-world application that can benefit from an RNN. Given a dataset of time-series data containing:

- Close - a floating point value containing the closing price of a stock at that given Date.
- Date - as a date-time variable formatted as Year-Month-Day-Hour:Minute:Second (YYYY-MM-DD-HH:MM:SS),
- Volume - a floating point value containing the total number of shares bought and sold at that given Date

With this data let's say we want to predict the closing price of the following day using the previous Close value, Volume value, and Date. RNNs allow us to make a "Many-to-One" model that takes in many sequences as input and generates one output [2]. In RNNs, input is considered as time steps. A time step for our data could potentially look like "$455.56" is Date(0), "35M" is Date(0), "$465.76" is Date(1), and "89M" is Date(1).[2]

### B. Basics of RNN

Assuming a basic understanding of a Feed-Forward Neural Network (FFNN), a Recurrent Neural Network (RNN) builds upon this structure. Each cell in an RNN utilizes a set of FFNNs due to the presence of time steps in the input. In an FFNN, we have input $X$, hidden layers $H$, and output $y$. In an FFNN, the weights between hidden layers, such as $W_{h1}$ and $W_{h2}$, are typically different, as each hidden layer may have its own set of learned parameters. By leveraging a set of FFNNs, we can understand the basic structure of an RNN. Unlike an FFNN, in an RNN, the hidden layer values are calculated not only from the current input but also from the values at previous time steps. Additionally, the weights $W$ at the hidden layers in an RNN remain the same across all time steps. Figure 1 shows a visualization of a basic FFNN.

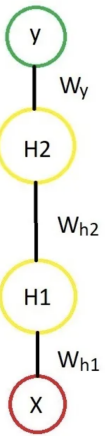


Fig. 1. Visualization of a basic Feed-Forward Neural Network (FFNN).

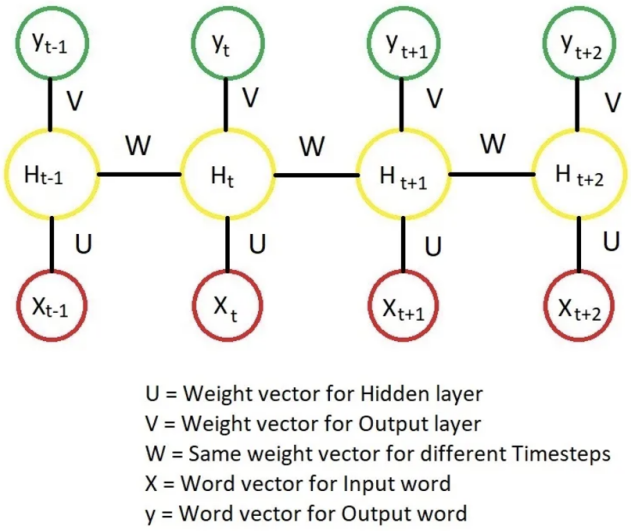


Fig. 2. Illustration of a Recurrent Neural Network (RNN) demonstrating sequential hidden layer calculations.

Figure 4 builds on this foundation, demonstrating the sequential calculation of hidden layer values at each time step in an RNN.[2]

### C. Mathematical Theory of RNN

Now that we have established the basics of how an RNN functions, I will explain the mathematical calculations of the hidden layer values at each time step, and how these values influence subsequent calculations and the final model output. Using Figure 4 a RNN first calculates the Hidden layer value at each time step. So, H(t) will be calculated by applying some activation function using the weighted sum of the current input $(input*H_W)$ and the previous hidden state $(W*H_{t-1})$, where $W$ is constant across all time steps. The resulting equation for each hidden layer value being:

$$H(t) = ActivationFunction(input*H_W + W*H_{t-1}) \quad (1)$$

$ActivationFunction$ can vary from tanh, ReLu, and sigmoid. Hyperbolic tangent (tanh) is typically used to condense the output into a range of $[-1, 1]$ which gives it the ability to balance gradients in sequential data. As for ReLu, it condenses the output into a range of $[0, \infty)$ giving it an advantage in helping mitigate vanishing gradients. Sigmoid is the activation function that will mainly be used in LSTMs. The sigmoid activation function condenses its output into a range of $[0, 1]$ which allows for probabilistic interpretations of the output.[3]

| | Propagation |
|---|---|
| Sigmoid | $y_s = \frac{1}{1+e^{-x_s}}$ |
| Tanh | $y_s = \tanh(x_s)$ |
| ReLu | $y_s = \max(0, x_s)$ |

Fig. 3. Activation Functions and their Propagation Equations.

After determining the appropriate $ActivationFunction$ we can now build on Equation 1 to calculate $H(t)$ for all time steps using the following steps[2]:

- Calculate $H_{t-1}$ from $U$ and $X$.
- Calculate $y_{t-1}$ from $H_{t-1}$ and $V$.
- Calculate $H_t$ from $U$, $X$, $W$, and $H_{t-1}$.
- Calculate $y_t$ from $V$ and $H_t$, and so on.

Step one is determining the value of the previous hidden state value $H_{t-1}$ using the inputs of the constant Weight vector for the Hidden layer $U$ and Value vector for Input Value $X$. Following this, compute the previous output $y_{t-1}$ by applying the Weight vector for the Output layer and $V$ to the previous Hidden state value $H_{t-1}$. Now the current Hidden state value $H_t$ can be found using the current input $X$, weights $U$ and $V$ and $W$, and the calculated previous Hidden state value $H_{t-1}$. Finally, the output $y_t$ can be found by applying the Weight vector for the Output layer $V$ to the calculated current Hidden layer value $H_t$. After finding $y_t$ these four steps can be repeated using the updated calculated values for every time step. It is important to note that $U$ and $V$ are weight vectors, meaning their values are different for each time step; being randomized initially[2].

### D. Back Propagation for Error Calculation

Once the RNN has calculated all of its values and finished all of its output, we must find any sort of error that the model may have produced. To calculate this error we will use well-known Neural Network methods "Back Propagation."

Back Propagation is a calculus-based mathematical method that uses Output as the input for each time step. Essentially,

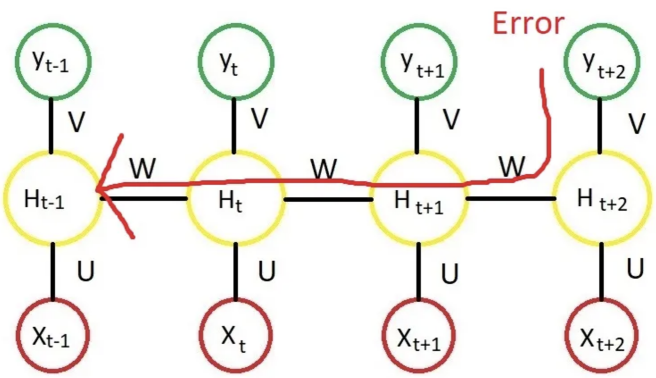


Fig. 4. Illustration of Back Propagation in RNN.

working backward through the RNN to find the model's error on its given Output compared to the original given Input of the dataset. Using the chain rule we can find the gradient of the loss at each time step as follows[2]:

**Backpropagation for RNN**

$$\frac{\partial J_t}{\partial v} = \sum_t \frac{\partial J_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial v}$$

$$\frac{\partial J_t}{\partial w} = \sum_t \frac{\partial J_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial w} = (1 - h_t^2) \cdot h_{t-1}$$

$$\frac{\partial J_t}{\partial u} = \sum_t \frac{\partial J_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial u} = (1 - h_t^2) \cdot x_t$$

By finding the error at each time step $\frac{\partial J_t}{\partial v}$ for instance, we can summarize every sequential error (shown as $\sum_t$) to calculate the TOTAL Error of the model.

## III. Mathematical Theory Behind LSTM

Now that the mathematics behind Recurrent Neural Networks has been established, I will now go into the basics of LSTM as well as the mathematical backbone of these incredible models.

Looking back at the discussion on the back-propagation of RNN a common problem that RNNs face is what mathematicians call a Vanishing Gradient or an Exploding Gradient. The Vanishing Gradient problem occurs during back-propagation when activation function gradients diminish across layers in a deep neural network. This leads to tiny weight updates, slowing or even halting training.[4] The Exploding Gradient problem occurs during backpropagation in deep neural networks when gradients grow excessively large as they are propagated backward through the layers. This results in massive weight updates and difficulty in convergence during training.[4] These two problems led to the development of many solutions such as RMS Propagation, GRU's, and most importantly LSTMs.[5]

### A. Basics of LSTM

Much similar to an RNN, LSTMs have time steps. However, the key advantage of an LSTM RNN is that it contains "Memory" within each cell of the LSTM. At it's core the cell of an LSTM contains[5]:

- **Forget Gate** $f$
- **Candidate Layer** $\tilde{C}$
- **Input Gate** $I$
- **Output Gate** $O$
- **Hidden State** $H$
- **Memory State** $C$

Keeping those key elements in mind, Figure 5 visually represents how these cells are organized internally. Before getting into the meat of the "Memory" itself it is important to understand that:

- $C_{t-1}$ represents the Previous Cell Memory
- $C_t$ represents the Current Cell Memory
- $H_{t-1}$ represents the Previous Cell Output
- $H_t$ represents the Current Cell Output
- $X_t$ represents the Input Vector

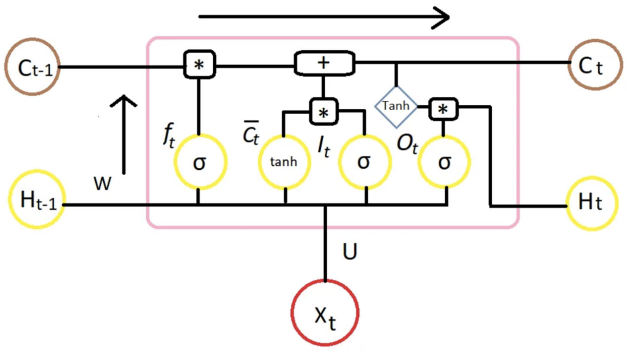


Fig. 5. Illustration of a singular LSTM cell.

Basic operation of an LSTM at each time step will contain an input of $X$ which is the current input, $H$ which is the previous Hidden state of the model, and $C$ which is the previous Memory state of the model. This outputs an updated Current Hidden state $H_t$ and Current Memory state $C_t$ to be used in sequential time steps as the next memory cell's input. So, you may be asking how is this different from the way and RNN works and the answer is the mathematical breakdown of the Memory cell itself.

### B. Mathematical Theory of LSTM

Thinking back to our key components of a memory cell, the first thing to focus on is the different gates and layers. The forget gate is a single-layered neural network using Sigmoid as its activation function. Similarly, the input and output gate work the exact same way even in their activation function. Unlike the others, the candidate layer uses the hyperbolic tangent activation function to make its output. These gates take input vector $U$ and previous Hidden state $W$ and apply their respective activation function to a concatenation of these

two inputs that produce an output vector (between 0 and 1 for sigmoid, -1 to 1 for hyperbolic tangent)[5]. So, the output of an LSTM memory cell for every output is four vectors $f$, $\tilde{C}$, $I$, and $O$.

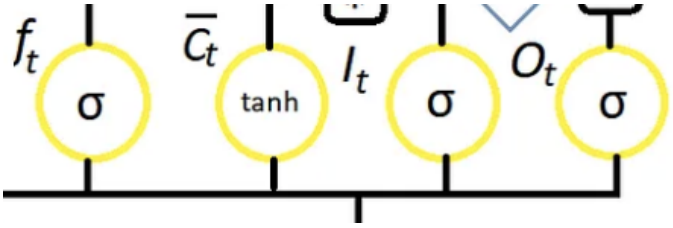


Fig. 6. Illustration of Output Vectors.

Figure 6 zooms into the individual gates within the memory cell shown in Figure 5. After the discussion of these gates, we can get a better understanding of how these vectors take in input as well as function as output visually in the grand scheme of things, but how do these Output Vectors give the model the ability to memorize?

To get a better understanding of how this model develops memory let's think back to our real-world example about the stock market closing price. Imagine a scenario where over the past few days a stock's closing price has been increasing due to a company road map release. The LSTMs captures this in $C_t$ which is then used by the model to predict the stocks closing price and stores it in $H_t$. However, let's say that some days later Forbes magazine release a statement highlighting the flaw of that company's road map and produces negative sentiment around the stock's current price. This new information is introduced, and the forget gate $f$ is utilized to update the memory of $C_t$. Mathematically, the forget gate operates using the following equation:

$$C_t = C_{t-1} \cdot f_t \tag{2}$$

In this equation, the current memory state $C_t$ is evaluated using $C_{t-1}$, which represents the retained memory from the previous time step (e.g., the good news of an upward trend in stock price). This is multiplied by $f_t$, the output of the forget gate, which utilizes the sigmoid activation function to produce values between 0 and 1. The forget gate determines whether to retain (values near 1) or discard (values near 0) the previous memory:

The next step involves updating the memory state by incorporating new information. The memory cell uses the input gate $I_t$, which also applies a sigmoid function, to filter the contribution of the new candidate memory $\tilde{C}_t$ generated by the candidate layer (a network with a hyperbolic tangent activation function). The updated memory state is calculated as[5]:

$$C_t = C_t + (I_t \cdot \tilde{C}_t)$$

Given that:

- $C_t$: The updated memory state at the current time step.
- $I_t$: The input gate output, determining how much of $\tilde{C}_t$ to add.
- $\tilde{C}_t$: The candidate memory, which introduces new context or information relevant to the current input.

By combining $C_{t-1} \cdot f_t$ and $I_t \cdot \tilde{C}_t$, the LSTM cell dynamically balances past trends with new data. For instance, in the stock market scenario, if a sudden drop in stock prices occurs (new context), the input gate allows the new information to update the memory while the forget gate selectively retains or discards previous trends, resulting in a context-aware $C_t$ ready to influence the next prediction.

The LSTM then calculates the final output, which is a filtered version of $C_t$. To achieve this, it uses the output gate $O_t$ and the current cell state $C_t$. The process is as follows[5]:

- First, the cell state $C_t$ is passed through the $\tanh$ activation function, scaling its values to the range $(-1, 1)$.
- Next, the result of $\tanh(C_t)$ is multiplied element-wise with the output gate $O_t$, which uses a sigmoid function to determine which parts of the cell state should contribute to the hidden state.
- The hidden state $H_t$ is thus calculated as:

$$H_t = \tanh(C_t) \cdot O_t$$

- $H_t$: Represents the short-term context-aware output of the LSTM for the current time step.
- $C_t$: Retains the long-term memory and is passed to the next time step.

In the stock market example, $H_t$ might represent the predicted price movement for the next time step, taking both old news and new news into consideration. The updated cell state $C_t$ and hidden state $H_t$ are then passed to the next LSTM cell, repeating the process for sequential predictions.

Figure 7 below is an illustration of how these different uses of gates, addition, and multiplication are fed through the remaining portion of the memory cell from Figure 5.

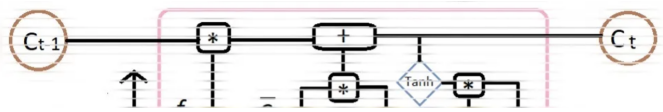


Fig. 7. Illustration of Output Vectors used for Context updates.

## IV. Conclusion

Long Short-Term Memory (LSTM) networks have revolutionized the field of sequential data analysis. Through their innovative use of memory cells, gates, and context updates, LSTMs effectively learn long-term dependencies while combating problems like vanishing and exploding gradients. This has solidified their need in applications such as time-series analysis, natural language processing, and speech recognition.

In this paper, we explored the mathematical foundations of RNNs and LSTMs, demonstrating how their architectures use things like backpropagation and activation functions to process sequential data. The detailed breakdown of LSTM components, including the forget, input, and output gates, highlights their ability to balance the retention of past information with

the integration of new data, giving it the ability to make context-aware decisions.

LSTMs ability to model sequential patterns and long-term dependencies has led to successful sequential data analysis in many different industries. In finance, they have improved the accuracy of stock price forecasting and fraud detection. Additionally, their role in natural language processing has enabled advancements in machine translation, transforming how machines understand and generate human language.

By providing a clear and thorough mathematical perspective, this paper not only deepens the understanding of LSTM networks but also shows their impact on machine learning today. As the field continues to evolve, a strong understanding of this mathematical theory will be necessary for advancing the design and application of sequential models.

## REFERENCES


[1] A. Sherstinsky, "Fundamentals of Recurrent Neural Network (RNN) and ...," Arxiv, https://arxiv.org/pdf/1808.03314 (accessed Nov. 28, 2024).

[2] M. Sanjeevi, "Chapter 10: Deepnlp - recurrent neural networks with math.," Medium, https://medium.com/deep-math-machine-learning-ai/chapter-10-deepnlp-recurrent-neural-networks-with-math-c4a6846a50a2 (accessed Nov. 28, 2024).

[3] J. Brownlee, "How to choose an activation function for deep learning," MachineLearningMastery.com, https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/ (accessed Nov. 29, 2024).

[4] "Vanishing and exploding gradients problems in deep learning," GeeksforGeeks, https://www.geeksforgeeks.org/vanishing-and-exploding-gradients-problems-in-deep-learning/ (accessed Nov. 29, 2024).

[5] M. Sanjeevi, "Chapter 10.1: Deepnlp - LSTM (long short term memory) networks with math.," Medium, https://medium.com/deep-math-machine-learning-ai/chapter-10-1-deepnlp-lstm-long-short-term-memory-networks-with-math-21477f8e4235 (accessed Dec. 1, 2024).