**ADTA 5560 Final Project**

**Student ID: 11651973**

**Name: Keerthi Malathkar**

Table of Contents

[PART I: A Time-series Data Set 2](#_Toc153021802)

[PART II: RNN: Simple RNN with Sine Wave Data 3](#_Toc153021803)

[Import Libraries and generate sine wave data: 4](#_Toc153021804)

[Build model: 4](#_Toc153021805)

[Evaluation: 5](#_Toc153021806)

[PART III: RNN: LSTM Neural Network 6](#_Toc153021807)

[Vanishing gradient problem: 6](#_Toc153021808)

[Exploding gradient problem 7](#_Toc153021809)

[Limitations of the SimpleRNN neural network 8](#_Toc153021810)

[Advantages of LSTM: 8](#_Toc153021811)

[PART IV: RNN: LSTM with Time-Series Data 11](#_Toc153021812)

[Build an LSTM neural network: 11](#_Toc153021813)

[Train the model: 11](#_Toc153021814)

[Test: 12](#_Toc153021815)

[Retrain: 13](#_Toc153021816)

[Forecast: 13](#_Toc153021817)

[PART V: Redesign the LSTM Neural Network to improve performance 14](#_Toc153021818)

[PART VI: Compare Network Performance 16](#_Toc153021819)

[PART VII: Understanding of Simple RNN and LSTM Models 16](#_Toc153021820)

[Part VIII: YouTube Link 17](#_Toc153021821)

## PART I: A Time-series Data Set

Dataset Name: Air Quality

Source: <https://archive.ics.uci.edu/dataset/360/air+quality>

Description:

The dataset comprises 9,358 data points of hourly averaged responses generated by a set of 5 metal oxide chemical sensors integrated into an Air Quality Chemical Multisensor Device. The device was situated in a highly polluted area, specifically at road level, within an Italian city. Data collection spanned from March 2004 to February 2005, encompassing a year and representing the lengthiest openly accessible recordings of air quality chemical sensor responses deployed in the field. Hourly averaged concentrations for CO, Non Metanic Hydrocarbons, Benzene, Total Nitrogen Oxides (NOx), and Nitrogen Dioxide (NO2) were established as Ground Truth, with values provided by a co-located reference certified analyzer.

Data Dictionary:

|  |  |  |
| --- | --- | --- |
| **Variable Name** | **Description** | **Data Type** |
| Date | Collected record data frame | Date |
| Time | Time in Format (HH.MM.SS) | Categorical |
| CO(GT) | True hourly averaged concentration CO in mg/m^3 (reference analyzer) | Integer |
| PT08.S1(CO) | hourly averaged sensor response (nominally CO targeted) | Categorical |
| NMHC(GT) | True hourly averaged overall Non Metanic HydroCarbons concentration in microg/m^3 (reference analyzer) | Integer |
| C6H6(GT) | True hourly averaged Benzene concentration in microg/m^3 (reference analyzer) | Continuous |
| PT08.S2(NMHC) | hourly averaged sensor response (nominally NMHC targeted) | Categorical |
| NOx(GT) | True hourly averaged NOx concentration in ppb (reference analyzer) | Integer |
| PT08.S3(NOx) | hourly averaged sensor response (nominally NOx targeted) | Categorical |
| T | Temperature in C | Continuous |

## PART II: RNN: Simple RNN with Sine Wave Data

Perceptron is the simplest form of neural network, consists of single neuron with adjusted synaptic weight. Each input I is multiplied by a weight W (synaptic strength). The primary principle of RNN is to store the output of particular layer and feeding back to the input to predict the output of the layer. The simple recurrent neural network it is specialized for processing a sequence of data x1 to xn with time index ranging from 1 to n. Example of simpleRNN are speech recognition and audio transformation. The RNN is called recurrent because it performs the repetitive tasks for every element of sequence, while output decision depends on previous computation. The advantage of RNN is it has in-state memory, which capture the computational information calculated so far.

Forward pass formula: In this we observe the total loss for the given input sequence x values paired with output sequence of value y would be the sum of loss over time steps. We use softmax function to obtain output vector y for probabilities of outputs.

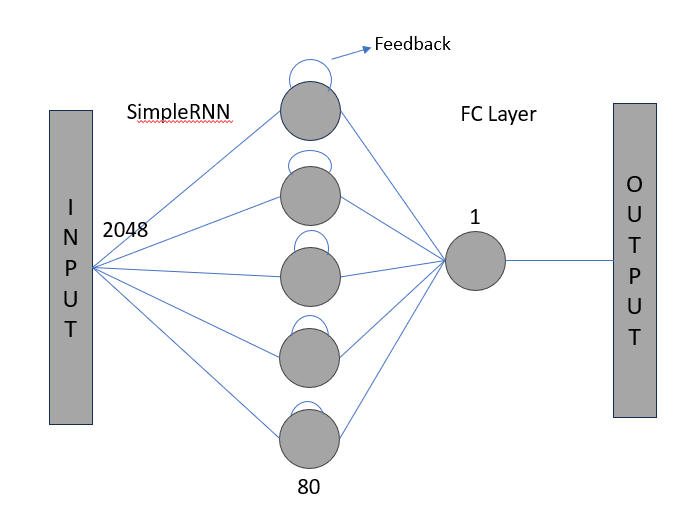


Figure 1

### Import Libraries and generate sine wave data:

Let’s import the important python libraries to work on sinewave generation model. In the below code we have imported the MinMaxScaler, Keras and Tensorflow libraries.

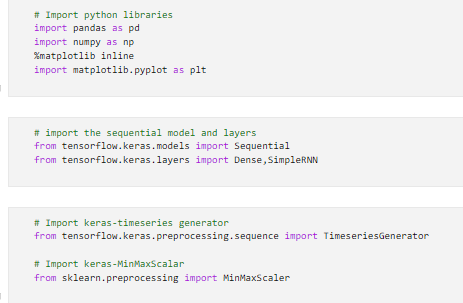


Figure 2

In this project we are working on 2048 data points in the range of 0 to 72. Then loaded the dataset into the pandas dataframe. Apply the split 20% percentage on the original data for test and training model. The total size of dataset is 2048, where as test dataset size after split is 410. Apply the MinMaxscaler function on the train and test dataset to transform the features in the range of (0,1).

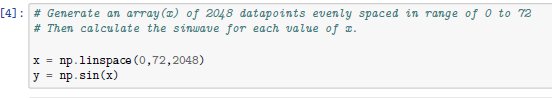


Figure 3

### Build model:

I have used the two layers, simpleRNN with 80 neurons and Fully-controlled neural network (Dense). Now set the length of sequence to 50 for predicting the 51th value of the dataset. Set the feature to 1 as we have only one feature(sine) in the dataset. Then compile the build model using the mean square error (MSE) loss function and adam optimizer.

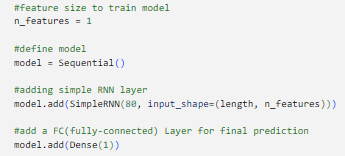


Figure 4

### Evaluation:

After train and testing the mode, lets evaluate the results. In the below image, we see a line chart between loss and epochs. At the beginning, the loss is highest. Upon training the model over different epochs we see that model loss have significantly dropped to nearly zero. Which means a crucial factor in modeling.

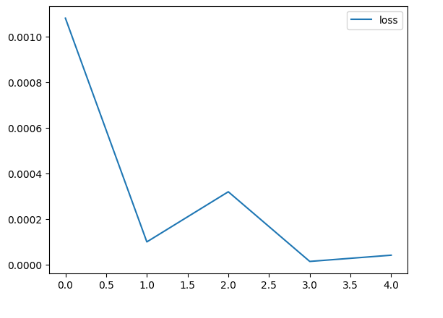


Figure 5

After transforming back, the features to its original values. Added new column for predicted values by name Predictions. Below is the screenshot of the updated dataset with the actual and predicted values. Now, we see the predictions about the data.

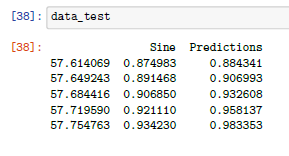


Figure 6

With the given training dataset as input, the simple RNN model has predicted the sine wave values. Using the below plot between series and sine wave values. The blue line represents actual sine wave and predictions represent in orange. We see that the predictions follow actual sine wave at almost all instances. Which means the modeling is working well in doing predictions. However, the slight gap between these lines may be due to loss value we discussed above.

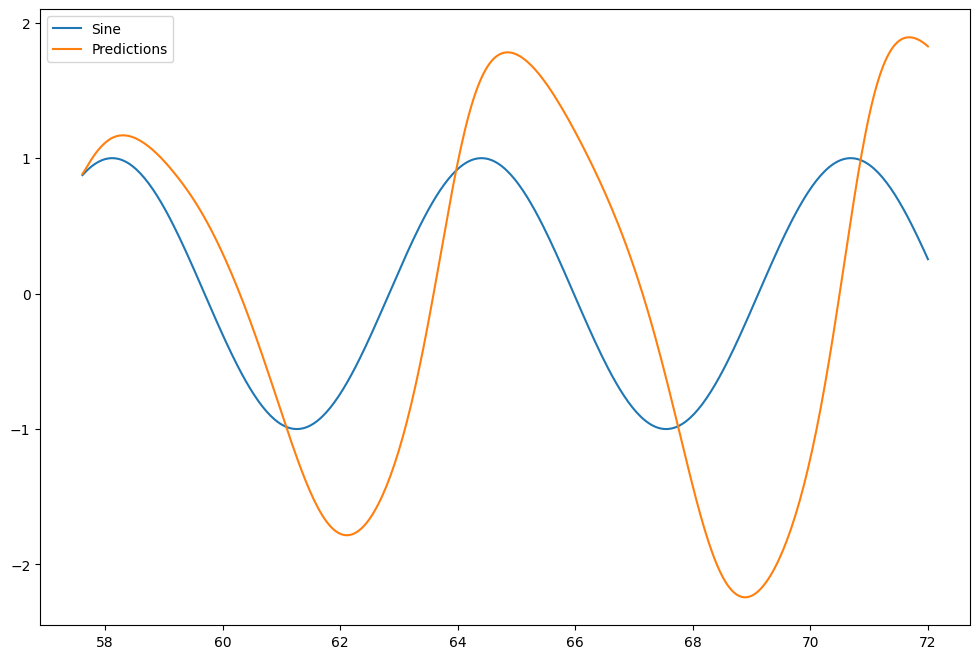


Figure 7

## PART III: RNN: LSTM Neural Network

### Vanishing gradient problem:

With the increase in the layer in the neural network architecture using the activation code which gradient of the loss function to zero, making network hard to train.

Reason:

In the sigmoid function, the activation function squishes the large input space into small input space in range of 0 to 1. Which indeed cause large channge in input of activation function causes the small changes in the output. Therefore deviation become small.

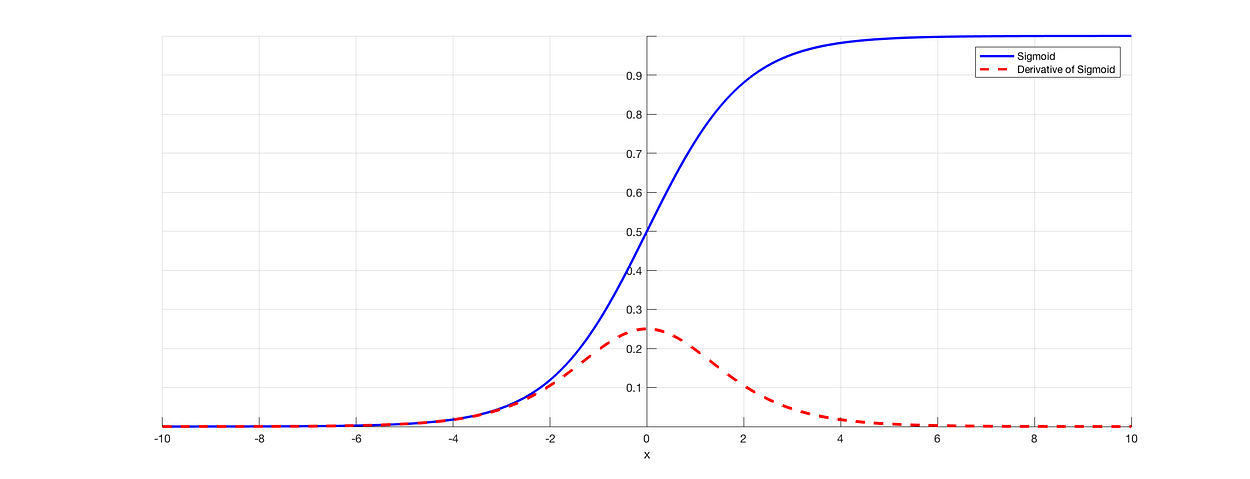


Figure 8

From the above figure 8, we observe as the sigmoid input value increases, its derivative output decreases.

For shallow network with less layers are not an issue for this activation function. But as the layer’s increases, it causes the gradient too small for training to work efficiently. Using the backpropagation, we can find the gradient of neural network. It helps in finding the backpropagation of the network by moving derivative from last year to the initial layer. Using the chain rule, the derivates of each layer are multiplied down the network to compute the derivates of the initial layers. For example, if n layers are used, then the activation function used by these layers will multiplies those n causing the gradient decreases exponentially as we progress it to initial layer.

Solutions:

The simplest solution is of using the functions like ReLU, which doesn’t derivate as layer progress.

Another solution is batch normalization. The problem arises when the large input space is mapped to small one, causing derivates to disappear. It reduce the problem by normalizing the input so the determinant of the input doesn’t exceeds the outer edge of the sigmoid function.

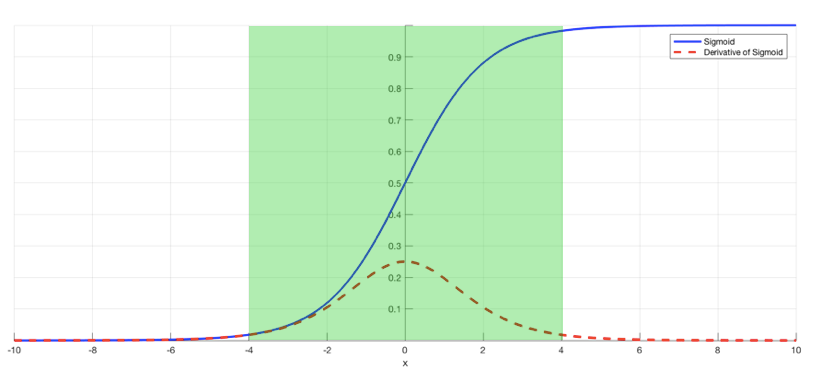


Figure 9

### Exploding gradient problem

The direction and magnitude of an error gradient are determined during a neural network's training process and are utilized to update the network weights in the appropriate direction and to the appropriate degree.

Error gradients can build up during an update in deep networks or recurrent neural networks, leading to extremely large gradients. Large updates to the network weights as a result lead to an unstable network. NaN values can occur when weight values reach an extreme point where they overflow.

Gradients with values greater than 1.0 are repeatedly multiplied through the network layers, causing an explosion through exponential growth. Exploding gradients in recurrent neural networks can lead to, at worst, an unstable network that cannot learn over lengthy input sequences of data and an unstable network that cannot learn from training data.

**Solution:**

Exploding gradients can be addressed in a variety of ways; some best practices are listed in this section.

1. Re-Design the Model of the Network

Exploding gradients in deep neural networks can be avoided by rearranging the network's layers. While training the network, there might also be some advantages to using a smaller batch size.

The exploding gradient problem in recurrent neural networks may be lessened by updating across a smaller number of previous time steps during training, a technique known as truncated Backpropagation through time.

2. Employ networks with long short-term memory

Given the inherent instability in recurrent neural network training—for example, via backpropagation through time, which effectively turns the recurrent network into a deep multilayer Perceptron neural network—gradient exploding can happen in recurrent neural networks.

The Long Short-Term Memory (LSTM) memory units and possibly related gated-type neuron structures can be used to mitigate explosive gradients.

A new recommended practice for recurrent neural networks used in sequence prediction is the adoption of LSTM memory units.

3. Make Use of Gradient Clipping

Even with very deep Multilayer Perceptron networks and large batch sizes, as well as LSTMs with very long input sequence lengths, explosive gradients can still happen.

If explosive gradients are still happening, you can look for them and set a gradient size limit when your network is being trained. We refer to this as gradient clipping.

### Limitations of the SimpleRNN neural network

Feedforward neural networks and multilayer perceptrons are various names for simple neural networks. They are made up of weighted connections between layers of neurons. The data are received by the input layer, processed by the hidden layer(s), and output layer yields the desired outcome.

Some of the drawbacks of simple neural networks are:

Overfitting: When a simple neural network overfits the data, it fails to generalize to new data and instead memorizes the training set. Inaccurate forecasts and subpar performance may result from this. Regularization strategies like dropout, weight decay, and early stopping can help minimize overfitting.

Vanishing gradient: When weights propagate back through a simple neural network, they may experience the vanishing gradient problem, which causes the gradients of the weights to either zero or very small. The network may stop learning as a result, or it may converge slowly. Vanishing gradient can be lessened by utilizing optimization techniques like Adam that adjust the learning rate and non-saturating activation functions like ReLU.

Local minima: When simple neural networks find a suboptimal solution that is not the best option, they can become stuck in a local minima. This may hinder the network's ability to perform better and realize its full potential. Stochastic gradient descent, which adds noise and randomness to escape from local minima, and momentum, which quickens the gradient descent, are two strategies for avoiding local minima.

### Advantages of LSTM:

LSTMs, or long short term memory networks, are a unique class of RNNs that have the ability to learn long-term dependencies. Following their introduction by Hochreiter & Schmidhuber (1997), numerous individuals improved and popularized them. They are now widely used because they perform incredibly well on a wide range of problems.

LSTMs are purposefully made to circumvent the issue of long-term dependency. They don't struggle to learn; rather, being able to retain information for extended periods of time is practically their default behavior!

Every recurrent neural network is composed of a series of neural network modules that repeat. This repeating module in conventional RNNs will have a very basic structure, like a single tanh layer.

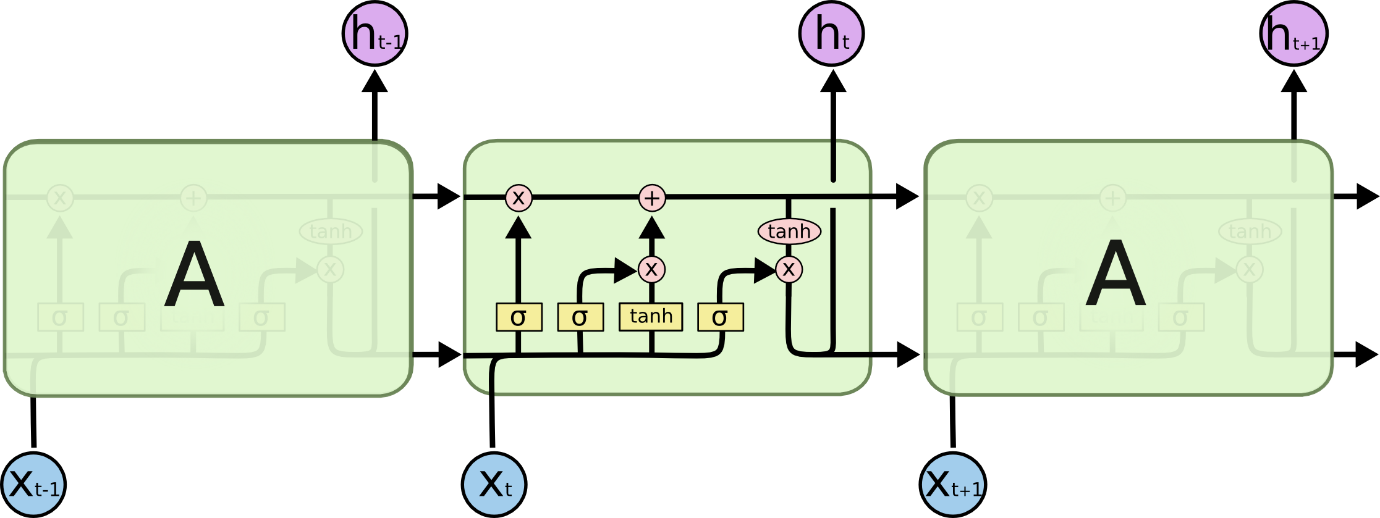


Figure 10

The horizontal line that passes through the top of the diagram, representing the cell state. In certain ways, the cell state resembles a conveyor belt. With only a few minor linear interactions, it runs straight down the entire chain. Information can easily just flow along it unaltered. With the help of structures known as gates, the LSTM is able to carefully control what information is added or removed from the cell state.

Information can be optionally allowed through gates. They consist of a pointwise multiplication operation and a sigmoid neural net layer.Numbers ranging from zero to one are produced by the sigmoid layer, which indicate how much of each component should pass through. A value of one indicates that "everything through," whereas a value of zero indicates "let nothing through."

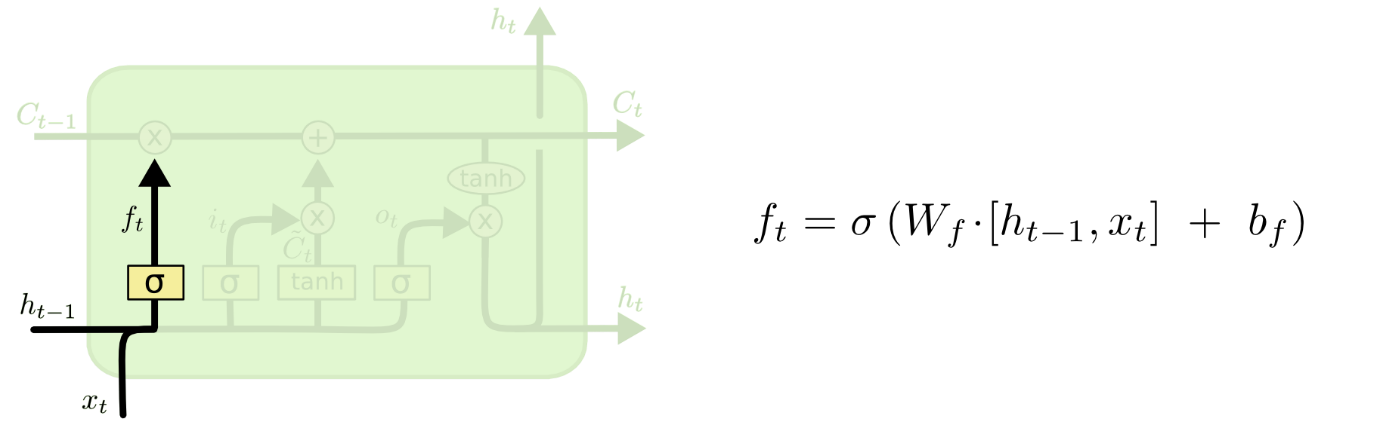


Figure 11

The three gates are present in an LSTM to safeguard and regulate the cell state. Determining what data to discard from the cell state is the first step in our LSTM. The "forget gate layer" is a sigmoid layer that makes this determination. It examines ht−1 and xt, and for each number in the cell state Ct−1, it outputs a number between 0 and 1. "Completely keep this" is represented by a 1 and "completely get rid of this" by a 0.

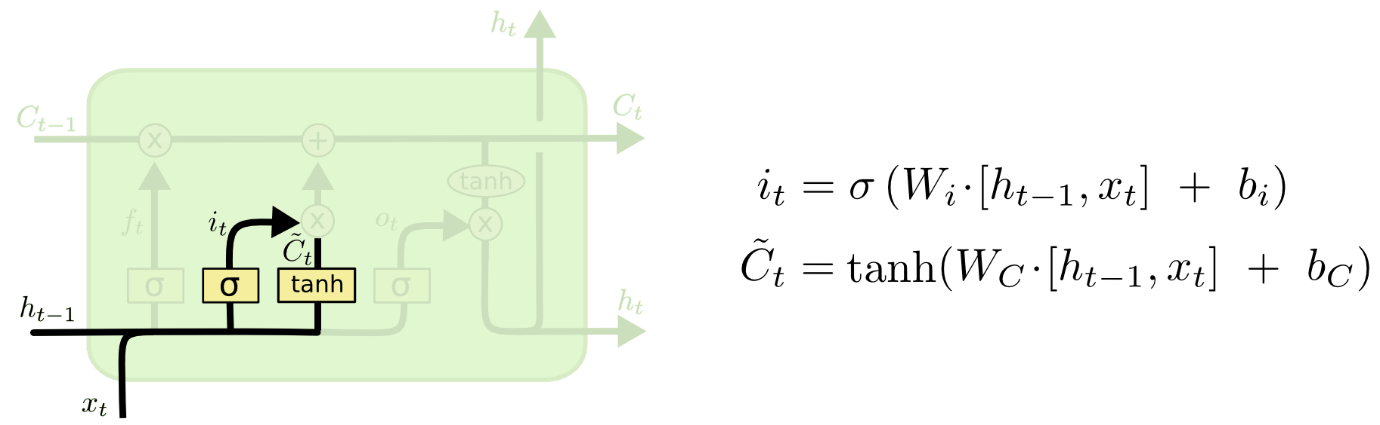


Figure 12

Choosing the new data to be stored in the cell state is the next stage. There are two components to this. First, the values to be updated are selected by a sigmoid layer known as the "input gate layer." After that, a tanh layer generates a vector, C-t, of potential new candidate values that could be included in the state. We'll combine these two to produce a state update in the following step. The old cell state, Ct−1, needs to be updated into the new cell state, Ct. Everything has already been decided in the preceding steps; all that remains is to carry it out.

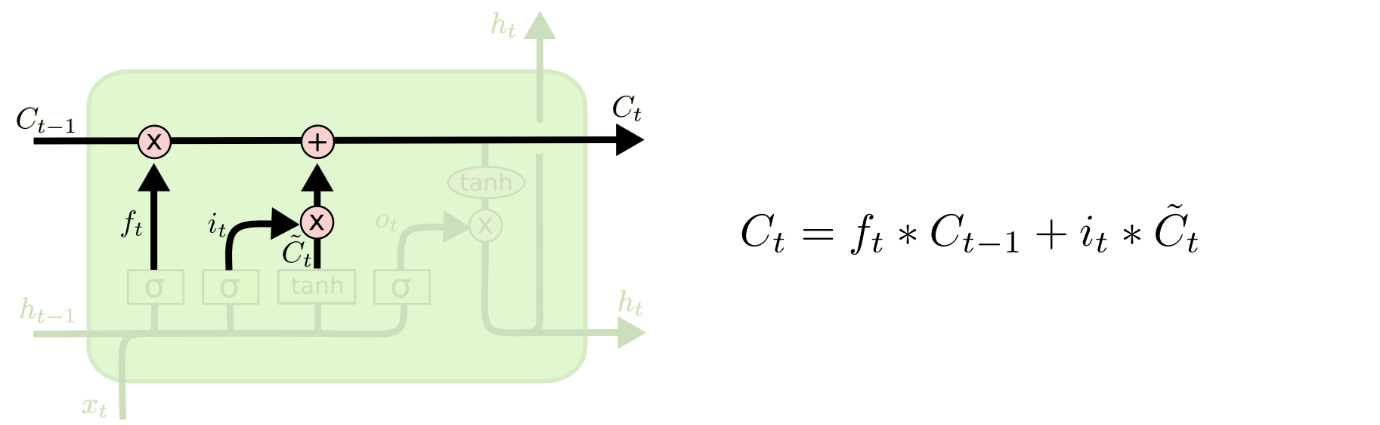


Figure 13

We multiply the previous state by ft, disregarding the information we had previously chosen to ignore. Next, we include it∗C~t. This represents the updated candidate values, adjusted for the amount that we chose to update each state value.

Ultimately, we must choose what we will produce. This will be a filtered version of the output, based on our cell state. The sigmoid layer is run first, and it determines which portions of the cell state we will output. The cell state is then multiplied by the sigmoid gate's output after being passed through tanh to force the values to lie between -1 and 1. This allows us to output only the portions that we have chosen.

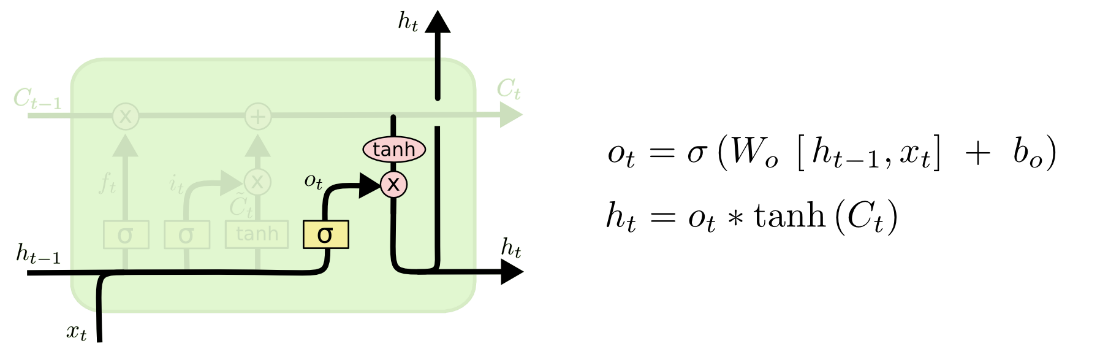


Figure 14

## PART IV: RNN: LSTM with Time-Series Data

### Build an LSTM neural network:

Loaded historical stock price data from the provided CSV file. Explored the dataset's structure, dimensions, and data types. Conducted a statistical summary to understand the distribution of the data. The total length of the dataset is 1382 and columns are 7. We have assigned the total length of 60 as the input sequence.

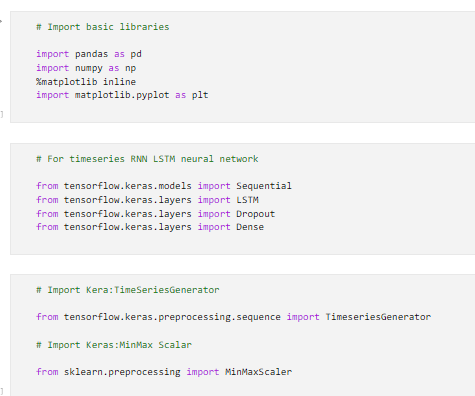


Figure 15

Filtered the dataset to include only the closing prices. Visualized the closing prices over time to gain insights. Split the data into training and testing sets, reserving 10% of the data for testing. After splitting the dataset for train and test, the total length of test is 138 and total length of train is 1244.

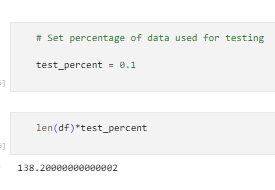


Figure 16

### Train the model:

Scaled the training and testing data using MinMaxScaler to normalize values between 0 and 1. Created a TimeseriesGenerator for training the LSTM model with a sequence length of 60. Also set the timeseries generator batch of 32 to predict every 33th value in the sequence. Now build the model using the sequential LSTM, Dropout, and Dense layers. In the LSTM model we have set the 50 layers and activation function as ‘relu’. The Droput layer is set to 20 percentage and one Dense layer.

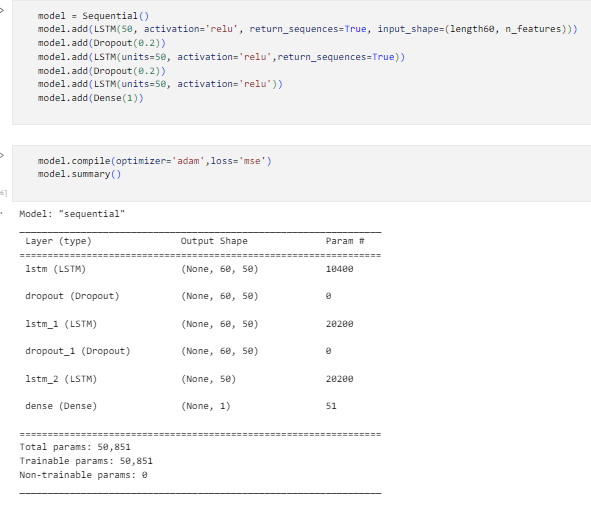


Figure 17

### Test:

Designed a Sequential LSTM model with three layers, each followed by a Dropout layer to prevent overfitting. Compiled the model using the Adam optimizer and mean squared error loss.

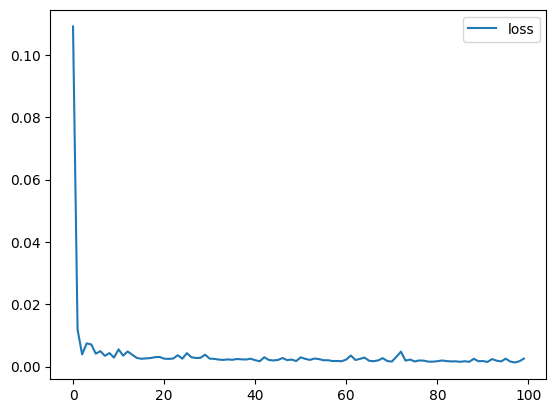


Figure 18

Trained the model for 100 epochs using the TimeSeriesGenerator. Extended the analysis to forecast future stock prices beyond the original dataset. Created a new TimeseriesGenerator for forecasting. Trained the model on the extended dataset for an additional 100 epochs.

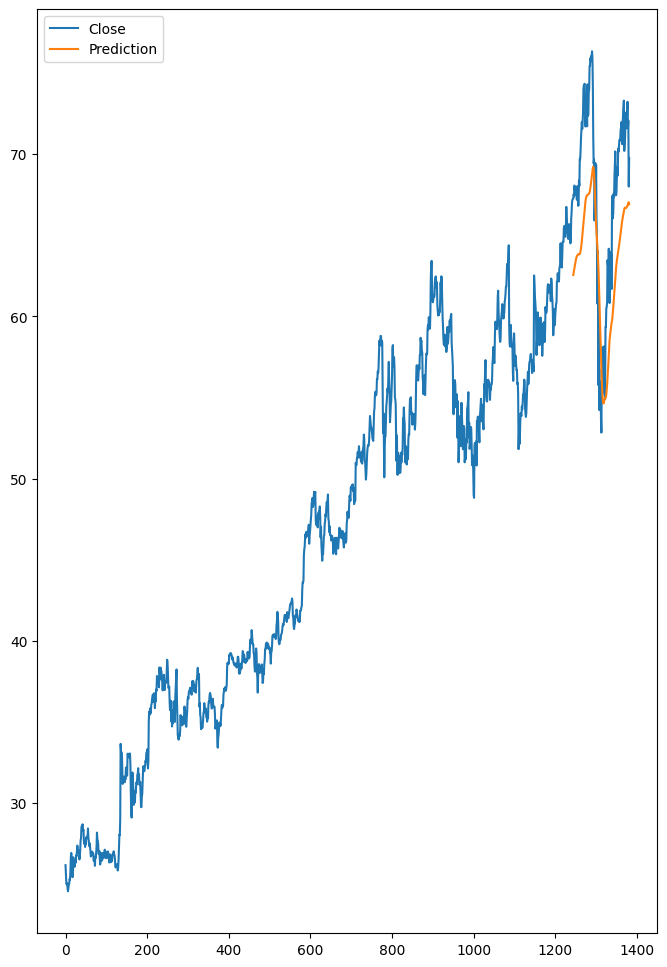
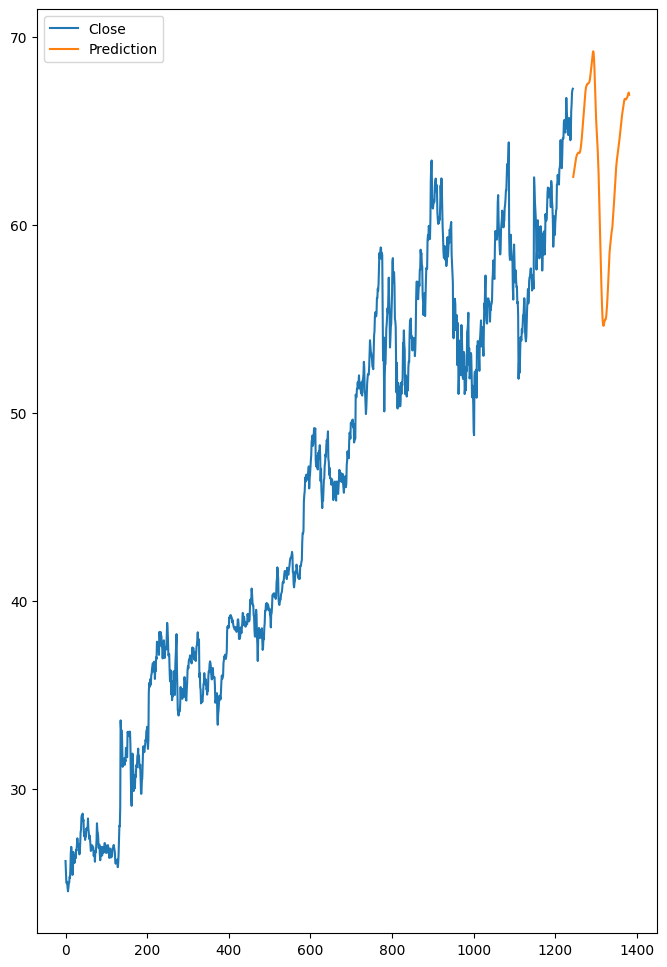
 

Figure 19 Figure 20

### Retrain:

Below screenshot represents the testing data comparision with predicted output. We observe that the predictin of the google stocks are decreased. From the outcome we can conclude that buying the google stock will be a risk as the forecasted value is decreased abruptly.

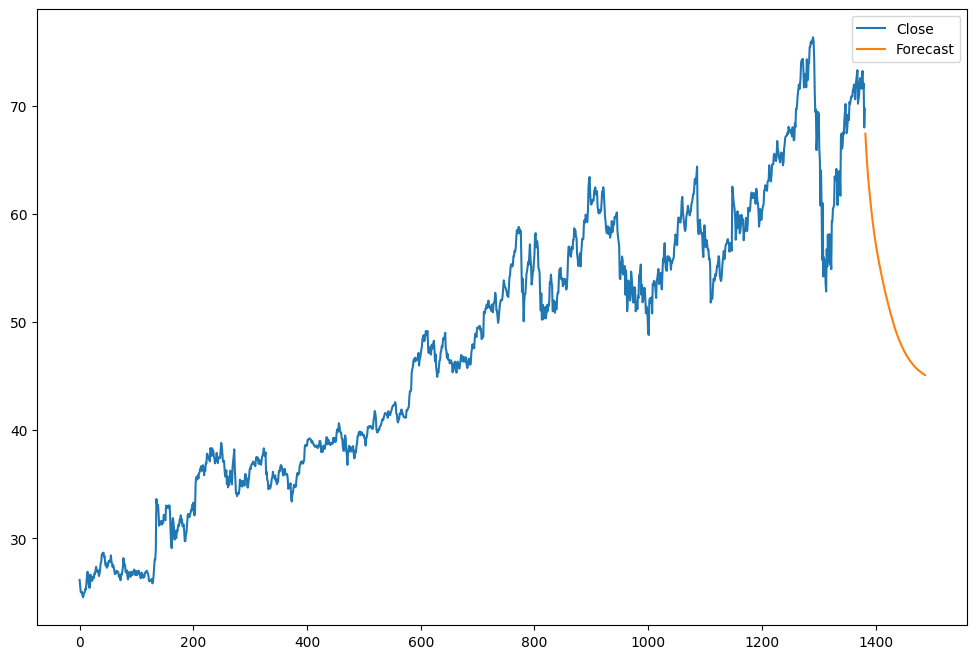


Figure 21

### Forecast:

Now retrained again model using the new dataset from range of July 2020 to November 2020. Below graph shows the plot for validation the model performance on most recent dataset. The model shows accurate results in predicting both the test set and forecasting future stock prices.

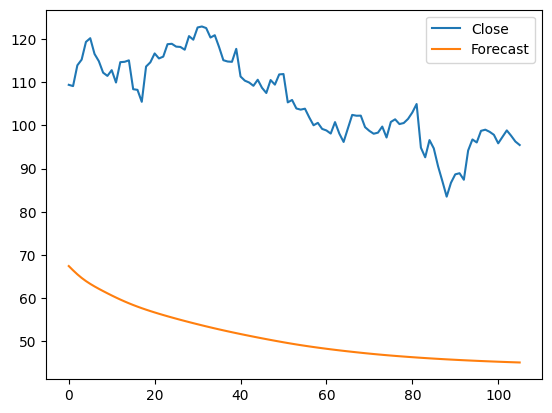


Figure 22

* Percentage of data for testing - **10%**
* How many layers of LSTM? - **3**
* Number of neurons in each LSTM layer? **50**
* Any DropOut layer? - **2**
* If with DropOut layer: the percentage to drop **20%**
* Length of the time-series input sequence **60**
* Batch size for training **32**
* Number of epochs for training **100**

## PART V: Redesign the LSTM Neural Network to improve performance

Now redesigned the LSTM to check the performance of the model. In this analysis we have set the training and testing split percentage to 20. Also changed the batch for prediction to 40. Also changed the neuron number of LSTM layers to 60. We have redesigned the input sequence to 61 and number of batch for predicting the sequence is set to 40 to predict every 41th value in the sequence.

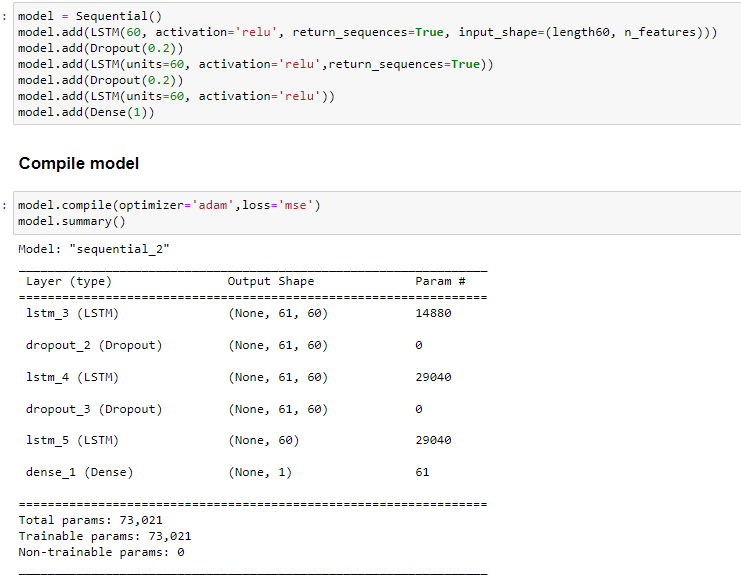


Figure 23

From the prediction plot we observe that

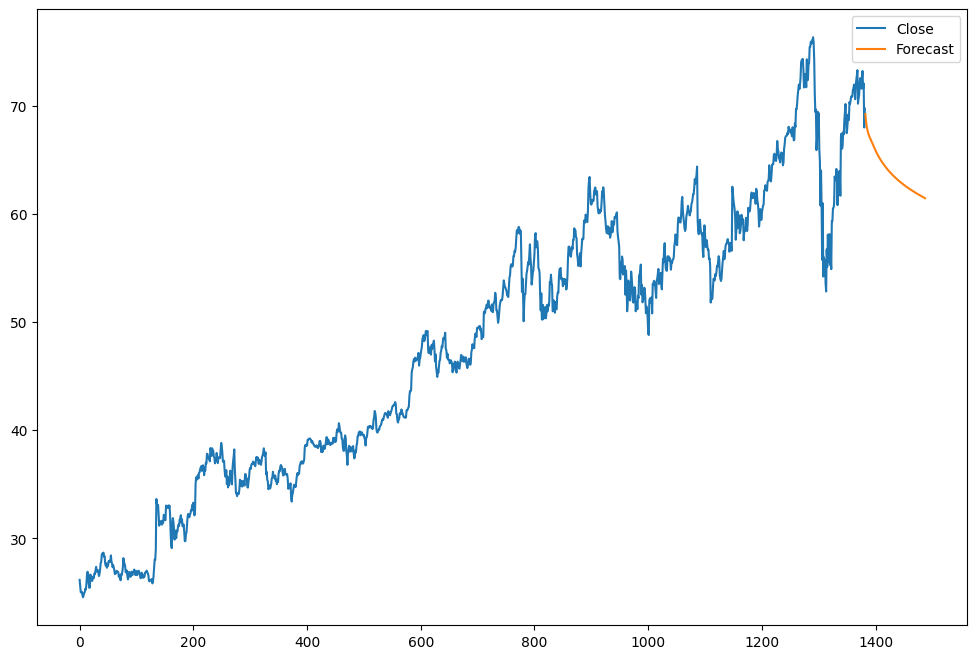


Figure 26

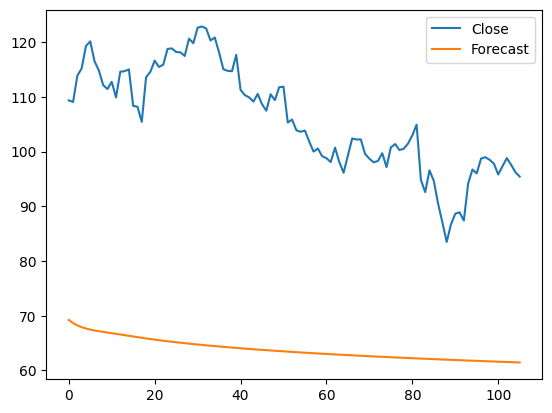


Figure 27

* Percentage of data for testing - **20%**
* How many layers of LSTM? - **3**
* Number of neurons in each LSTM layer? **60**
* Any DropOut layer? **2**
* If with DropOut layer: the percentage to drop **20**
* Length of the time-series input sequence **61**
* Batch size for training **40**
* Number of epochs for training **100**

## PART VI: Compare Network Performance

In this project we have compared the two different LSTM model with different parameters to compare the result of model performance. When compared the both models, we observe that the model loss performance is improved in the PART V. The loss plot for the redesigned model is constantly zero, which states that model has zero error.

When compared the both models’ prediction plot, we observe there is no much difference in the prediction of the model. The forecasted values for both models are same even if the number of neurons are different to both model.

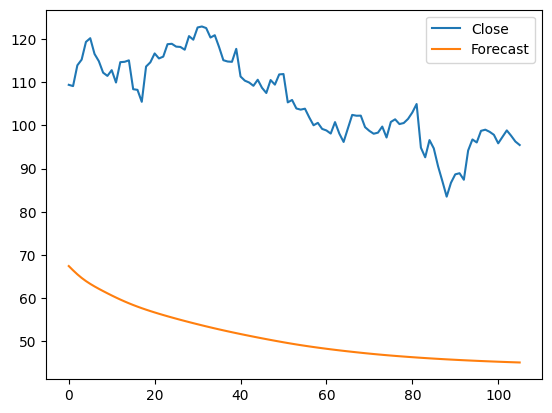
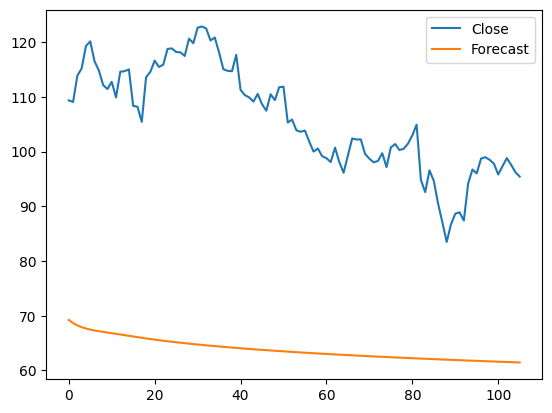


Figure 28

## PART VII: Understanding of Simple RNN and LSTM Models

Because they are the only neural network type with an internal memory, RNNs are strong and powerful neural networks that rank among the most promising algorithms currently in use. Recurrent neural networks are relatively old algorithms, just like many other deep learning algorithms. Although they were first developed in the 1980s, their full potential has only recently come to light. RNNs have become much more prominent as a result of advances in computing power, the vast volumes of data we now work with, and the development of long short-term memory (LSTM) in the 1990s.

RNNs can predict events with high accuracy because of their internal memory, which helps them recall crucial details about the input they have received. For sequential data such as time series, voice, text, financial data, audio, video, weather, and much more, this is the reason they are the recommended algorithm. Compared to other algorithms, recurrent neural networks are capable of forming a much deeper understanding of a sequence and its context.

RNNs encountered two main issues, which are best understood by first understanding what a gradient is.

A partial derivative with respect to its inputs is called a gradient. If you're not familiar with the term, consider it this way: a gradient represents the amount that a function's output changes when its inputs are slightly altered.

Another way to conceptualize a gradient is as a function's slope. A model can learn more quickly and at a steeper slope when the gradient is higher. However, the model ceases to learn if the slope is zero. To put it simply, a gradient measure how all weights change in relation to the error change.

RNNs are extended by long short-term memory networks (LSTMs), which essentially expands memory. It is therefore ideal for learning from significant events with very long intervals between them.

The LSTM has the ability to read, write, and remove data from its memory. This memory can be thought of as a gated cell. A gated cell is one that chooses which information to store and discard based on the weight it gives to each piece of information. Weights are used to assign importance, and the algorithm also learns these. This essentially indicates that it gradually discovers what information is significant and what is not.

Three gates are present in a long short-term memory cell: input, forget, and output. These gates control whether data is removed because it is unnecessary (forget gate), allowed to affect the output at the current timestep (output gate), or allowed to enter new data (input gate).

## Part VIII: YouTube Link

<https://youtu.be/RPX3hFo_Xc8>

References:

1. Vito,Saverio. (2016). Air Quality. UCI Machine Learning Repository. <https://doi.org/10.24432/C59K5F>.
2. Chi-Feng Wang. (2019). The Vanishing Gradient Problem.

<https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484>

1. Jason Brownlee. (2019). A Gentle Introduction to Exploding Gradients in Neural Networks.

<https://machinelearningmastery.com/exploding-gradients-in-neural-networks/>

1. Colah Blogs. (2015). Understanding LSTM Networks.

<https://colah.github.io/posts/2015-08-Understanding-LSTMs/>