CHAPTER 1: NEURAL NETWORKS

INTRODUCTION

The concept of **Deep Learning** evolved in the 1950s, it has taken many advances to become what it is in today's world. The idea behind deep learning is to make a machine think and work like a human brain, in other words, Deep learning architectures are used to build an intelligent and smart Machine that resembles a human brain.

With the help of deep learning, we can use smart computers and mobile phone devices that can identify objects from images, Classify images, and several other uses like Chat Bots, Self-Driving Cars, Human Disease identification, and Classification ETC…

What is making Deep Learning drive the world today?

It is because of the increasing need for automation to reduce manpower dependencies, also because deep learning Machines can accomplish a task more efficiently and they are a reliable source.

In the initial years of its invention it was not as smart and robust as it is today, because they were using a Simple Perceptron i.e. a single Neuron Neural Network, however, it is now developed into a Multi-Layer Neural Network and with the help of free source Python Libraries like Keras, Tensor-flow, etc., we can deploy the deep learning models more effectively in today's World.

Upon completing this chapter, you will get a clear understanding of:

- Basic Introduction and an overview of neural networks.
- Single neuron Neural Network (simple Perceptron).
- The architecture of a Multilayer Network (Artificial Neural Networks).
- Weights Initialization techniques in Artificial Neural Networks.
- Forward Pass in a Neural Network.
- Backward Propagation and Chain rule of Differentiation.
- Loss Functions
- Gradient Descent Optimizer and its types.
- Adaptive Optimizers.

OVERVIEW

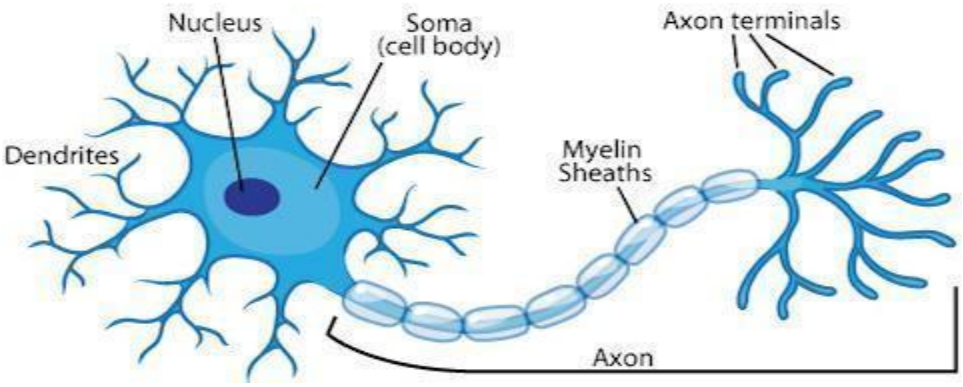
A neural network can be used to solve even the most complex problems of today's world, it can be applied for both Regression and Classification use cases, neural networks are experts in handling linearly non-separable data. That is the reason it has advanced from a single-layer network to a Multilayer network.

What are the various Neural Network Architectures?

- Single Layer Neural network (**Perceptron**)
- Multilayer Neural Network (Artificial Neural Network – **ANN**)
- Recurrent Neural Networks (**RNN**)
- Convolutional Neural Networks (**CNN**).

The whole idea of developing a neural network started from the thought of building a machine that is intelligent and smart, for this, the researchers have used the functioning of a human brain to simulate the same functioning into the artificial neural networks and deploy them into machines. Therefore, a neural network draws its architecture from a human brain neuron.

A human neuron architecture appears like below,



A human brain possesses millions of such neurons, the idea is to make you understand the similarity between human and artificial neurons.

Let us start with a single-layer neural network – Perceptron, then followed by Multi-Layer neural network – ANN (Artificial Neural Network).

SIMPLE PERCEPTRON:

"A Simple Perceptron"

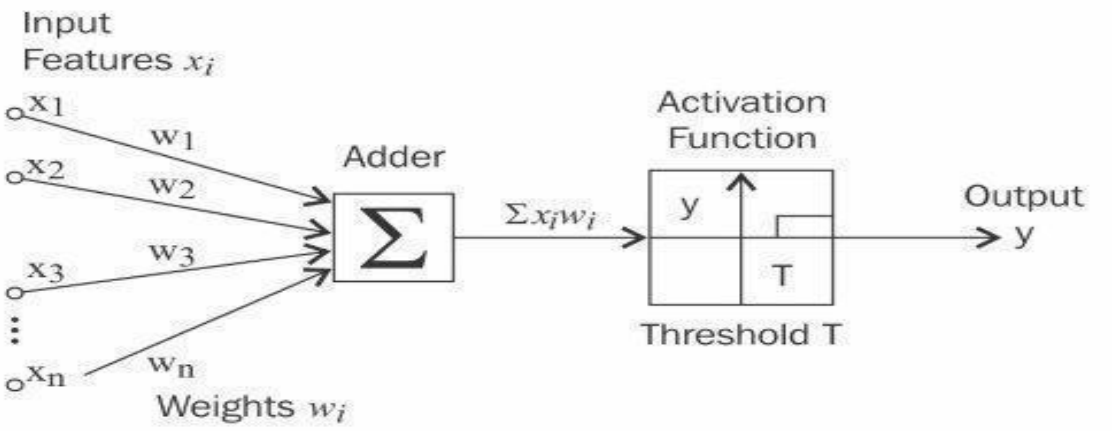
Perceptron is a supervised Machine Learning Algorithm of binary classifiers, i.e. it predicts whether a given input feature vector belongs to a specific class or not.

The Perceptron algorithm was invented in the year 1958 by Frank Rosenblatt. It is a supervised learning algorithm used for 'Linear Binary Classification' use cases. It is the simplest form of any neural network available, it is the base for any neural network architecture, be it a multilayer neural network or a Convolutional Network they are built on Perceptron.

The architecture of a simple Perceptron comprises of:

1) The Input Layer
2) Weights
3) Bias
4) A Single Neuron and
5) The Output Layer

Below is the architecture of a simple perceptron,



Here x1, x2, ... xn are the input variables (or independent variables). As the input variables are fed into the network, they get assigned some random weights (w1,w2...wn). Alongside, a bias (wo) is added to the network (explained below). The adder adds all the weighted input variables. The output (y) is passed through the activation function and calculated using the equation

$$y = g\left(w_0 + \sum_{i=1}^{p} w_i x_i \right)$$

Where,

Wo = Bias,

Wi = weights,

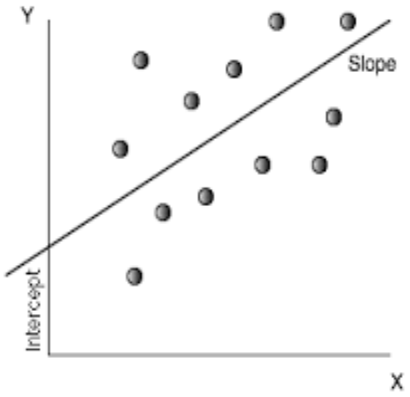
Xi = input variables.

The function **g()** is the activation function. In this case, the activation function works like this: if the weighted sum of input variables exceeds a certain threshold, it will output 1, else 0.

$$g(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

Why is bias added to a neural network?

Bias (**Wo**) is similar to intercept in linear regression It helps improve the accuracy of prediction by shifting the decision boundary along Y-axis. For example, in the image shown below, had the slope emerged from the origin, the error would have been higher than the error after adding the intercept to the slope.



In Neural Networks, **Bias** helps in shifting the decision boundary to achieve better predictions.

Let us consider the below variables and have a glance on how it works within a perceptron

Monthly Salary	Credit Score	Credit History (Years)	Loan Application Status
X1	**X2**	**X3**	**Y**

"The **feature vectors/independent variables** are used as inputs and then **weights (Random numerical values)** get assigned automatically by python packages (there are various weight initialization techniques which is a hyper-parameter), the dot product of weights and feature vectors and its summation ('**Z'**) are passed through the neuron.

The mathematical representation of the same is,

$$\sum_{i=1}^{p} w_i x_i$$

To calculate the values of **Z** for the above example, we do it by,

Z = [Monthly Salary value (Weight 1) + Credit Score value (Weight 2) + Credit History value (Weight 3)]

A **bias** value is added to this value of **Z** and is passed through an **activation function** in the neuron to get the output if the loan application is accepted or rejected**."(The role of an activation function is to make a non-linear transformation of the input before sending it to the subsequent layers to derive the output),** there are different types of activation functions viz. Sigmoid, Relu, Softmax.

You will later be getting a clear and detailed understanding of various techniques about weight initialization, along with various Activation functions and their role in Neural Networks.

MULTI-LAYERED NEURAL NETWORK (ARTIFICIAL NEURAL NETWORKS).

"Multi-layered Neural Network"

A Multi-layered Neural Network is a class of Feedforward Artificial Neural Network which consists of minimum of three layers i.e. Input layer, Hidden layer(s), Output layer.

Each of these layers are comprised of neurons, except for the neurons in the input layer, all the other neurons uses a non-linear activation function.

Each neuron within a layer is connected to all the other neurons of its next layer to form a fully connected neural network, for training this Network uses a supervised learning technique called 'Backpropagation'.

Although a Simple Perceptron is smart and can solve Binary Classification problem statements, it works best when data is linearly separable, and this is a major drawback of a single neuron network, therefore, to solve linearly non-separable data we need a multi-layered neural network, apart from solving binary classification use-cases it can also be used to solve multi-class classification use cases as well. It can comfortably work well with large datasets.

Multi-layered neural networks can be used to solve both Regression and Classification problem statements. These neural networks are more robust than a simple perceptron and can deliver quick and reliable predictions/output.

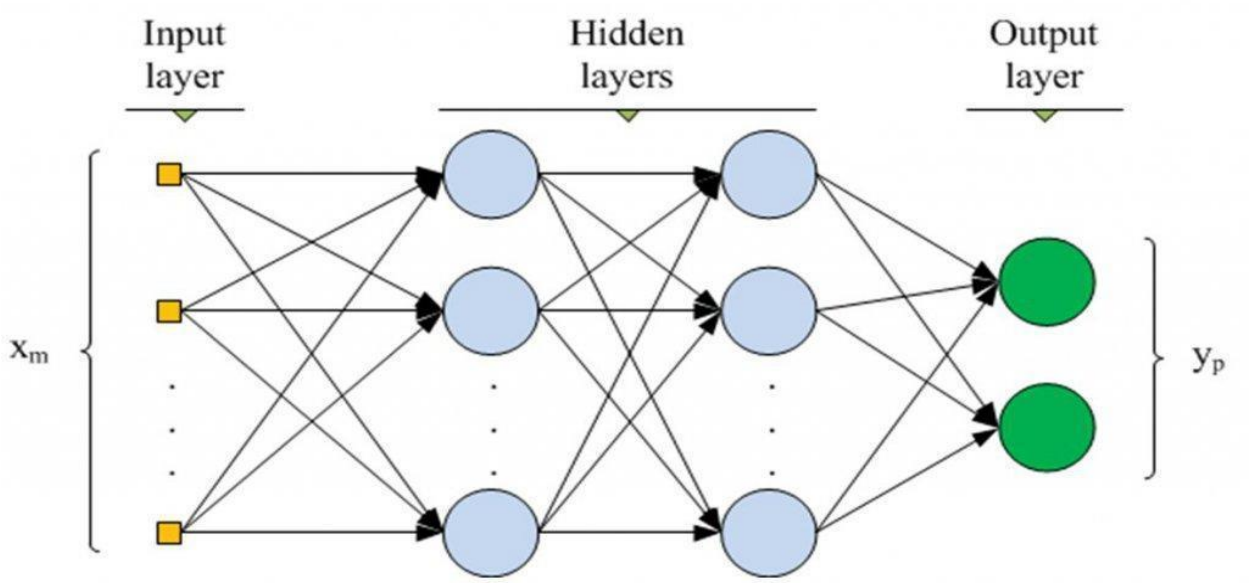
Though there are countless advantages of Multi-Layered Neural Networks in the field of Deep Learning it is computationally expensive and time-consuming if implemented with traditional and low configuration computers, it works well with a high configuration computer and GPU(s).

A multilayered neural network comprises a chain of interconnected neurons which creates the neural architecture. Each Neuron is connected to other neurons of its next layer with certain coefficients and input is passed on from one layer neurons to the neurons of the subsequent layers to return an output.

In simple words, it is a connection of numerous artificial neurons which are connected to the neurons of its next layer to form a fully connected neural network architecture that passes the input from one layer to the subsequent layers to return an output.

Few use cases of Neural Networks are Face Identification, pattern recognitions, Signal Classifications, Object Recognition, Recognition of speech, text, handwriting, Medical Diagnosis, Social network filtering, E-Mail Spam Classification.

Below is a basic intuition of how a multilayered neural network appears like,



- The **input layer** consists of neurons and the number of these neurons is equal to the number of features in the feature vector.
- The total number of **Hidden layers** and the count of neurons in each of these hidden layers are reliant on the input data as each use case is unique, ultimately the number of hidden layers and the neurons within these layers are hyper-parameters.
- In the **Output layer,** the solution/prediction is returned.

Multilayered neural networks are preferred when the number of features is large, thus it is being used to work on complex use cases like images, text data, etc. The two commonly used processes of neural networks are:

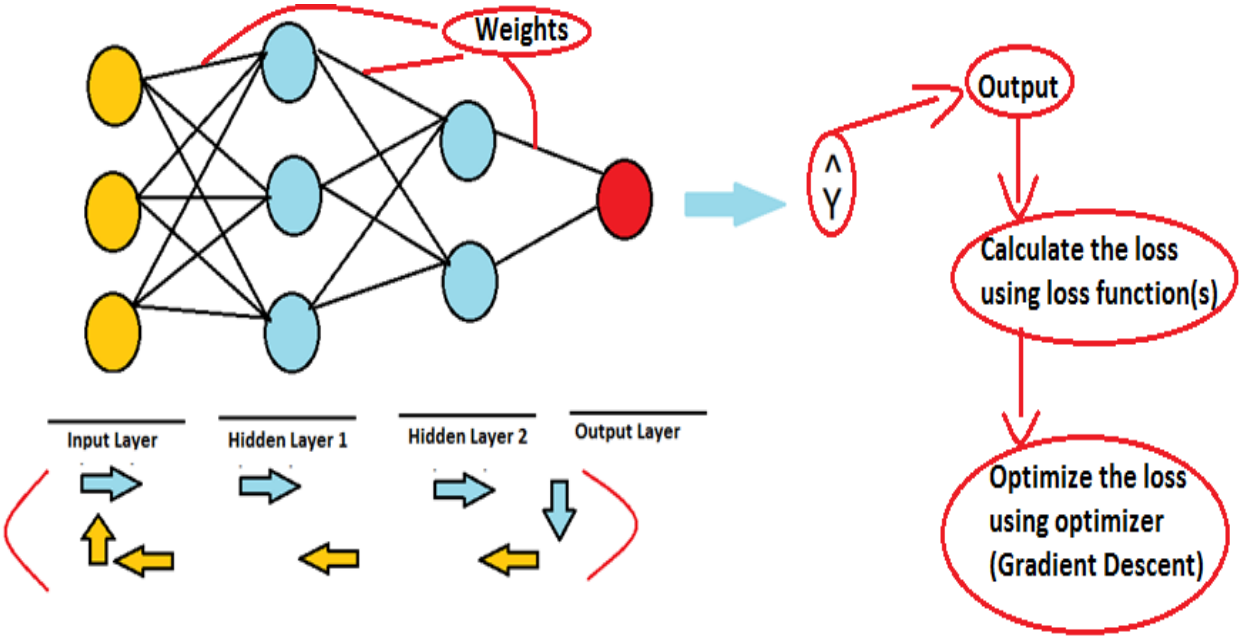
1. **Artificial Neural Network:** In this network, the information flows in a forwarding direction, i.e., from the input node to the output node, and then back-propagates to reduce the loss.
2. **Recurrent (or Feedback) Neural Network:** In this network, the information flows from the output neuron back to the previous layer as well.

Later by the time you complete this chapter, you will understand above mentioned neural networks more in detail.

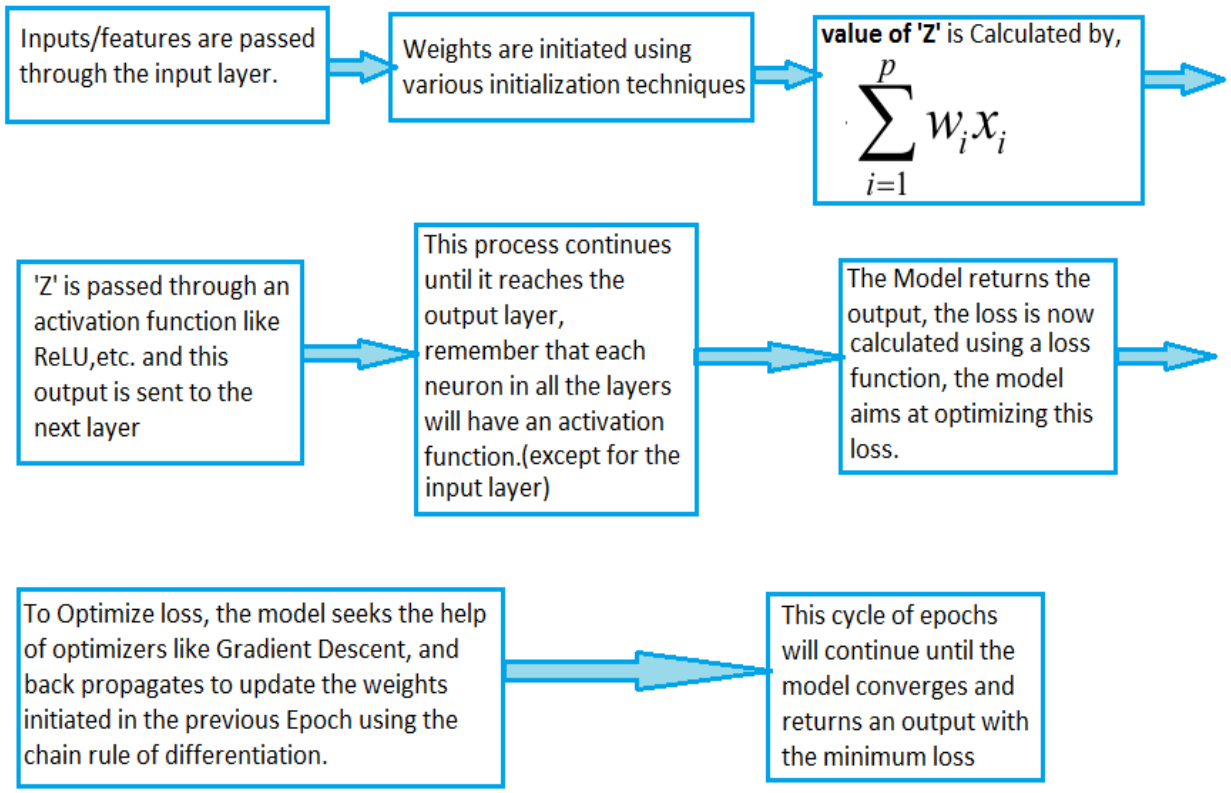
Now that you have the basic intuition of how the architecture of a Multilayered Artificial Neural Network is constructed, let us drill down a bit more to understand the functioning of the neural network i.e.

- What do you feed into the input layer?
- How are the values of a neuron computed?
- How are the weights initialized and updated with back-propagation?
- Activation Functions and various types of activation functions along with their uses in a neural network,
- Loss Functions, and Optimizers to optimize the model.

Let us have a glance at the algorithm in a nutshell.



Below are the details in a chart,

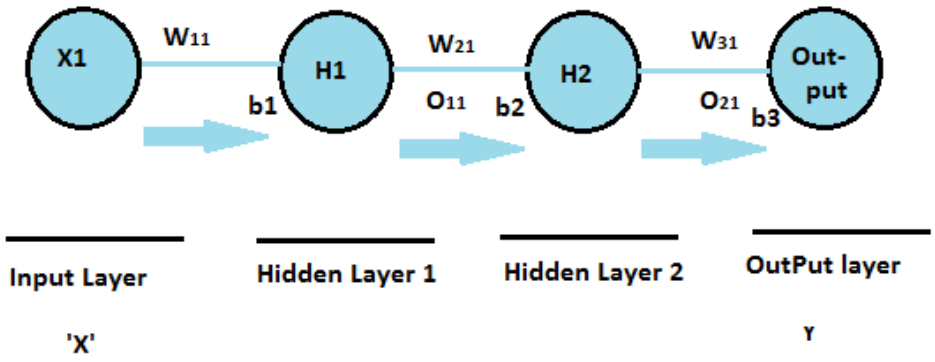


What do you feed into the input layer?

We feed the input layer with the features in the feature vector, each neuron of an input layer is the value of each feature from the feature vector. The total number of neurons is based on the number of features you are using to train the model.

How are the values of a neuron computed?

For a better understanding of calculations let us pick only the first neurons in each layer,



Where,

X1 = Input, **H1** = Hidden neuron 1, **H2** = Hidden Neuron 2, **W** = Weight, **O** = Output, **b** = Bias

Calculation:

X_1 = Value of the input Feature,

W_{11} = Weight pertaining to X1,

$H_1 = [b_1 + (X_1 * W_{11})]$

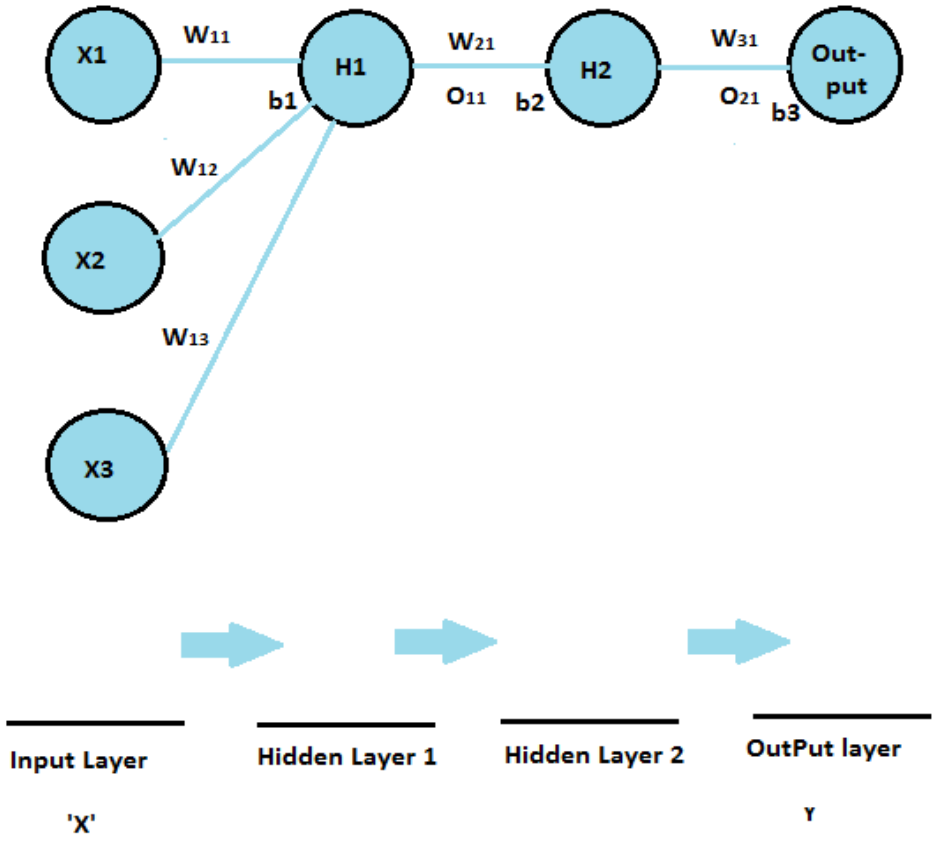
O_{11} = Output after H1 is passed through an activation Function,

W_{21} = Weight of H_1

$H_2 = [b_2 + (O_{11} * W_{21})]$

Similarly, the computation happens for the subsequent layers in all the neurons until the output is returned, this is how the data is passed from the input layer to the output layer, and however there would be a slight change in computation when you have multiple neurons in each layer,

Let us understand the computation when you have three inputs,



Where, **X1** = Input, **H1** = Hidden neuron 1, **H2** = Hidden Neuron 2, **W** = Weight, **O** = Output, **b** = Bias.

Calculation:

Reiterating what you have learned in simple perceptron.

$$y = g\left(w_0 + \sum_{i=1}^{p} w_i x_i \right)$$

H1 = [b1 + ((X_1 * W_{11}) + (X2 * W12) + (X3 * W13))]

O_{11} = g * (H1) Where, **g** = Activation function.

A similar calculation is done for all the other subsequent neurons in all the layers.

In practice, based on the use case and input data you may have a large number of hidden layers and neurons. However, Python libraries are experts in handling these calculations. Thus, you need not worry about manually computing the values for each of the neurons in the network. While implementing multilayered neural networks with the help of Python you will only need to specify the total number of input features, the number of hidden layers and neurons in each of these hidden layers, the type of activation function required for the neurons in each layer.

WEIGHTS INITIALIZATION

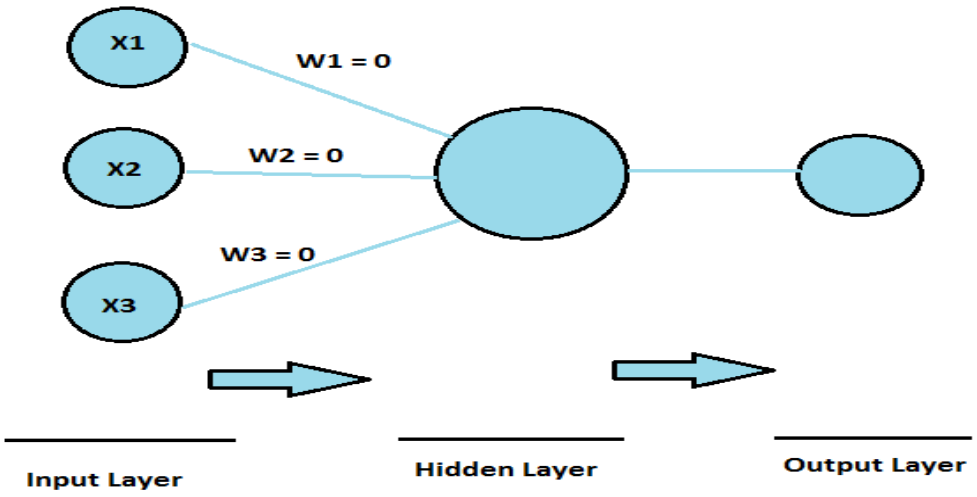
How are the weights initialized?

So far, you have studied that weights are used for computation in a neural network, but couldn't recognize the significance of weights, why weights are used and how the weights are initiated.

What are weights?

Weights are numerical values associated with features that state the importance of each feature in predicting the output.

Based on the total number of neurons in a network you will have the total count of weights, and the values of these weights need to vary i.e. shall not be the same, let us see what will happen to the output if the weights are the same and do not vary.

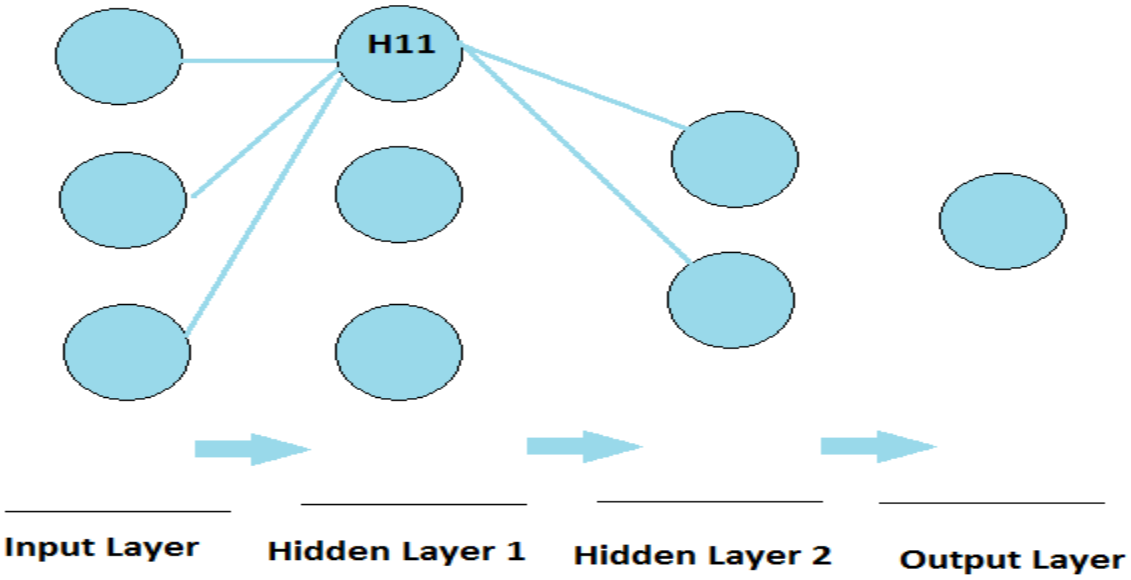


Now, if you run the computation, the dot product of the input features and weights would be Zero, thus obligating the output to return a **0**, hence it is important to have different values of weights for each of the neuron/connections to ensure an optimum output

There are various weight initialization techniques. Let us understand the below techniques,

- Xavier/Glorot Normal
- Xavier/Glorot Uniform
- He Uniform
- He Normal

Before going further you need to understand what is the **number of inputs and number of outputs,** with the help of the below architecture,



For **H11** Neuron, Number of inputs = 3, Number of outputs = 2, because 3 connections are coming in from its preceding layer and 2 connections going out to the succeeding layer, similarly you can calculate for the remaining neurons in a network.

Xavier/Glorot Normal:

XAVIER/GLOROT NORMAL INITIALIZATION

$$W_{ij} \sim N(0,\sigma)$$

Where,

$$\sigma = \sqrt{\frac{2}{\text{No. of Inputs + No.of Outputs}}}$$

In this technique weights follow a normal Distribution, with **Mean** as **0** and standard deviation **σ** of **"Square root (2 / Number of inputs + Number of outputs)"**

Xavier/Glorot Uniform:

XAVIER/GLOROT UNIFORM INITIALIZATION

$$W_{ij} \sim u\left[\frac{-6}{\sqrt{\text{No. of inputs + No. of Outputs}}}, \frac{6}{\sqrt{\text{No. of inputs + No. of Outputs}}}\right]$$

$$a \quad , \quad b$$

Here, weights will follow a uniform distribution which ranges between (a, b)

He- Uniform weights Initializer:

HE - UNIFORM INITIALIZATION

$$W_{ij} \sim u\left[-\sqrt{\frac{6}{\text{No. of Inputs}}}, \sqrt{\frac{6}{\text{No. of Inputs}}}\right]$$

$$a \quad , \quad b$$

Here, weights follow a uniform distribution between the range of (a, b).

He- Normal Initializer:

He - Normal Initializer

$$W_{ij} \sim N(0, \sigma)$$

Where,

$$\sigma = \sqrt{\frac{2}{\text{No. of Inputs}}}$$

With He-Normal initializer the weights initiated will follow a Normal Distribution, with **mean** = 0 and **σ** = Square Root (2 / Number of inputs).

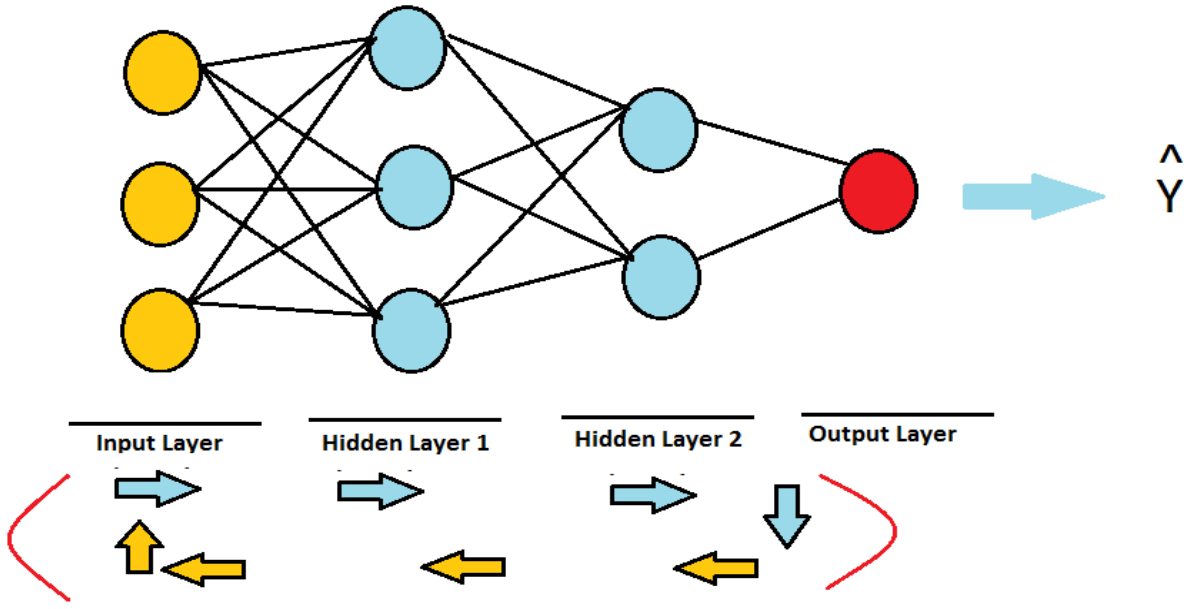
BACKPROPAGATION & CHAIN RULE OF DIFFERENTIATION:

"Back propagation"

It is used widely for training neural networks, this technique tunes the weights layer by layer starting from the final layer to the initial layer in a backward direction based on the error rate from the output, thus proper tuning of the weights will lead to an improved model as it reduces the error rate and generalizes the model.

Now that the weights are initiated you need to know more about the weights, if you could remember the architecture of a multilayered neural network, it consists of an input layer, hidden layers, and output layer. The data is passed into the network from the input layer and then weights are initiated, later the data is transferred from one layer to another layer to get the output.

So far, you have studied the flow of data from the input layer to the output layer which is a forward-pass network, now you will learn about Backpropagation, Back-propagation in a neural network is used to toss more optimum and efficient results. It is the essence of a neural network, it helps in tuning the weights and reducing the error rate/loss. During the process the weights get tuned/updated in a reverse flow i.e. it starts from the output layer to the weights initiated by the input layer.



The flow in the above image, the weights started initiating from input layer to the output layer then back-propagated from the output layer to the input layer weights, this iteration is called an **Epoch**, depending on the model the network may have 'n' number of epochs to tune the weights, it is a hyper-parameter and this chain of updating the weights will continue until we reach convergence (you will understand about it along with Gradient Descent Optimizer) For now, understand that reaching convergence means optimized/reduced the error rate from the output $\hat{Y}$.

In simple terms, **Error = $(Y - \hat{Y})$, where Y = Actual value and $\hat{Y}$ = Predicted value.**

Backpropagation aims to minimize the error/loss by updating/tuning the weights, minimizing the loss means improving the accuracy of the prediction/output, measuring the loss we use Loss functions, and to reduce the loss we use an optimizer like Gradient Descent optimizer.

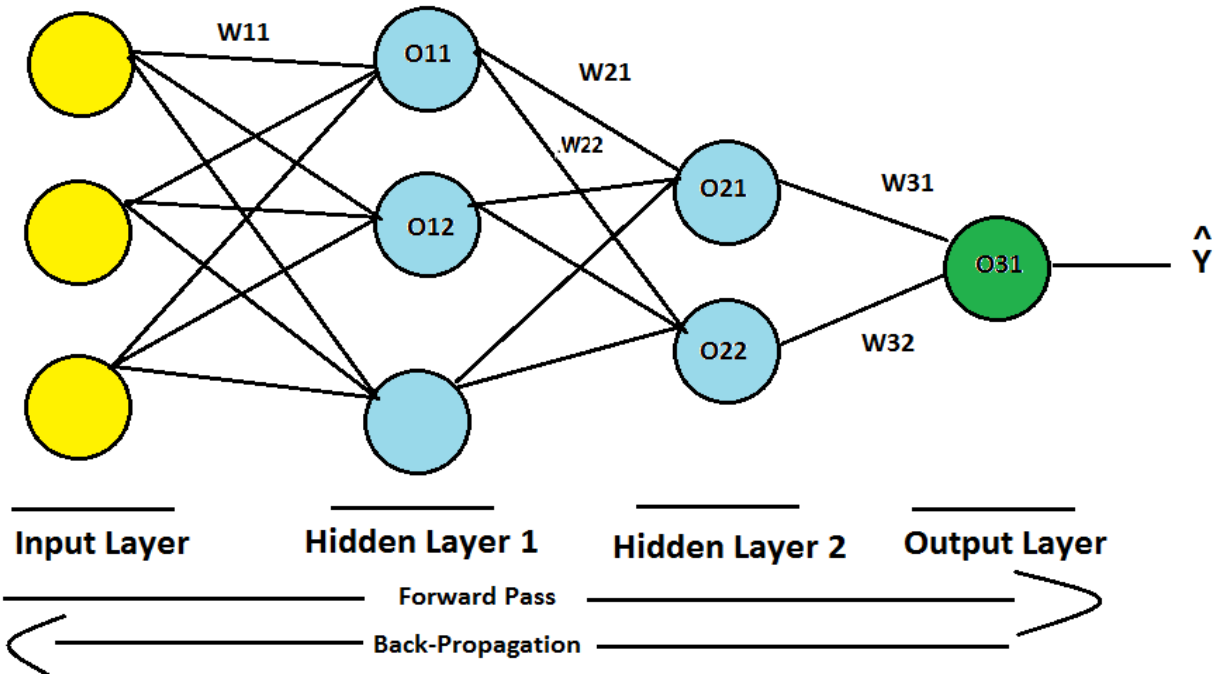
Backpropagation uses the **'Chain Rule of Differentiation** to tune the weights, below is the mathematical concept of the chain rule of differentiation in simple terms,

Chain Rule of Differentiation

Multiply $\frac{dy}{dx}$ with $\frac{dz}{dz}$ we get,

$$\left[\frac{dy}{dx} \times \frac{dz}{dz}\right] = \left[\frac{dy}{dz} \times \frac{dz}{dx}\right] = \frac{dy}{dx}$$

Backpropagation uses a similar kind of calculation to tune the weights and reach convergence. You will understand more about the chain rule in back-propagation with the help of the below image.



The algorithm will start by tuning the weights for W32 and W31, then the weights of its preceding layers will be tuned one after the other and once it reaches the initial layer, the process of forwarding pass will continue again, this chain of forward-pass and then tuning the weights with back-propagation would only stop upon the model convergence, i.e. the model shall have least/optimal error rate.

Let us have a glance at how the new weight for W31 is calculated,

$$W31_{New} = W31_{Old} - \eta \frac{dl}{dW31_{Old}}$$

$$= \left[\frac{dl}{dW31_{Old}} = \frac{dl}{dO31} \times \frac{dO31}{dW31_{Old}} \right]$$

Where,

η = Learning Rate

$\frac{dl}{dW31_{Old}}$ = Slope of the weight

You can similarly calculate the updated weight for **W32,** now let us have a look at how the weight for W21 is updated,

$$W_{21New} = \left[\frac{dl}{dO_{31}} \times \frac{dO_{31}}{dO_{21}} \times \frac{dO_{21}}{dW_{21Old}}\right] + \left[\frac{dl}{dO_{31}} \times \frac{dO_{31}}{dO_{22}} \times \frac{dO_{22}}{dW_{22Old}}\right]$$

Similarly, the remaining weights in the neural network are updated. In simple terms the formula for new/updated weights is,

$$Weight_{New} = w_{Old} - \eta \frac{dl}{dw_{Old}}$$

Where, η is the learning rate

Learning rate is a hyper-parameter used in optimizing the model, you will study more about it later when you deal with Gradient Descent optimizer.

LOSS FUNCTION AND VARIOUS TYPES:

"Loss Function"

A loss function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "Loss" associated with the event. The aim of any model is to minimize the loss. In statistical terms, loss is the difference between the actuals and the predictions.

The objective of an optimizer is to minimize the loss produced by a model, a higher loss is bad for any model, and the model shall produce minimum loss to generalize itself. Higher loss implies a higher deviation of the prediction from the actuals, thus the least loss is always optimal for the model. Gradient Descent algorithm is used while back-propagating the weights to minimize the loss in an artificial neural network model. (Going Forward, You will understand more about it).

But, how do you compute Loss?

Loss can be computed with the help of loss functions based on the type of use-case, i.e. regression or classification.

For regression, you may use **Squared Loss (L2 Loss), Absolute loss, Pseudo-Huber loss,** and for classification, you may use **Cross-Entropy and binary cross-entropy loss functions.**

Squared Loss (L2 Loss) function:

It is the sum of all the squared differences between the actual and the predicted value.

$$\text{Squared Loss /L2 Loss} = \sum_{i=1}^{n} \left(Y_{actual} - Y_{Predicted} \right)^2$$

Absolute Loss function:

It is the sum of all the absolute differences between the actual and the predicted value.

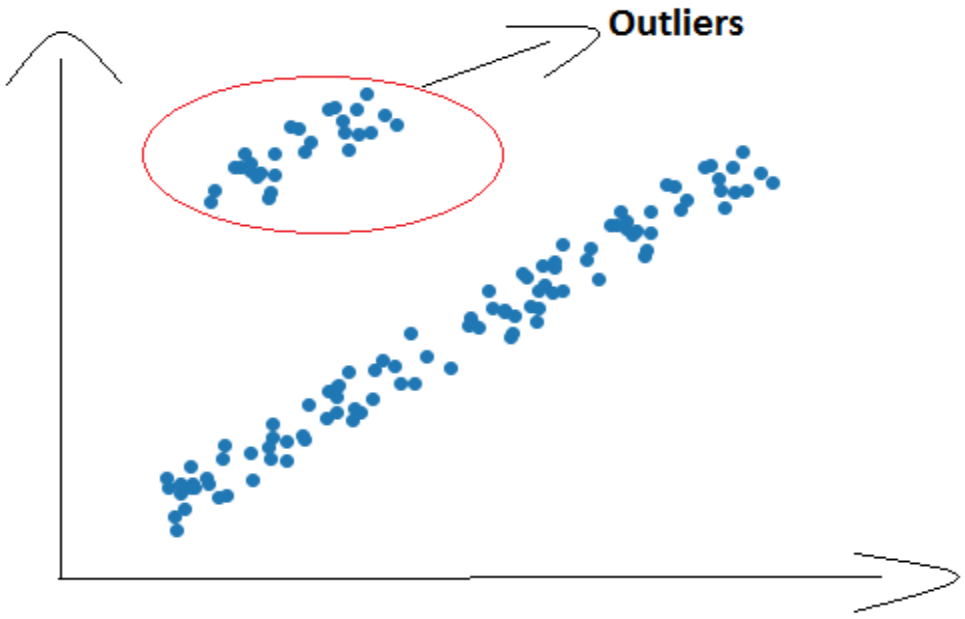
$$\text{Absolute Loss} = \sum_{i=1}^{n} \left| Y_{actual} - Y_{predicted} \right|$$

What if there are outliers in the model?

Generally, the L2 Loss function will try to capture the outliers, and the Absolute Loss Function is not affected by the outliers, however, if you try to use the L2 Loss function when outliers are prevailing it would affect the performance because it calculates the square of the differences which may result in a much higher value of the loss, thus if you want to use L2 Loss function try removing the outliers before using it.

Pseudo-Huber Loss Function:

By now you might have understood that outliers may hamper the performance and it is not always practically feasible to remove the outliers from the model, but the question is, how do we handle situations with outliers, is there any better Loss Function to use?



Pseudo – Huber Loss function is the solution, it is Robust compared to L2 and Absolute loss functions.

Pseudo-Huber Loss function

$$\begin{cases} (y - \hat{y})^2 ; & |y - \hat{y}| \leq \alpha \\ |y - \hat{y}| ; & \text{Otherwise} \end{cases}$$

Where, y = true value, $\hat{y}$ = predicted value, α is hyper-parameter.

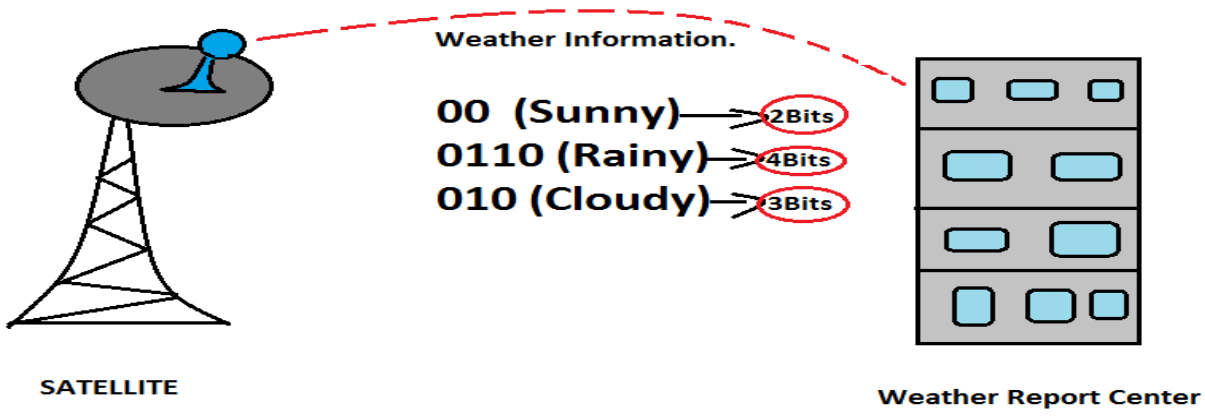
The Pseudo-Huber loss function states that it is general and adaptive to outliers.

- If the data point is having a relatively low error, it takes the squared (L2) loss,
- If the data point is an outlier it takes the absolute loss.

Thus pseudo-Huber loss function helps in reducing the effects of outliers on the model while still being differentiable.

Cross-entropy loss function:

This Loss function is mainly used for classification use-cases, let us understand this particular loss function with a crude example,



From the above image, assume that the satellite is transmitting the weather information to the weather report center, the information is transmitted in **BITS,** now from the above image we can calculate the Entropy, Entropy is the average number of BITS required to convey the

information. In our example **Entropy** = (2+4+3)/3 = **3. (Here, you can refer to entropy as the actual value)**

Imagine that the satellite is wrongly transmitting 5 BITS instead of 3 BITS due to some technical issue, this is Cross-Entropy. I.e. **Cross-entropy = 5, (Here, you can refer to Cross-entropy as the predicted value)**

The aim is to ensure that the difference between Entropy and Cross-entropy is minimum.

"The cross-entropy loss function is also called as Logarithmic loss or Log loss function. This function compares the probability of the predicted class to the actual class desired output and the loss is calculated based on its deviation from the expected value, a perfect model will have cross-entropy loss as 0."

$$\text{Cross-Entropy Loss} = -\sum_{i=1}^{n} y_i \log(p_i)$$

Where, Yi = Actual value, Pi = probability of i'th class in the output layer activation function.

If the use-case is Binary Classification, it is termed as **Binary Cross entropy loss function**. Based on the use case you can select the appropriate ones.

GRADIENT DESCENT OPTIMIZER:

'Gradient Descent Algorithm.'

Gradient Descent is a first order iterative optimization algorithm for finding the global minimum of a differentiable function. It mainly helps in optimizing the loss occurring from an artificial neural network.

The goal of the backpropagation algorithm is to optimize the weights associated with neurons so that the network can learn to predict the output more accurately. Once the predicted value is computed, it propagates back layer by layer and re-calculates weights associated with each neuron. In simple words, it tries to bring the predicted value close to the actual value. The

backpropagation algorithm optimizes the network performance using a loss function. This loss function is minimized using an iterative sequence of steps called the gradient descent algorithm.

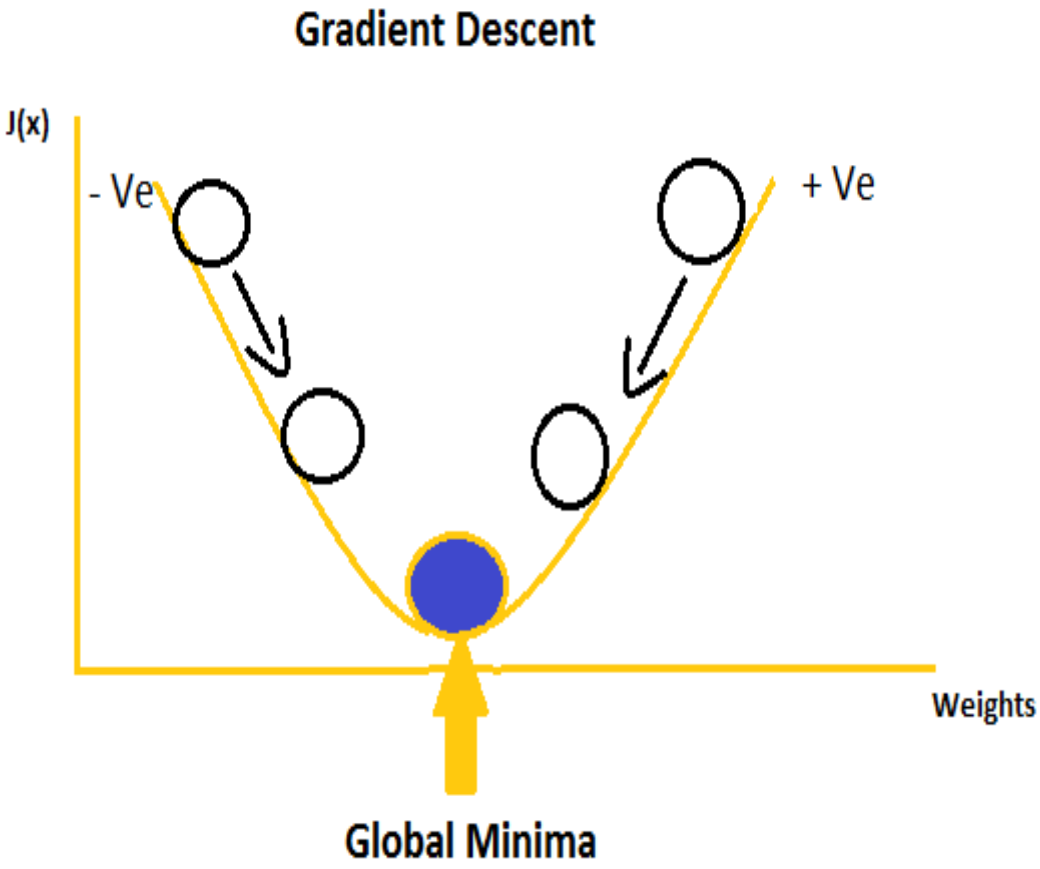
Artificial Neural Network uses Gradient Descent optimizer to reduce the loss. As discussed earlier, higher loss means higher deviation of our prediction from the actuals, thus loss needs to be reduced to the minimum and it is reduced by updating the weights through backpropagation, but the question is how do we know that the updated weights are optimum for the model?

We can solve this question with the help of a gradient descent optimizer,

The loss function is convex, it is identified by a U-shaped steep curve with the global minimum at the bottommost position. A great property of the convex function is that it is guaranteed to provide the lowest value when differentiated at zero. Think of a ball rolling down the U-shaped curve. It will take a few rounds of rolling (up and down) to slow down and it settles at the bottom. That bottom point is the minimum. And, that's where we want to go!

Let us replicate the same thought of a ball rolling down the U-shaped curve into the Gradient Descent algorithm, it is a process that aims at resampling the gradients of the model parameter in the opposite direction based on the weight until it reaches the global minima.

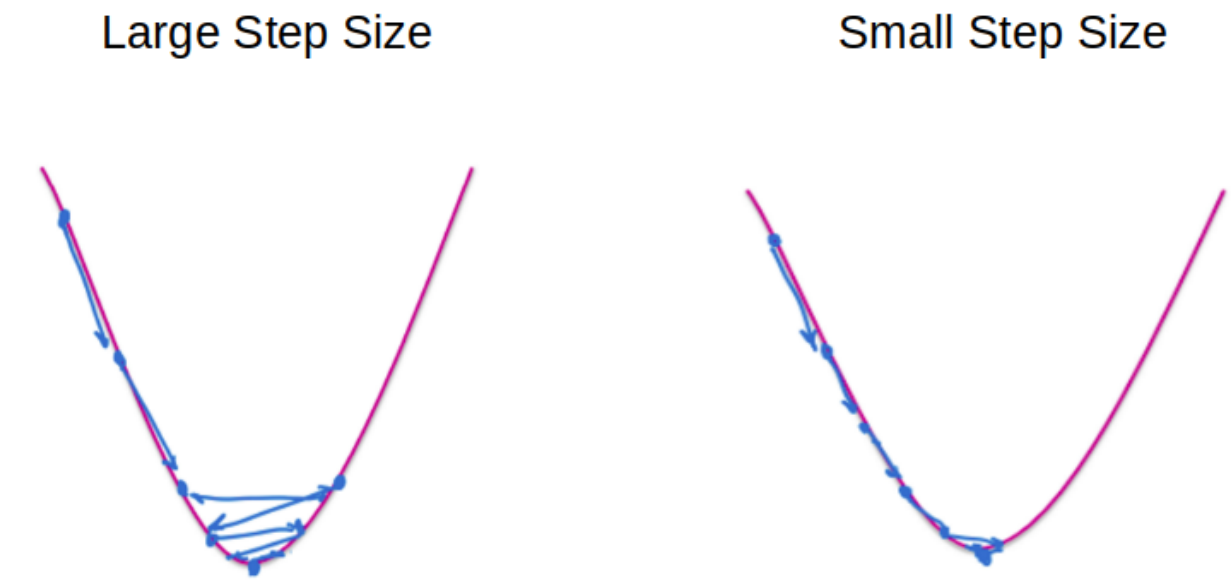
Here, **Gradient** means the sum of all first-order derivatives of variables in a function, **Parameter** refers to the weights, **Descent** refers to the steps taken by each iteration to reach the global minima. In addition to the above terminology, there is a hyper-parameter called the **Learning Rate,** which determines the pace of the model.



J(x) = Loss

In the above image, you can observe that the loss function forms a convex curve, the gradients are being resampled in the opposite direction for the model to converge i.e. reach the global minima. The model would not converge if they are resampled in the same direction as the aim is to reach the global minima.

The learning rate determines the speed of the gradient descent algorithm, If the learning rate is too high (large step-size) it may bypass the global minima and go beyond. If the learning rate is low (Small step-size) it will increase the computation time**,** hence it is important to select the appropriate learning rate. However, smaller steps are always better than a large step size.



ACTIVATION FUNCTIONS:

"Activation Function"

An activation function is used to define the output of a neuron with the given input. It also helps in determining the output of a neural network. In simple words, activation function is used to activate the neurons in a neural network.

You have studied the computation of weights and how the values of a neuron are computed in an artificial neural network along with back-propagation, now the time has arrived to understand more about the neural network functioning,

You will now deal with activation functions and might be you are having various questions running in your mind with regards to activation functions.

- What is an activation function?
- What are the different kinds of activation functions?
- How to select the appropriate activation function?

By the time you complete this concept, you will find the answers to all these questions.

What is an activation function?

An activation function is a transfer function that transforms the output of a neuron based on the given input.

Reiterating the below calculation,

$$y = g\left(w_0 + \sum_{i=1}^{p} w_i x_i \right)$$

Where,

Wo = Bias,

Wi = weights,

Xi = input variables.

The function **g()** is the activation function. In this case, the activation function works like this: if the weighted sum of input variables exceeds a certain threshold, it will output 1, else 0.

$$g(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

The above-quoted is a clear example of how an activation function transforms the given inputs of a node to return output in the form of 0 or 1.

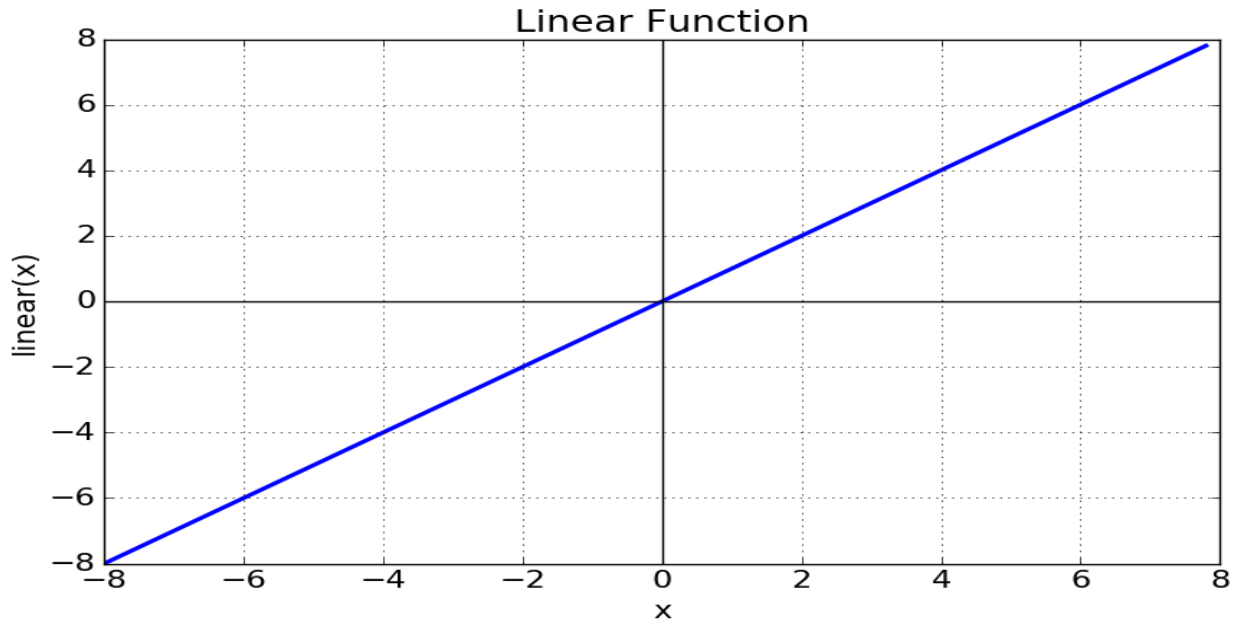
What are the various activation functions available?

There are various types of Activation functions available, based on the use-case and the required output you may select an appropriate activation function.

Activation functions are broadly classified into 2 types:

1. Linear Activation Functions.
2. Non- Linear Activation Functions.

Linear Activation Functions – The name signifies it is a linear Function and is in the form of **f(x) = ax +c**, i.e. it would have a straight line and the output will range between (–infinite, infinite). On the multilayered artificial neural networks, Linear Activation functions are not as useful as the non-linear activation functions.



Non-Linear activation functions: Non-Linear activation functions are classified based on their Curves and Range. Few of them are,

- Sigmoid,
- Tanh,
- ReLU (Rectified Linear units), Leaky ReLU,
- Swish,
- Softmax.

Sigmoid Activation Function: It is the same function that is used in the logistic regression algorithm, this function will have an 'S'- Shaped curve, which ranges between 0 to 1. This function can mostly be used in binary classification use cases where the desired output is either 0 or 1, especially in the output layer. It is not recommended to use the sigmoid activation function in the hidden layers of the network.

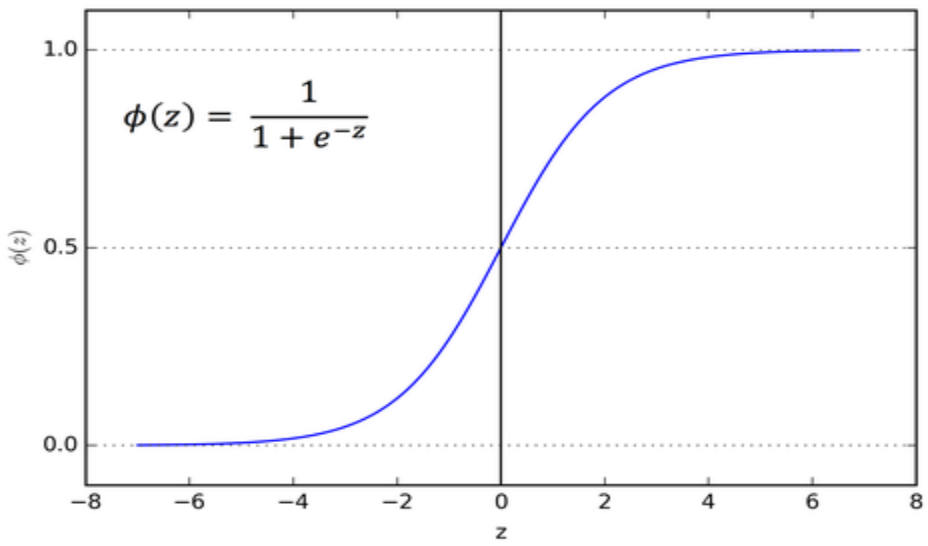
$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Where,

$$Z = \sum_{i=1}^{n} W_i X_i + b_i$$

The curve of a sigmoid function appears as below, based on the threshold of 0.5, the function will classify all the values above 0.5 as 1 and below 0.5 as 0, however, the threshold value can be changed if required.

$$\begin{cases} 0 & Z \geqslant 0.5 \\ 1 & Z < 0.5 \end{cases}$$

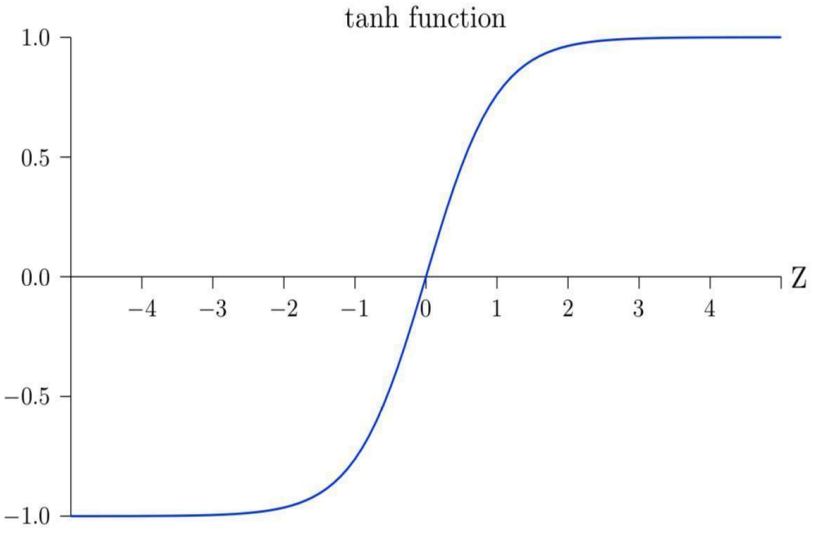


Though the Sigmoid Activation function is one of the prominent activation functions used in binary classification use cases, it may at times leads to vanishing gradient problem as the derivatives of sigmoid ranges between 0 & 0.25. (You will study more about the vanishing gradient problem in the next chapters), for now, understand that the vanishing gradients problem is a hurdle for the model to reach Global minimum and it will halt the model from converging.

Tan-h Activation Function:

Tanh (Hyperbolic tangent) is an alternative to sigmoid, it also possesses an 'S' shaped curve, However, in sigmoid, the values range between 0 & 1, but in tan-h activation function, the values range between -1 & +1. Thus, negative inputs to the Tan-h activation function will map to negative outputs. The Vanishing gradient problem still prevails with Tan-h as well.

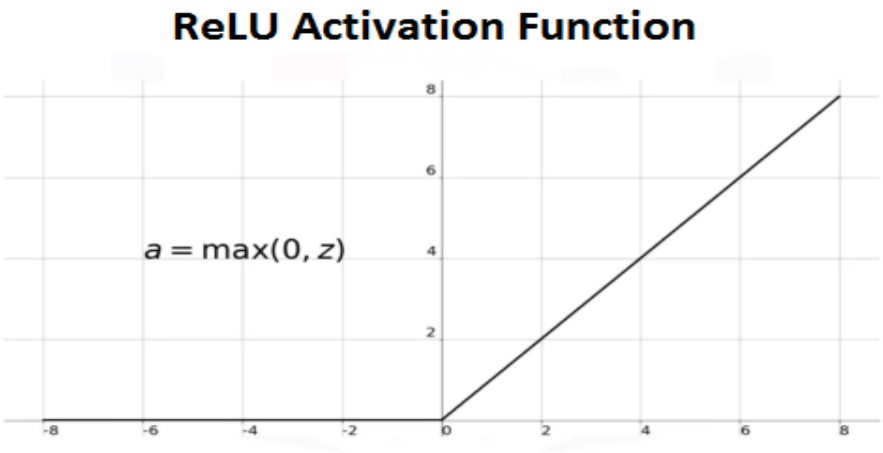
$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$



ReLU (Rectified Linear Units):

ReLU activation function is mainly used in the hidden layers of a multilayered artificial neural network, if the input is a positive value it will return exactly the input value as the output, in case if the input value is negative, it would simply return a 0 as output.

ReLU f(X) = Max(0,X)
i.e.
X , When X > 0
Else , 0



It is the most commonly used activation function, the best part is that it does not activate all the neurons in a network at the same time, as negative inputs to ReLU will transform the output to 0 it would not activate such neurons, thus it would make the computation more effective.

The ReLU activation function can be considered as a solution for the Vanishing Gradients problem.

Leaky ReLU Activation function:

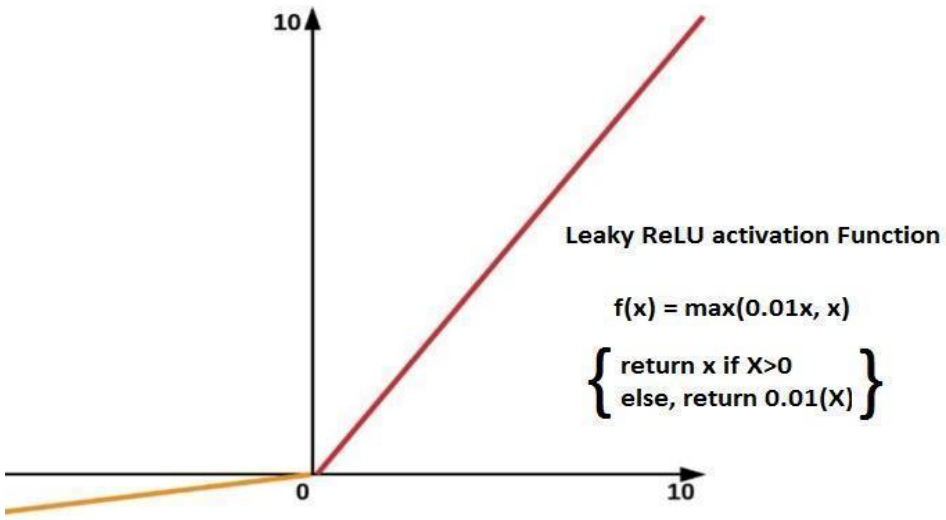
This activation function is an advanced version of the ReLU activation function, though ReLU activation function is an expert in handling the problem of vanishing gradients there are certain disadvantages of the ReLU activation function, mainly the model might encounter the problem of dying ReLU because ReLU returns zero in case of negative inputs, it will create a dead neuron as no computation will happen on these dead neurons with value as 0, hence leaky ReLU will add a small numerical value instead of 0 for negative values, say 0.01, etc. to avoid a dead neuron.

Below is the function for Leaky ReLU,

Leaky ReLU activation Function

$$f(x) = \max(0.01x, x)$$

$$\begin{cases} \text{return } x \text{ if } X>0 \\ \text{else, return } 0.01(X) \end{cases}$$



So far you have studied that,

- Sigmoid and Tan-H activation functions are responsible for the vanishing gradients problem,
- Researchers came up with ReLU to overcome the vanishing gradients problem, but ReLU may encounter the problem of dying ReLU,
- To overcome the problem of Dying ReLU we may try using Leaky ReLU, as Leaky ReLU uses a small numerical value like 0.01 instead of negative values it may again lead to the problem of vanishing gradients.

Swish Activation Function:

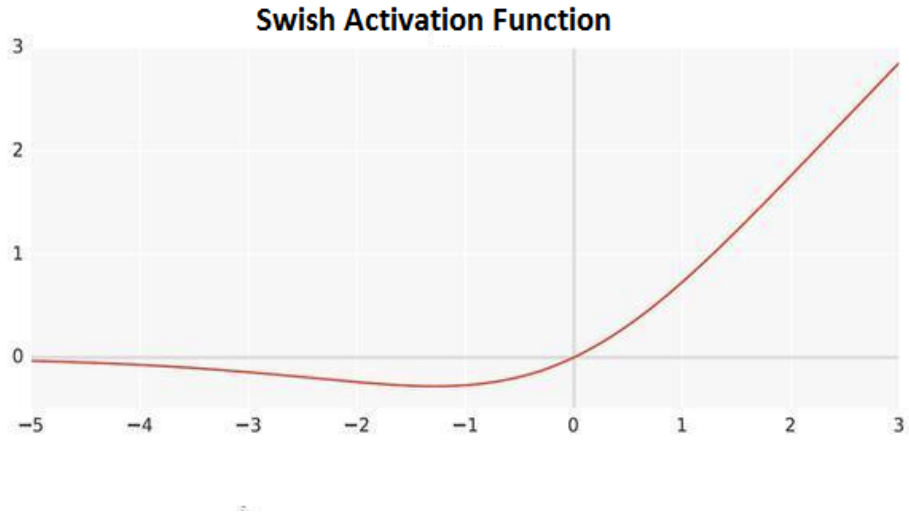
This activation is especially used with Recurrent Neural Networks - LSTM Models, it works best when the neural network is deep i.e. if there are a large number of layers in the network, and this activation function uses sigmoid activation function within it.

Swish Activation Function

$$y = X * \text{Sigmoid}(X)$$

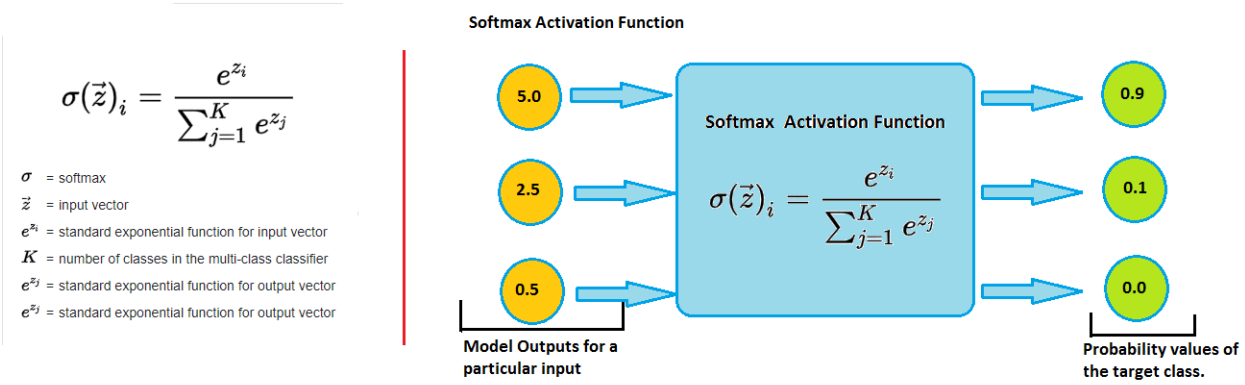
$$\text{Where, } X = W_i X_i + b$$

Swish is self-Gated, i.e. it requires only one single input. This Activation function can be used to replace ReLU as it outperforms ReLU. Swish is a Zero Centric Function.

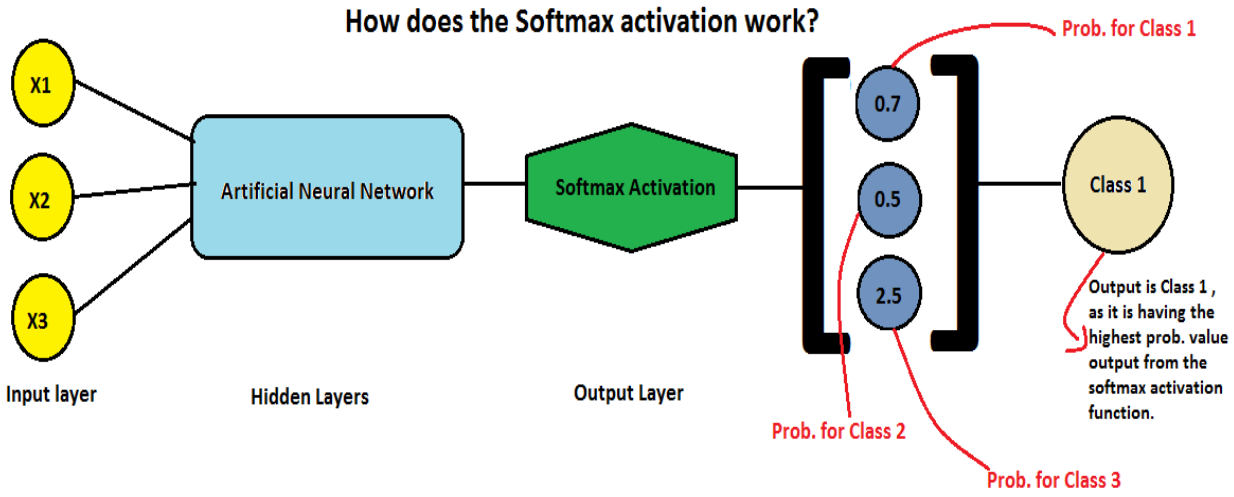


Softmax Activation Function:

The softmax activation function is mainly used in the output layer of the neural network, it works best on multinomial classification use-cases, this activation function calculates the probabilities of each target class over all possible target classes, and these probability values determine the output target class of the given input.



The sum of all the probability values is always 1, and the target class with the highest probability wins. Suppose you are solving a classification use-case with three classes in the dependent variable, you can observe in the below image that the input features are sent into the network through the input layer and the Softmax activation function in the output layer has returned the probability values of each of the three classes of the dependent variable, and the model has returned the final output as class 1 which is having the highest probability value.



Remember that the Sigmoid and Softmax activation functions are not recommended to be used in the hidden layers of the neural network, they would work best only when used in the output layer.

To summarize, you have studied the entire flow of artificial neural networks, i.e. how the features are fed into the model through the input layer, the computation of neurons in the hidden layers, various activation functions, computation of the predictions based on the activation function used in the output layer and the values returned by the output layer,

computation of loss, back-propagation and gradient descent optimizer to tune the weights to reduce the loss, now let us discuss the other forms of gradient descent optimizer namely, **Stochastic gradient** descent and **Mini-batch Stochastic Gradient Descent** algorithms.

STOCHASTIC GRADIENT DESCENT ALGORITHM:

A stochastic gradient descent algorithm is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or sub-differentiable). It can be regarded as a stochastic approximation of gradient descent optimization since it replaces the actual gradient (calculated from the entire data set) with an estimate thereof (calculated from a randomly selected subset of the data). Especially in high-dimensional optimization problems, this reduces the computational burden, achieving faster iterations in trade for a speedy convergence.

Suppose you have 'n' data points in the dataset to feed into the neural network model, once you feed the entire data the model will start initializing the weights and computes the values of the neurons in each layer respectively, and calculates the loss using a loss function and then back propagates to tune the weights to minimize the loss, for this it uses an optimizer (Gradient Descent).

The stochastic Gradient descent algorithm works similar to the Gradient Descent Algorithm, however, the major difference is that Gradient Descent Algorithm will try to train all the 'n' number of records in the dataset all at once, but the stochastic gradient descent algorithm will try to train one record at once, i.e. assume you are having 10 million records in the dataset, Gradient Descent algorithm will try to complete the entire computation of these 10 million records in a single Epoch, but Stochastic Gradient Descent will take 10 million iterations to complete 1 epoch.

If you could notice that 10 million records are a large number to handle, and you try using Gradient Descent, in case you are using a low configuration traditional machine to train the algorithm it is really difficult to complete the task, similarly if you try training the model with the help of stochastic gradient descent it will take 10 million epochs to return the output, both these algorithms are time killing and expensive, Thus **mini-batch Stochastic Gradient Descent algorithm** is a better option to use.

How does Mini-batch stochastic gradient descent algorithm differ from Gradient Descent and Stochastic Gradient Descent algorithms?

Unlike the Gradient Descent and Stochastic Gradient Descent algorithms, Mini-batch stochastic gradient descent algorithm will compute the data in batches, i.e. chunks, from the

above-quoted example of 10 million records in the dataset, with the help of a mini-batch gradient descent algorithm you can compute the data in chunks of 1 million each which will take 10 iterations to complete an epoch. This will reduce the stress and load of computation of large datasets into the model and helps in a speedy convergence.

The batch size can be selected with the help of the K-value and it can be varied based on the use case scenario for optimum results.

The weights calculated using Gradient Descent and Stochastic Gradient Descent are not the same values, however, they are approximately equal.

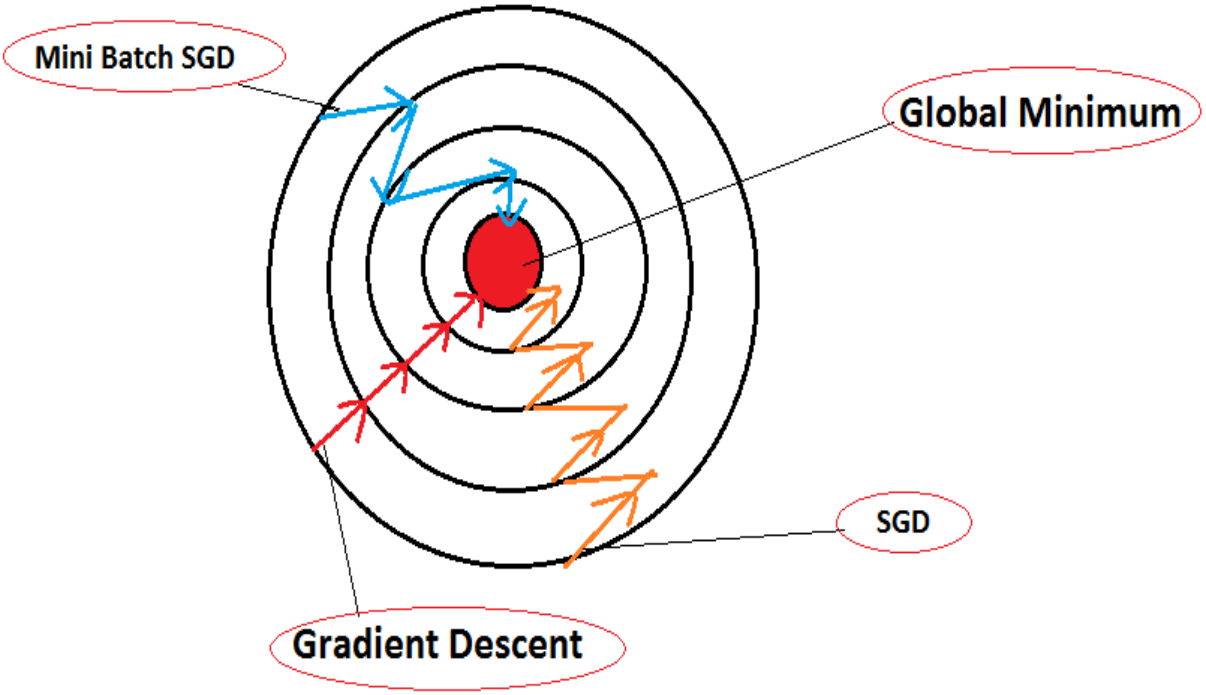
Gradient Descent	Stochastic Gradient Descent
$Weight_{New} = W_{Old} - \eta \frac{dl}{dW_{Old}}$	$Weight_{New} = W_{Old} - \eta \frac{dl}{dW_{Old}}$
Where, η is the learning rate	Where, η is the learning rate

$$\frac{dl}{dW_{Old}} \approx \frac{dl}{dW_{Old}}$$

(Gradient Descent) **(Stochastic Gradient Desent)**

The slight difference in the values is because Gradient Descent will train the entire data in one single epoch, whereas, Stochastic Gradient Descent will train the same data in 'n' number of epochs. Thus, stochastic gradient descent will form a zig-zag path to converge rather than converging smoothly like the Gradient Descent. This zig-zag path is the noise in the data,

Below, you can visualize the path on how they would reach the global minima in a sample counterplot.



The noise for Stochastic Gradient Descent is higher in comparison to Gradient Descent and Mini-Batch-Gradient Descent, Technically in most of the situations, Mini-Batch Gradient descent is the better option in comparison to the other two algorithms (Gradient Descent and SGD).

The reason behind the noise in Stochastic Gradient Descent and Mini-Batch Stochastic Gradient Descent is because the data is not computed as a whole at once, the convergence of mini-batch SGD and SGD is slow in comparison to Gradient Descent but, Mini-Batch Gradient Descent uses fewer resources resulting to low computational cost.

The goal of an optimizer is to reduce the loss resulting from the neural network model, now **is there any difference in the loss functions between Gradient Descent, Stochastic Gradient Descent, and Mini-batch Gradient Descent algorithms?**

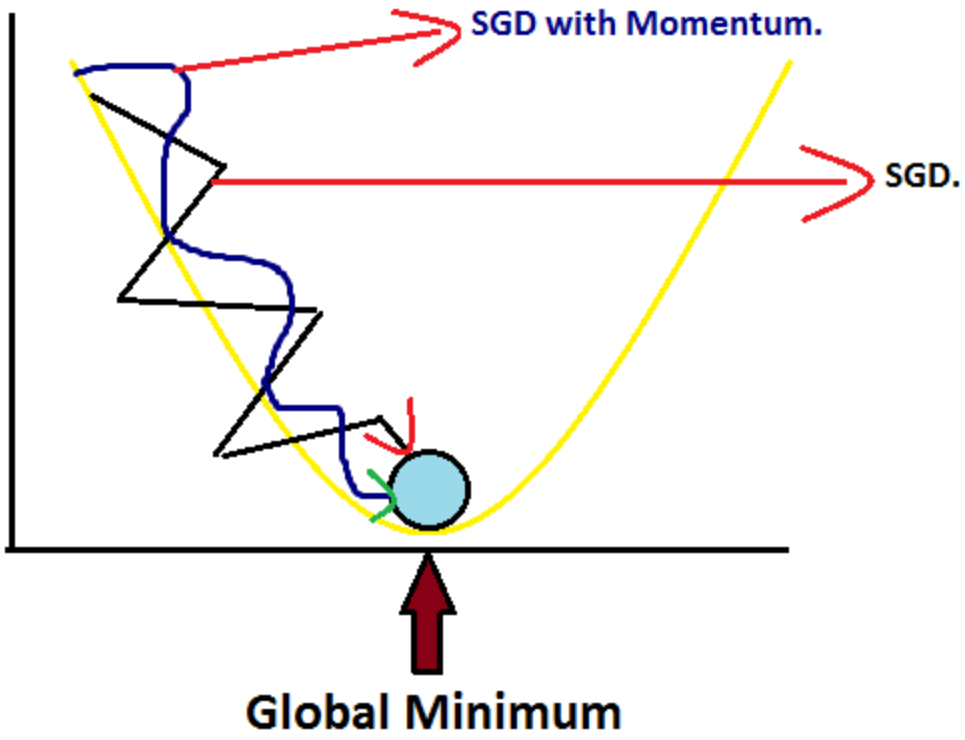
OPTIMIZER	Loss Function
Gradient Descent	$\sum_{i=1}^{n} (y - \hat{y})^2$
Stocastic Gradient Descent	$(y - \hat{y})^2$
Mini-Batch Stocastic Gradient Descent	$\sum_{i=1}^{K} (y - \hat{y})^2$

Where, K = Batch-size, say K=1000, the algorithm will send 1000 records to calculate $\hat{y}$ for each record.

Stochastic Gradient Descent with momentum:

Though SGD and Mini-Batch Gradient Descent are practically efficient, the noise is more compared to Gradient Descent, so, **is there any solution to reduce the Noise?**

The answer is **Stochastic Gradient Descent with momentum,** SGD with momentum helps in accelerating the gradients in the correct direction by taking longer steps thus, leading to a faster convergence, it typically smoothens the noisy zig-zag path to converge, resulting in reduced noise and accelerated convergence. You can visualize the path in the below image.



It is observed that the zig-zag path is smoothened, resulting in reduced noise and an accelerated convergence.

You have already studied the calculation for tuning weights, reiterating the calculation,

$$\text{Weight}_{\text{New}} = w_{\text{Old}} - \eta \frac{dl}{dw_{\text{Old}}}$$

Where, η is the learning rate

For SGD with momentum calculation, there is a small change,

$$w_t = w_{t-1} - \eta \frac{dl}{dw_{t-1}}$$

Where, η is the learning rate
t=iteration

The Bias also gets updated along with weights, for updating bias replace **W** with **b** in the above formula.

SGD with momentum uses **Exponential Weighted Average** to smoothen the curve.

Assume, having the below series of data,

Iteration	t1	t2	t3	t4	---------- tn.
Data	a1	a2	a3	a4	---------- an.

Let us begin by initiating a new variable '**V**'

Say, $V_{t1} = a1$,

$V_{t2} = \beta * V_{t1} + (1 - \beta) * a2$ {Calculation for Exponential weighted average}

Where, β is a hyper-parameter and it resembles the importance of the previous data.

Let us consider β value as 0.95 and substituting the same in the above equation,

$\Rightarrow V_{t2} = \beta * a1 + (1 - \beta) * a2$ {since, $V_{t1} = a1$}

$\Rightarrow 0.95 * a1 + (0.05) * a2$

Where**, 0.95 * a1** has more importance for previous data and **(0.05) *a2** has less importance for the next data,

If, you select $\beta = 0.1$ instead of 0.95 then,

$V_{t2} = \beta * a1 + (1 - \beta) * a2$

$\Rightarrow 0.1a1 + (0.9) * a2$

Where **0.1* a1** has less importance for previous data and **(0.9) * a2** has more importance for the next data,

Calculation of V_{t3},

$V_{t3} = \beta * V_{t2} + (1 - \beta) * a3$

Similarly, you can calculate for up to V_{tn}.

Upon implying this calculation to the weights updating formula i.e.

$$w_t = w_{t-1} - \eta \frac{dI}{dw_{t-1}}$$

Where, η is the learning rate
t=iteration

The formula now becomes,

$$W_t = W_{t-1} - \eta V_{dW}$$

Where, V_{dw} is the exponential weighted average.

During the calculation of Exponential Weighted Average, it is recommended to consider a β of 0.95 for optimum results.

ADAGRAD OPTIMIZER:

Adagrad stands for **Adaptive Gradient Descent Optimizer**, during gradient descent optimizer, you have studied that the Learning Rate in the algorithm decides the pace, and this learning rate is constantly applied to the 'n' iterations/epochs in the model for back-propagating the weights.

Reiterating the calculation of updating weights using Gradient Descent,

$$Weight_{New} = w_{Old} - \eta \frac{dI}{dw_{Old}}$$

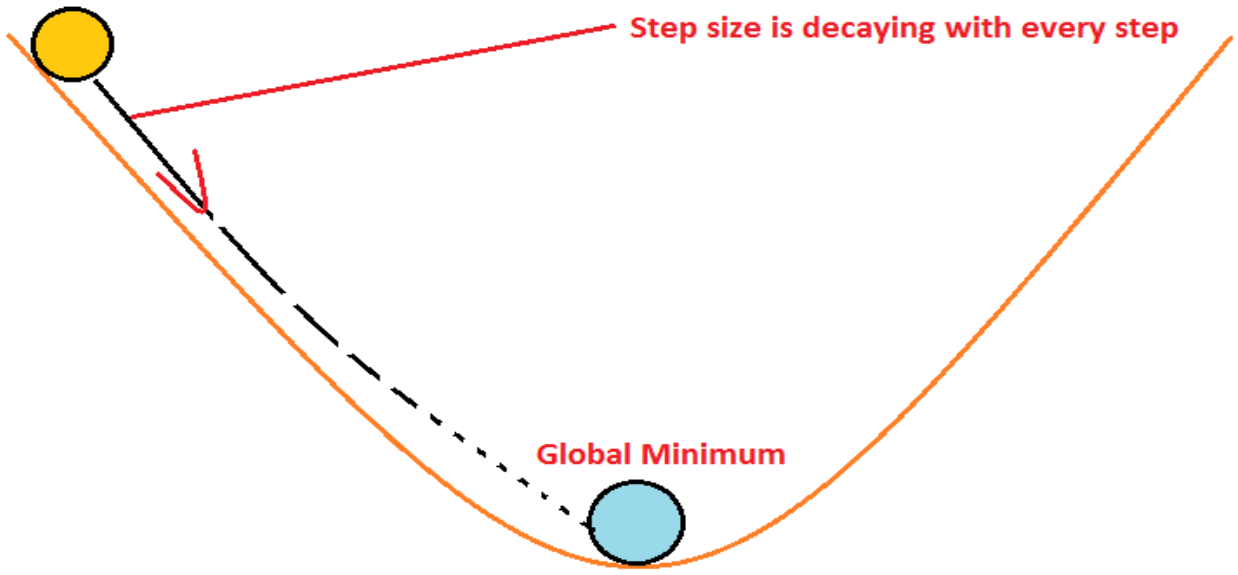
Where, η is the learning rate

As weights are being updated on every iteration/epoch, the formula now appears like,

$$w_t = w_{t-1} - \eta \frac{dl}{dw_{t-1}}$$

Where, η is the learning rate
t=iteration

Adagrad Instead of using a constant learning rate for each parameter base on iteration uses a different learning rate for each parameter base on iteration. This works best on sparse data, though the learning rate keeps changing it would decay before the model converges.



You can observe that initially the step size was large and it kept diminishing by the time it reached the global minimum, it is because the learning rate was large initially and it kept decaying with every step.

Let us have a look at the math behind it,

$$\text{Adagrad } W_t = W_{t-1} - \eta'_t \frac{dl}{dW_{t-1}}$$

$$\text{Where, } \eta'_t = \frac{\eta}{\sqrt{\alpha_t + \varepsilon}}$$, ε is a small numerical value to avoid 0 in the output.

$$\text{and, } \alpha_t = \sum_{i=1}^{K} \left(\frac{dl}{dW_t}\right)^2$$

As you are squaring the values in the α_t calculation, the value of α_t keeps increasing along with the iterations, and when the value of α_t increases it leads to a higher value in the denominator of the learning rate formula, thus reducing the learning rate along with iterations.

However, the disadvantage is that when you are training a very deep neural network, i.e. with a large number of layers, the value of α_t becomes so high and the learning rate becomes very small, which is an obstacle for the model to smoothly reach the global minimum.

The solution for this problem is **'RMS Prop'.**

RMSPROP OPTIMIZER:

RMSprop is a Gradient Descent-based algorithm, which balances the learning pace. To calculate the value of α_t in Adagrad you are training on all the epochs, i.e. there is no control of α_t value, and after a certain point the learning rate diminishes and the model will experience difficulties in converging.

$$\text{Adagrad Calculation of } \alpha_t = \sum_{i=1}^{K} \left(\frac{dl}{dW_t}\right)^2$$

This problem is mostly faced when the network is large and deep, since Adagrad is trained on all the epochs you might try to restrict it to control the value of **α_t** and overcome this problem.

$$\text{Adagrad calculation of learning rate } \eta'_t = \frac{\eta}{\sqrt{\alpha_t + \varepsilon}}$$, ε is a small numerical value to avoid 0 in the output.

For RMSProp we change the calculation of this learning rate by using weighted average i.e. S_{dw} instead of α_t.

$$\text{RMSProp calculation of learning rate } \eta' = \frac{\eta}{\sqrt{S_{dw} + \varepsilon}}$$

$$\text{Where, } Sdw_t = \beta\, Sdw_{t-1} + (1-\beta)\left(\frac{dl}{dw}\right)^2 \text{ , and } \beta = 0.95 \text{ or } 0.9$$

Finally, RMSprop weight updation calculation is,

$$W_t = W_{t-1} - \eta' \frac{dl}{dW_{t-1}} \quad \text{and} \quad \eta' = \frac{\eta}{\sqrt{S_{dw} + \varepsilon}}$$

As you are applying **(1-β)** in the weighted average calculation i.e. in **Sdw** calculation, which is generally a low numeric value we can control and balance the model unlike in Adagrad the value of **αt** was uncontrollable and it went very high in case of deep neural networks.

ADAM OPTIMIZER:

So far you have learned RMSProp is the best adaptive optimizer, however, researchers came up with an even more powerful adaptive optimizer that is more efficient than any of the other

optimizers. You have read it correctly, ADAM (Adaptive moment estimation) combines both the properties of Momentum and RMSProp making it the most powerful and efficient adaptive optimizer.

During the concept of Momentum and RMSprop optimizers you have studied the calculations of V_{dw} and S_{dw} respectively**,** now let us see the math behind how ADAM optimizer uses them to update the weights and increase the efficiency of the model for accelerated and efficient convergence.

$$W_t = W_{t-1} - \frac{\eta * V_{dw}}{\sqrt{S_{dw} + \varepsilon}}$$

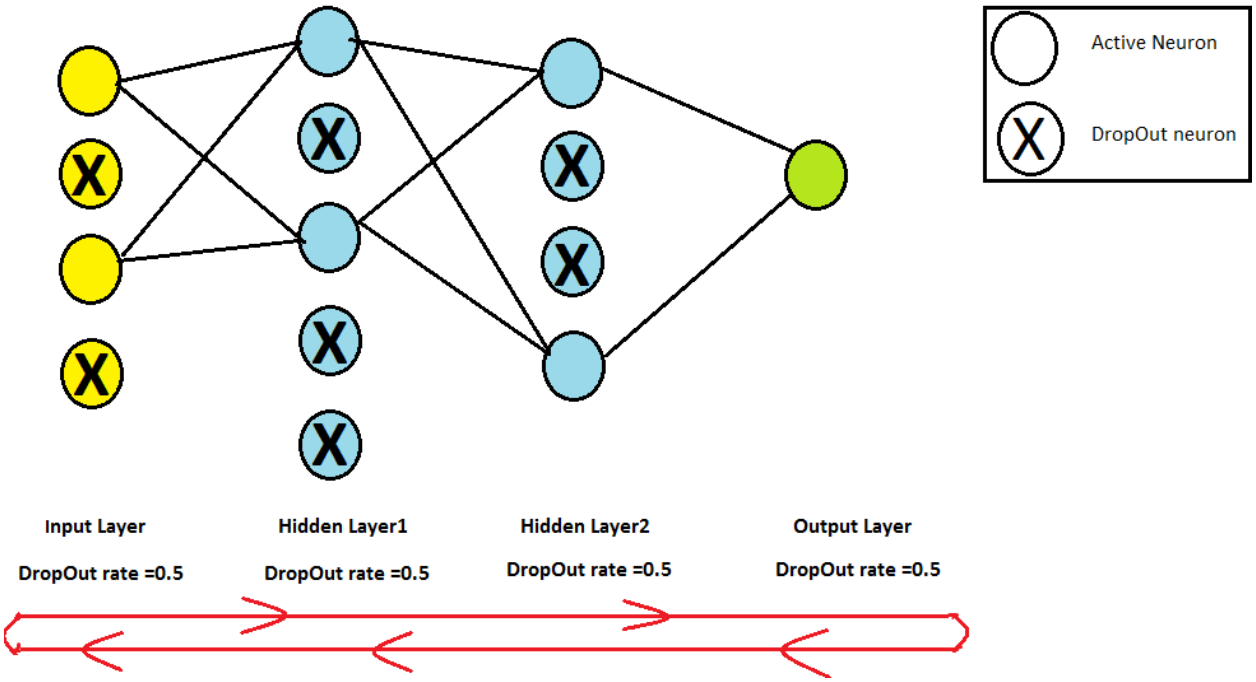
Where, η is the initial learning rate

In the above formula, you can observe that ADAM weights calculation comprises, both V_{dw} and S_{dw} within it, thus it is a combination of both smoothening and balancing the model to converge efficiently.

DROP_OUT IN NEURAL NETWORK:

To prevent the model from over-fitting you can apply a DropOut of layers to the model, The Dropout layer randomly sets input units to 0 with a frequency of **rate** at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by **1/(1 - rate)** such that the sum over all inputs is unchanged.

Drop-out uses a hyper-parameter i.e. **dropout rate** to drop the features. Say, you have applied a dropout rate of 0.5 to the neural network, the model would appear similar to below,



The model will simply avoid the computations in the DropOut Features even while tuning the weights during back-propagation. Only the active features weights will get updated.

In simple words, it is a process of deactivating few features from the model to avoid overfitting.

PRACTICAL IMPLEMENTATION OF ARTIFICIAL NEURAL NETWORKS:

Training a neural network with a traditional low configuration machine is complex, the training with a good configuration/GPU machine is robust. Implementation is quite easy with the help of python libraries and packages, Python mainly uses **Keras** and **Tensorflow** to train the neural network.

You will be studying the implementation of the end-to-end process of constructing a neural network, evaluating the performance of the neural network, and tuning the various hyper-parameters in a neural network.

Let us use the data-set available in the below URL,

https://www.kaggle.com/shrutimechlearn/churn-modelling?select=Churn_Modelling.csv

This data contains the details of the Bank's Customers, it has got 14 Features, and the dependent feature is the '**Exited'** column of the dataset**.**

The objective is to churn if a particular customer will continue or quit the banking services in the future based on the independent features.

You will solve this by constructing an artificial neural network for binary classification. Let us start the coding now, by using python,

Part 1: Preprocessing and loading libraries.

Step 1:

Import and load the required libraries in python using Jupyter notebook.

```
import numpy as np
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt

# Install and call keras
!pip install -q keras
import keras
```

Step 2:

Load the data and check for the initial 5 records of the data set.

```
# Importing the dataset

dataset=pd.read_csv('/content/drive/MyDrive/DL_ANN /churn_modelling.csv')
dataset.head()
```

This piece of code will return the first 5 records in the dataset.

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	15634602	Hargrave	619	France	Female	42	2	0.00	1	1	1	101348.88	1
1	2	15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1	112542.58	0
2	3	15619304	Onio	502	France	Female	42	8	159660.80	3	1	0	113931.57	1
3	4	15701354	Boni	699	France	Female	39	1	0.00	2	0	0	93826.63	0
4	5	15737888	Mitchell	850	Spain	Female	43	2	125510.82	1	1	1	79084.10	0

Step 3:

Check the summary of the dataset with code **dataset.info()**

```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   RowNumber        10000 non-null  int64  
 1   CustomerId       10000 non-null  int64  
 2   Surname          10000 non-null  object 
 3   CreditScore      10000 non-null  int64  
 4   Geography        10000 non-null  object 
 5   Gender           10000 non-null  object 
 6   Age              10000 non-null  int64  
 7   Tenure           10000 non-null  int64  
 8   Balance          10000 non-null  float64
 9   NumOfProducts    10000 non-null  int64  
 10  HasCrCard        10000 non-null  int64  
 11  IsActiveMember   10000 non-null  int64  
 12  EstimatedSalary  10000 non-null  float64
 13  Exited           10000 non-null  int64  
dtypes: float64(2), int64(9), object(3)
memory usage: 1.1+ MB
```

Step 4:

Separate the dependent and independent variables

```
# Split dependent and independent features.
x=dataset.iloc[:,3:13]
y=dataset.iloc[:,-1]
```

```
x.head()
```

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
0	619	France	Female	42	2	0.00	1	1	1	101348.88
1	608	Spain	Female	41	1	83807.86	1	0	1	112542.58
2	502	France	Female	42	8	159660.80	3	1	0	113931.57
3	699	France	Female	39	1	0.00	2	0	0	93826.63
4	850	Spain	Female	43	2	125510.82	1	1	1	79084.10

```
y.head()
```

```
0    1
1    0
2    1
3    0
4    0
Name: Exited, dtype: int64
```

You can also validate with **x.shape** and **y.shape** respectively, which will return the dimensions of the dataset.

Step 5:

From the data set create dummy variables for "**geography**" and "**gender**" categorical features.

```
# Creating dummy variables for categorical variables.
geography=pd.get_dummies(x["Geography"],drop_first=True)
gender=pd.get_dummies(x["Gender"],drop_first=True)
```

This will convert these two features into dummy variables and assign numerical values to the classes within them.

```
geography.head()
```

	Germany	Spain
0	0	0
1	0	1
2	0	0
3	0	0
4	0	1

```
gender.head()
```

	Male
0	0
1	0
2	0
3	0
4	0

Step 6:

Concatenate the above-created dummies to the independent data i.e. **x**

```
# Concat the dummy variables to data ie.  x
x=pd.concat([x,geography,gender],axis=1)
```

You can observe the dummy variables are now appended to the data (final 6 columns).

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Germany	Spain	Male	Germany	Spain	Male
0	619	France	Female	42	2	0.00	1	1	1	101348.88	0	0	0	0	0	0
1	608	Spain	Female	41	1	83807.86	1	0	1	112542.58	0	1	0	0	1	0
2	502	France	Female	42	8	159660.80	3	1	0	113931.57	0	0	0	0	0	0
3	699	France	Female	39	1	0.00	2	0	0	93826.63	0	0	0	0	0	0
4	850	Spain	Female	43	2	125510.82	1	1	1	79084.10	0	1	0	0	1	0

Step 7:

Drop the unwanted features i.e. original “Geography” and “gender” as we are including their dummy variables into the model.

```
# Drop unwanted columns ie.. categorical Gender and Geography
x=x.drop(['Gender','Geography'],axis=1)
```

Check if these variables are dropped or not.

```
x.head()
```

	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Germany	Spain	Male	Germany	Spain	Male
0	619	42	2	0.00	1	1	1	101348.88	0	0	0	0	0	0
1	608	41	1	83807.86	1	0	1	112542.58	0	1	0	0	1	0
2	502	42	8	159660.80	3	1	0	113931.57	0	0	0	0	0	0
3	699	39	1	0.00	2	0	0	93826.63	0	0	0	0	0	0
4	850	43	2	125510.82	1	1	1	79084.10	0	1	0	0	1	0

Part 2: working on the training data.

Step 8:

Split the data into train and test. You will need a sklearn package to do this.

```
# Split the data into train and test
import sklearn
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=.2,random_state=0)
```

Step 9:

Scale the data using a standard scaler, this makes the data independent of units and helps in accelerated computation. You will need to import **StandardScaler** to complete this task.

```
# Scale the data/Feature scaling
from sklearn.preprocessing import StandardScaler
sc=StandardScaler()
x_train=sc.fit_transform(x_train)
x_test=sc.transform(x_test)
```

Step 10:

You can have a glance at the dimensions of the data before proceeding further just to confirm.

```
x.shape
```

```
(10000, 11)
```

Part 3: Building and working on the Algorithm (ANN)

Step 11:

Import all the dependent libraries and packages required to build an artificial neural network.

```
# Importing dependent packages and libraries
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import ReLU,PReLU,ELU,LeakyReLU
```

Step 12:

Before starting to build the neural network you need to initiate a sequential model.

```
# Initializing the Artificial Neural Network
classifier=Sequential()
```

Step 13:

Build the input layer and the first hidden layer, you will use **Dense** to perform this operation, it will create a fully connected dense model.

```
# Adding the input layer and 1st hidden layer
classifier.add(Dense(6, activation='relu', kernel_initializer='he_uniform',input_dim=11))
```

In the above code, **6** is the number of neurons in the **hidden layer 1 (i.e.it outputs the number of neurons for the next layer)**, and **input_dim = 11** because there are 11 input features and you need 11 neurons in the input layer to feed the data into the network, the Activation function is **ReLU** in the hidden layer nodes and Weights initialization technique used is **he-uniform**.

Step 14:

Build the second hidden layer with 6 neurons, ReLU activation, and HE-uniform weights initialization.

```
# Adding the 2nd hidden layer
classifier.add(Dense(6, activation='relu',kernel_initializer='he_uniform'))
```

Step 15:

Add the output layer to the model, as the use-case is a binary classification model, you may use the **Sigmoid Activation function** and **1** neuron in the layer, use **Glorot-uniform** weights initializer**.**

```
# Adding the output layer
classifier.add(Dense(1,kernel_initializer='glorot_uniform',activation='sigmoid'))
```

Step 16:

Now that you have created all the required layers in the network you need to compile the model using **Adam Optimizer** and **binary cross-entropy loss function** to back-propagate the network and use **accuracy** metrics.

```
# Compiling the Neural Network
classifier.compile(optimizer = 'Adam', loss = 'binary_crossentropy', metrics=['accuracy'])
```

Step 17:

Fit the model with **a validation split of 33%, batch-size of 10 and 100 epochs** (you can tune these hyper-parameters)

```
# Fitting the ANN
final_model=classifier.fit(x_train,y_train,validation_split=.33,batch_size=10,epochs=100)
```

As specified the model will run for 100 epochs with 10 batch-size using binary cross-entropy The loss function and accuracy metrics, you will find the below output.

Summary of 1 to 10 epochs

```
Epoch 1/100
536/536 [==============================] - 3s 4ms/step - loss: 0.7861 - accuracy: 0.5825 - val_loss: 0.5143 - val_accuracy: 0.7982
Epoch 2/100
536/536 [==============================] - 2s 3ms/step - loss: 0.4894 - accuracy: 0.8063 - val_loss: 0.4716 - val_accuracy: 0.8050
Epoch 3/100
536/536 [==============================] - 2s 3ms/step - loss: 0.4633 - accuracy: 0.8027 - val_loss: 0.4443 - val_accuracy: 0.8076
Epoch 4/100
536/536 [==============================] - 2s 4ms/step - loss: 0.4188 - accuracy: 0.8225 - val_loss: 0.4317 - val_accuracy: 0.8111
Epoch 5/100
536/536 [==============================] - 2s 4ms/step - loss: 0.4135 - accuracy: 0.8233 - val_loss: 0.4244 - val_accuracy: 0.8103
Epoch 6/100
536/536 [==============================] - 2s 3ms/step - loss: 0.4110 - accuracy: 0.8229 - val_loss: 0.4175 - val_accuracy: 0.8183
Epoch 7/100
536/536 [==============================] - 2s 3ms/step - loss: 0.3940 - accuracy: 0.8336 - val_loss: 0.4104 - val_accuracy: 0.8251
Epoch 8/100
536/536 [==============================] - 2s 4ms/step - loss: 0.3961 - accuracy: 0.8342 - val_loss: 0.4044 - val_accuracy: 0.8323
Epoch 9/100
536/536 [==============================] - 2s 4ms/step - loss: 0.3858 - accuracy: 0.8421 - val_loss: 0.3986 - val_accuracy: 0.8334
Epoch 10/100
536/536 [==============================] - 2s 3ms/step - loss: 0.3743 - accuracy: 0.8432 - val_loss: 0.3936 - val_accuracy: 0.8326
```

Summary of 90 to 100 epochs

```
Epoch 90/100
536/536 [==============================] - 2s 3ms/step - loss: 0.3114 - accuracy: 0.8739 - val_loss: 0.3557 - val_accuracy: 0.8501
Epoch 91/100
536/536 [==============================] - 2s 4ms/step - loss: 0.3310 - accuracy: 0.8699 - val_loss: 0.3566 - val_accuracy: 0.8523
Epoch 92/100
536/536 [==============================] - 2s 3ms/step - loss: 0.3259 - accuracy: 0.8583 - val_loss: 0.3532 - val_accuracy: 0.8576
Epoch 93/100
536/536 [==============================] - 2s 3ms/step - loss: 0.3292 - accuracy: 0.8625 - val_loss: 0.3543 - val_accuracy: 0.8557
Epoch 94/100
536/536 [==============================] - 2s 4ms/step - loss: 0.3214 - accuracy: 0.8674 - val_loss: 0.3533 - val_accuracy: 0.8535
Epoch 95/100
536/536 [==============================] - 2s 3ms/step - loss: 0.3299 - accuracy: 0.8672 - val_loss: 0.3543 - val_accuracy: 0.8523
Epoch 96/100
536/536 [==============================] - 2s 4ms/step - loss: 0.3198 - accuracy: 0.8732 - val_loss: 0.3534 - val_accuracy: 0.8516
Epoch 97/100
536/536 [==============================] - 2s 3ms/step - loss: 0.3259 - accuracy: 0.8648 - val_loss: 0.3556 - val_accuracy: 0.8546
Epoch 98/100
536/536 [==============================] - 2s 3ms/step - loss: 0.3183 - accuracy: 0.8708 - val_loss: 0.3543 - val_accuracy: 0.8542
Epoch 99/100
536/536 [==============================] - 2s 3ms/step - loss: 0.3404 - accuracy: 0.8569 - val_loss: 0.3537 - val_accuracy: 0.8542
Epoch 100/100
536/536 [==============================] - 2s 4ms/step - loss: 0.3268 - accuracy: 0.8703 - val_loss: 0.3524 - val_accuracy: 0.8546
```

It is observed that the loss in the first epoch is higher, and in the 100th epoch the loss is minimized, it is because of the back-propagation, as you studied earlier the goal is to reduce the loss to the minimum.

Step 18:

Visualize the history of accuracy and loss respectively, based on epochs in a plot, before that let us find the keys in the **final_model.**

```
# Check the keys in final_model
final_model.history.keys()
```

This line of code will return the keys as below,

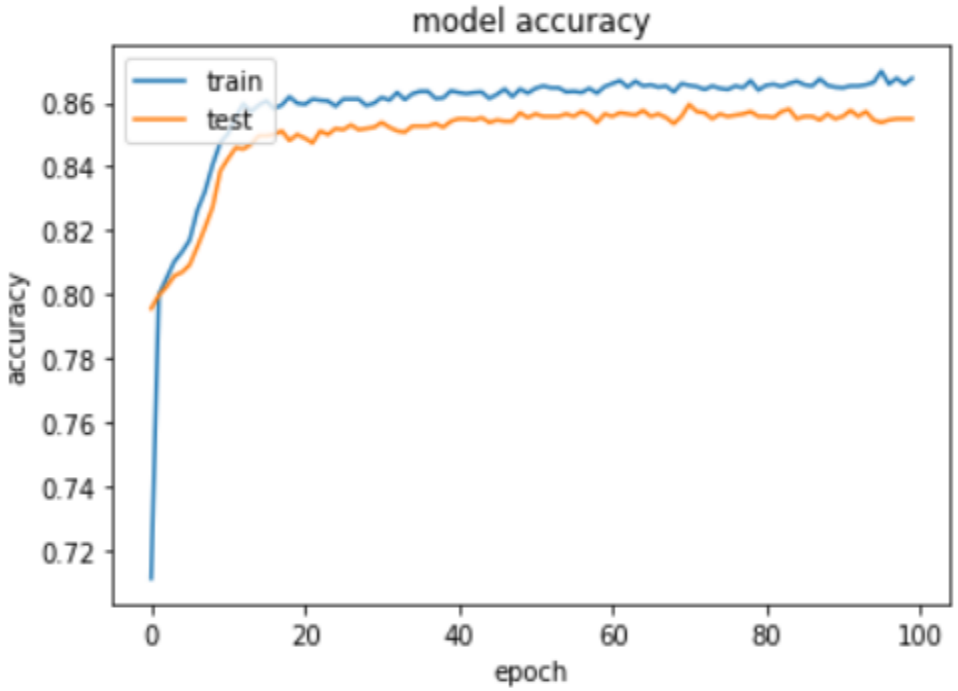
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

Step 19:

Plot the history of **accuracy** against epoch. You will use **matplotlib** to perform this task,

```
plt.plot(final_model.history['accuracy'])
plt.plot(final_model.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

This will return the below plot which shows the history of accuracy against epochs.



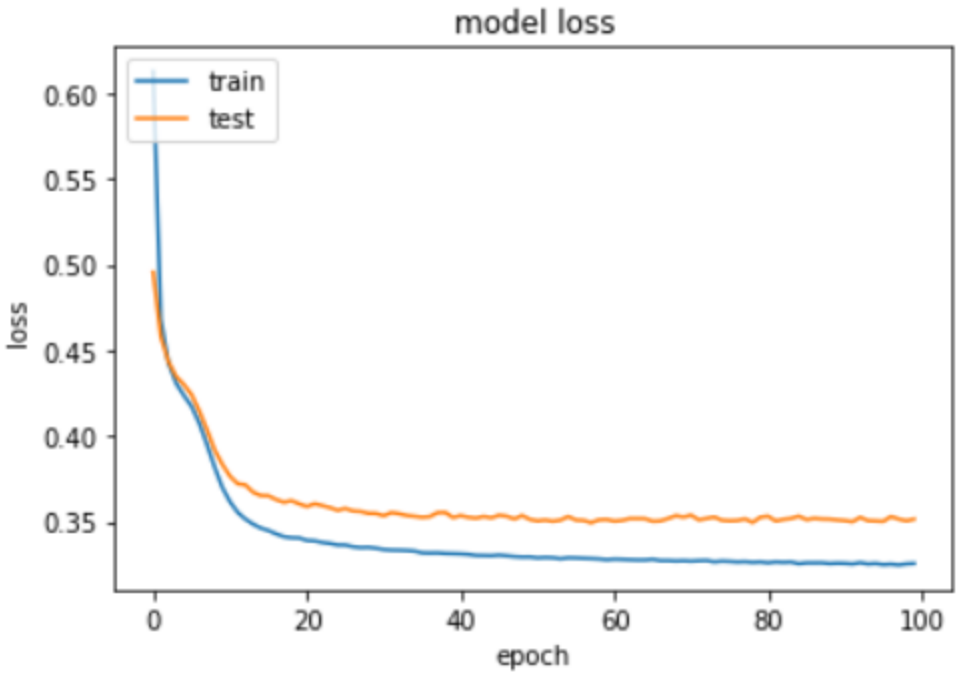
The model accuracy and the number of epochs for both train and test data increased.

Step 20:

Plot the history of **loss** against epoch. You will use **matplotlib** to perform this task,

```
plt.plot(final_model.history['loss'])
plt.plot(final_model.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

This will Return the below plot,



It is evident that the loss is minimized along with epochs, this is the goal of the model, and it is met.

Step 21:

Visualize the Neural Network Created above by writing the below code. For this, you need to install and import **ann_visualizer, graphviz packages.**

```
!pip3 install ann_visualizer
!pip install graphviz
```

Now, create the visualization of the neural network using these packages,

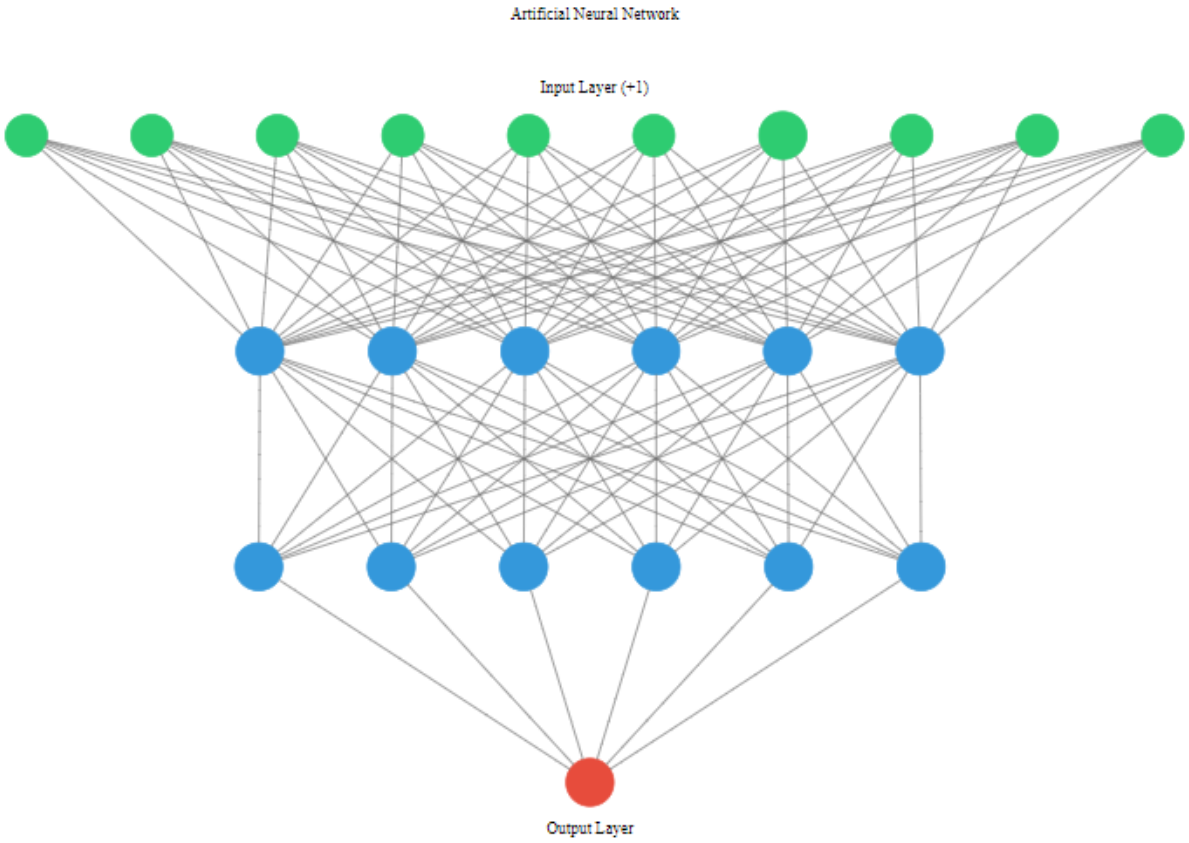
```
# Creating the Visualisation of our neural network.
from graphviz import Source
from ann_visualizer.visualize import ann_viz;

ann_viz(classifier,view=True,filename='network.gv', title="Artificial Neural Network'
```

Where **graphviz import Source** is the "Verbatim DOT source code string to be rendered by Graphviz", ann_viz is to visualize a sequential model, a classifier is our sequential model to be visualized, View=True will allow us to view the visualization, filename = destination to save the visualization. (.gv file), title = title for our visualization.

Call the '**network.gv'** file to view the visualization,

```
# calling the source file pertaining to our neural network visualization
graph_source=Source.from_file('network.gv') # .gv is a graphviz file
graph_source
```

Python will return the visualization of the sequential model i.e. the neural network you have created.



In the above image the neurons in input layer = 10 +1, hidden layer 1 = 6, hidden layer 2 = 6 and output layer = 1 respectively, it is the same as what you asked python to create in steps 13, 14, 15.

Step 22:

```
#Making the predictions and evaluating the model

# Predicting the Test set results
y_pred = classifier.predict(x_test)
y_pred = (y_pred > 0.5) # setting a threshold in the sigmoid function for >0.5

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

# Calculate the Accuracy
from sklearn.metrics import accuracy_score
score=accuracy_score(y_pred,y_test) # returns the Accuracy classification score.
```

Now, check the accuracy score.

```
score # accuracy
```

```
0.863
```

View the confusion matrix,

```
cm # Confusion Matrix
```

```
array([[1508,   87],
       [ 187,  218]])
```

So far you have created an artificial neural network with some random values in place of hyper-parameters, now it is time to tune them, what are the various hyperparameters in the model?

- The number of hidden layers and neurons in each hidden layer.
- Batch-Size and Number of epochs.
- Activation function.
- Optimizer algorithm.

Part 4: Hyper-Parameter Tuning.

Step 23:

Let us start by tuning for the **best number of hidden layers and the number of neurons in the hidden layers along with the activation function** using **GridSearchCV**.

- **Import the necessary packages,**

```
# importing the necessary packages for hyperparameter tuning
from sklearn.model_selection import GridSearchCV
from keras.wrappers.scikit_learn import KerasClassifier
from keras.layers import Dense, Activation, Flatten, Dropout
from keras.activations import relu, sigmoid,tanh,softmax
```

- **Define the model by creating a function as below,**

```
#Defining the model
def create_model(layers, activation):
    model = Sequential()
    for i, nodes in enumerate(layers):
        if i==0:
            model.add(Dense(nodes,input_dim=x_train.shape[1]))
            model.add(Activation(activation))
            model.add(Dropout(0.3))
        else:
            model.add(Dense(nodes))
            model.add(Activation(activation))
            model.add(Dropout(0.3))

    model.add(Dense(units = 1, kernel_initializer= 'glorot_uniform', activation = 'sigmoid'))
```

```
# compiling the model

model.compile(optimizer='adam', loss='binary_crossentropy',metrics=['accuracy'])
return model
```

Where, **Layers and activation** are the two arguments of the function, then a sequential model is initiated, later a loop is created with the subsequent layers and neurons, in which the '**if'** portion of the code pertains to the input layer and **'else'** portion of the code pertains to the remaining layers and neurons in the neural network.

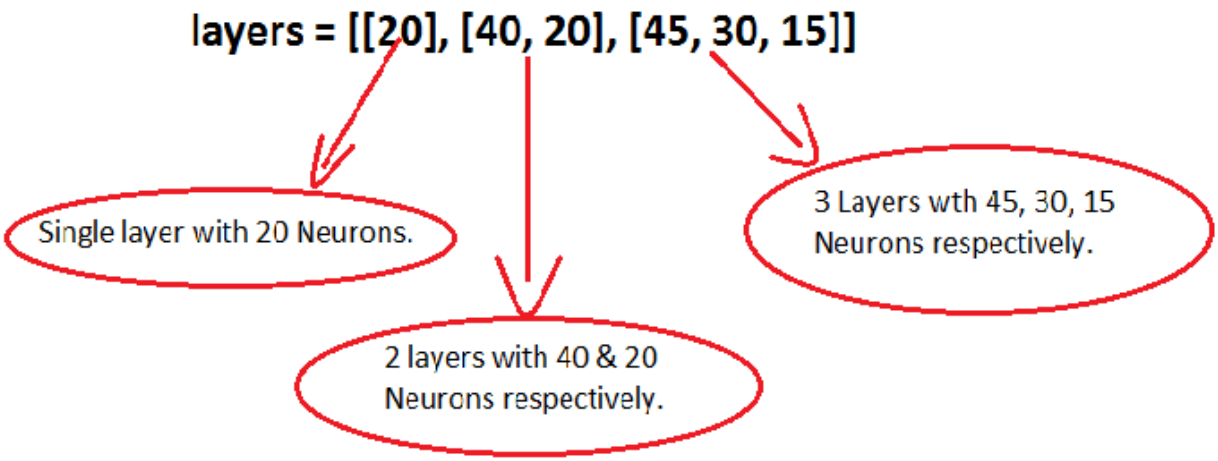
- **Build the above-created model now,**

```
# Building the model
model = KerasClassifier(build_fn=create_model, verbose=0)
```

- **Define the Grid Search parameters,**

```
# defining the grid search parameters
layers = [[20], [40, 20], [45, 30, 15]]
activations = ['sigmoid', 'relu','tanh','softmax']
```

Where, the grid parameters in the **layers** are various combinations of layers and neurons within it, you can interpret it as,



'Activations' parameters are the various activation functions within the grid, the aim is to find the best suitable layers and activation function for the model.

- **Create the grid with parameters,**

```
# creating the grid with parameters
param_grid = dict(layers=layers, activation=activations, batch_size = [128, 256], epochs=[100])
grid = GridSearchCV(estimator=model, param_grid=param_grid,cv=5)
```

- **Fit the grid,**

```
grid_result = grid.fit(x,y)
```

- **Check the result,**

```
[grid_result.best_score_,grid_result.best_params_]
```

Output:

```
[0.7963000059127807,
 {'activation': 'sigmoid', 'batch_size': 128, 'epochs': 100, 'layers': [20]}]
```

Interpretation: The GridSearch has returned the best activation function as 'sigmoid' and the number of layers as 1 layer with 20 Neurons with an optimum accuracy of 0.7963.

Step 24:

Tuning of Batch-size and number of epochs,

- **Let us re-begin by again defining a function to build a neural network by using the tuned values for the number of layers and neurons i.e. 1 layer with 20 neurons, along with activation function i.e. 'sigmoid'.**

```
# creating the neural network by defining the function  with 1 hidden layer an 20 neurons,
# and Sigmoid Activation function.
def ANN_model1():
    #defining my model
    classifier1 = Sequential() # initializing the Sequential model
    # adding the input layer and the first hidden layer with 20 Neurons
    classifier1.add(Dense(20, input_dim=11, activation='relu',kernel_initializer='he_uniform' ))
    # adding the output layer
    classifier1.add(Dense(1,activation='sigmoid', kernel_initializer='glorot_uniform'))

    # Compiling the model
    classifier1.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return classifier1
```

- **Build the defined model,**

```
# Buliding model
final_model1 = KerasClassifier(build_fn=ANN_model1)
```

- **Define the grid parameters for batch size and number of epochs, you don't need to use the same values in the grid, you can try changing the values and the grid size as well,**

```
# define the grid search parameters for batchsize and number of epochs
batchSize = [10, 25, 50]
epochs = [100, 125, 150]
```

GridSearch will test the model with **batch sizes of 10, 25, and 50 and number of epochs of 100, 125,150** with independent trails to find which one is the best for the model.

- **Create the Grid Parameters,**

```
# creating the grid with parameters
parameter_grid1 = dict(batch_size=batchSize, epochs=epochs)
```

- **Execute the GridSearch.**

```
mygrid = GridSearchCV(estimator=final_model1, param_grid=parameter_grid1, n_jobs=-1, cv=3)
grid_result = mygrid.fit(x,y)
```

Where, **estimator** = Estimator object, **param_grid =** dictionary or list of dictionaries, **n_jobs** = Number of jobs to run in parallel. None means 1, -1 means using all processors, **cv =** Cross-validation**.**

- **Check the results of the above grid-search,**

```
# Results of the above gridsearchcv
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

Output,

```
Best: 0.792200 using {'batch_size': 25, 'epochs': 125}
```

Interpretation: The model is optimum with batch-size of 25 and 125 epochs with the best accuracy of 0.7922

Step 25:

Tuning of Optimizer for the model,

Use the tuned values based on the result of GridSearchCV, i.e. **Layers = 1 with 20 neurons, activation function =sigmoid, batch-size = 25, epochs=125.**

- **Define a function to create the neural network as below,**

```
# creating the neural network by defining the function
def ANN_model2(optimizer='adam'):
    #defining my model
    classifier2 = Sequential() # initializing the neural network
    # adding the input layer and first hidden layer comprising of 20 neurons
    classifier2.add(Dense(20, input_dim=11, activation='relu',kernel_initializer='he_uniform' ))
    # adding the output layer
    classifier2.add(Dense(1,activation='sigmoid', kernel_initializer='glorot_uniform'))

    # Compiling the model
    classifier2.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return classifier2
```

- **Building the model,**

```
# Building the model
final_model2 = KerasClassifier(build_fn=ANN_model2, epochs=125, batch_size=25)
```

- **Define the GridSearch Parameters for Optimizer,**

```
# define the grid search parameters for optimizers
optimizer = ['SGD','Adadelta', 'RMSprop', 'Adagrad','Adam']
parameter_grid2 = dict(optimizer=optimizer)
```

- **Execute the GridSearch,**

```
grid2 = GridSearchCV(estimator=final_model2, param_grid=parameter_grid2, n_jobs=-1, cv=5, verbose=0)
grid_result2 = grid2.fit(x, y)
```

Where, **estimator** = Estimator object, **param_grid =** dictionary or list of dictionaries, n_jobs = Number of jobs to run in parallel. None means 1, -1 means using all processors, **cv =** Cross-validation**.**

- **Print the result of the above GridSearch,**

```
print("Best: %f using %s" % (grid_result2.best_score_, grid_result2.best_params_))
```

Output:

```
Best: 0.796300 using {'optimizer': 'SGD'}
```

The so far tuned hyper-parameters are,

- **Layers = 1 layer with 20 neurons,**
- **Activation function = 'sigmoid',**
- **batchsize = 25, epochs = 125,**
- **Optimizer = SGD.**

To use SGD Optimizer we will need the optimum values for Learning-rate and momentum, let us find them.

Step 26:

Tuning the Learning-Rate and momentum for SGD,

- **Import the required packages for the SGD optimizer.**

```
# Importing the required package for SGD
from keras.optimizers import SGD
```

- **Define a function with arguments learn_rate, momentum to build the neural network,**

```
# creating the neural network by defining the function
def ANN_model3(learn_rate=0.01, momentum=0):
    #defining my model
    classifier3 = Sequential() # initializing the neural network
    # adding the input layer and first hidden layer comprising of 20 neurons
    classifier3.add(Dense(20, input_dim=11, activation='relu',kernel_initializer='he_uniform' ))
    # adding the output layer
    classifier3.add(Dense(1,activation='sigmoid', kernel_initializer='glorot_uniform'))

    # Compiling the model
    optimizer=SGD(learning_rate=learn_rate, momentum=momentum)
    classifier3.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return classifier3
```

- **Build the model with the hyper-parameter tuned values of epochs and batch_size**

```
# Building the model
final_model3 = KerasClassifier(build_fn=ANN_model3, epochs=125, batch_size=25)
```

- **Define the GridSearch parameters i.e. learn_rate, momentum.**

```
# define the grid search parameters for optimizers
learn_rate = [0.01,0.1,0.2,0.4]
momentum=[0.0,0.1,0.2,0.4,0.6]
parameter_grid3 = dict(learn_rate=learn_rate,momentum=momentum)
```

- **Execute the GridSearch.**

```
grid3 = GridSearchCV(estimator=final_model3, param_grid=parameter_grid3, n_jobs=-1, cv=5, verbose=0)
grid_result3 = grid3.fit(x, y)
```

- **Check the best result and the parameters.**

```
print("Best: %f using %s" % (grid_result3.best_score_, grid_result3.best_params_))
```

Output:

```
Best: 0.796300 using {'learn_rate': 0.01, 'momentum': 0.1}
```

Interpretation: The learning rate of 0.01 and momentum of 0.1 are the best for the model.

Summary of Hyper-Parameter Tuning

Hyper-parameter	Tuned Result
Number of Layers and neurons	1 Layer and 20 Neurons
Activation Function	Sigmoid activation function
batch-size and number of epochs	batchsize = 25, epochs = 125
Optimizer algorithm	SGD
Learning_rate, momentum	Learning_rate =0.01, momentum=0.1

Programming Assignment:

1) Using the tuned hyper-parameter results in the above summary, construct an artificial neural network and evaluate the model performance.
2) Visualize the loss and accuracy charts.
3) Visualize the neural network with ann_visualizer and graphviz.

Solution 1:

Code for construction of the neural network,

```
# Initializing the Artificial Neural Network
classifier=Sequential()

# Adding the input layer and 1st hidden layer
classifier.add(Dense(20, activation='relu', kernel_initializer='he_uniform',input_dim=x_train.shape[1]))

# Adding the output layer
classifier.add(Dense(1,kernel_initializer='glorot_uniform',activation='sigmoid'))

# Compiling the Neural Network
classifier.compile(optimizer = SGD(learning_rate=0.01,momentum=0.1), loss = 'binary_crossentropy', metrics=['accuracy'])

# Fitting the ANN
final_model=classifier.fit(x_train,y_train,validation_split=.33,batch_size=25,epochs=125)
```

Code for evaluation of model performance,

```
#Making the predictions and evaluating the model

# Predicting the Test set results
y_pred = classifier.predict(x_test)
y_pred = (y_pred > 0.5) # setting a threshold in the sigmoid function for >0.5

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

# Calculate the Accuracy
from sklearn.metrics import accuracy_score
score=accuracy_score(y_pred,y_test) # returns the Accuracy classification score.
```

```
score # accuracy
```

```
0.8585
```

```
cm # Confusion Matrix
```

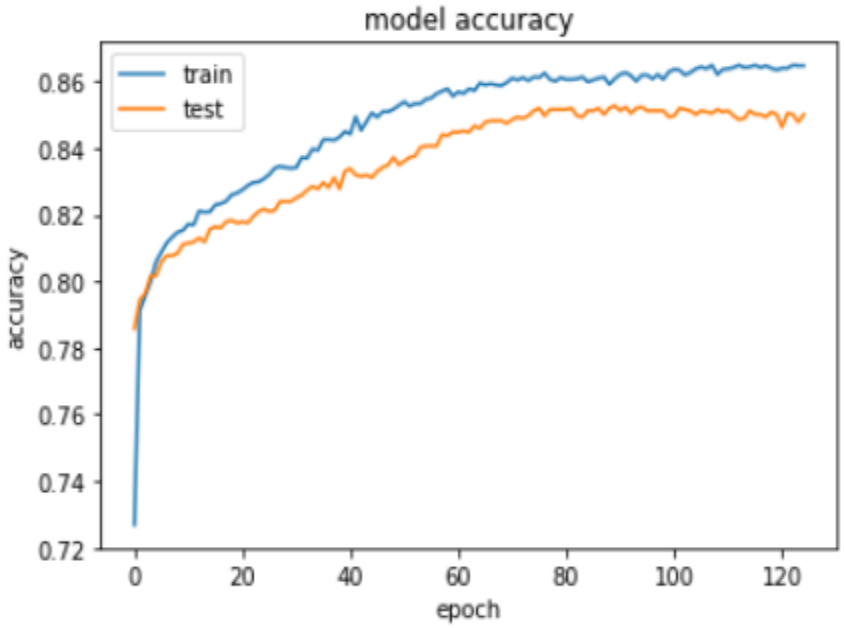
```
array([[1521,   74],
       [ 209,  196]])
```

Solution 2:

Visualizing the Accuracy and loss charts,

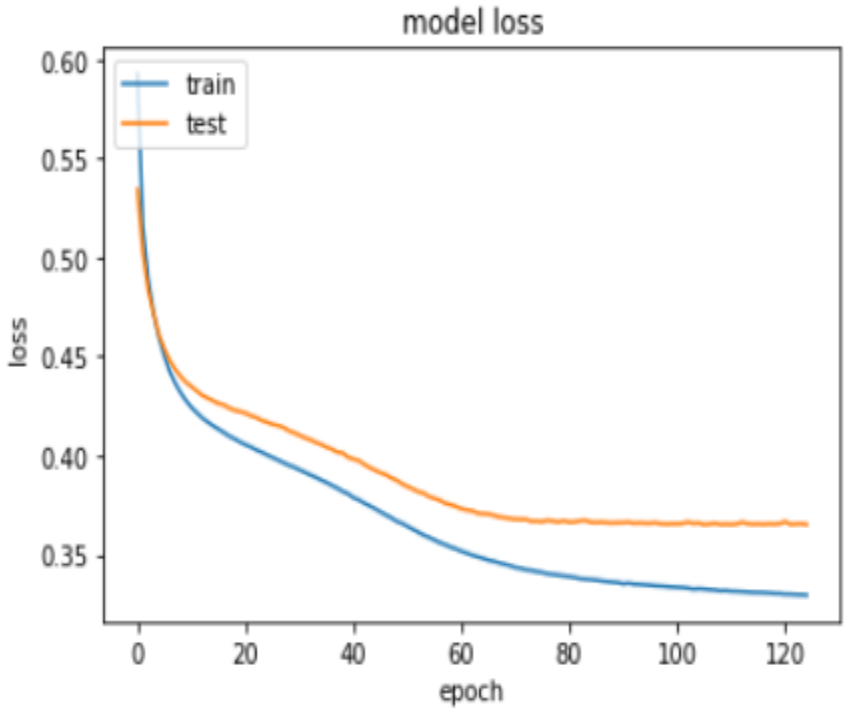
Code to visualize the accuracy against epochs:

```
plt.plot(final_model.history['accuracy'])
plt.plot(final_model.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



Code to visualize the loss against epochs:

```
plt.plot(final_model.history['loss'])
plt.plot(final_model.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



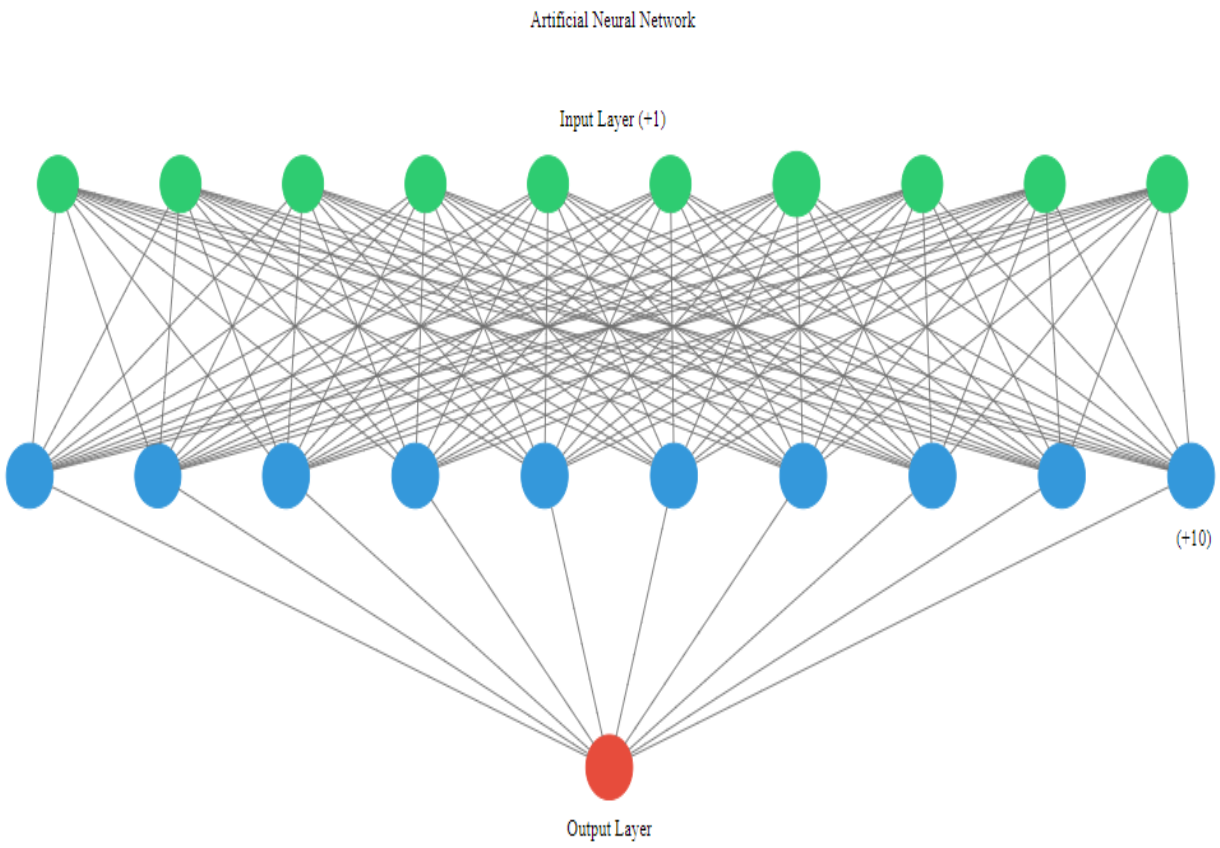
Solution 3: Visualize the neural network.

```
!pip3 install ann_visualizer
!pip install graphviz
```

```
# Creating the Visualisation of our neural network.
from graphviz import Source
from ann_visualizer.visualize import ann_viz;

ann_viz(classifier,view=True,filename='network.gv', title="Artificial Neural Network")
```

```
# calling the source file pertaining to our neural network visualization
graph_source=Source.from_file('network.gv') # .gv is a graphviz file
graph_source
```

Output:



SUMMARY

You have studied about simple Perceptron, it is a single neuron neural network, and multilayered artificial neural networks are the neural networks with at least 3 layers, you will feed the input into the network through the input layer. You have also studied various Weights initialization techniques, activation functions, loss functions, optimizer algorithms, and the process of back-propagating and tuning the weights to optimize the loss.

ASSESSMENT

Choose the appropriate option:

1) **In a Multilayered artificial neural network, how many layers can be created?**
   A. Maximum of 100 layers.
   B. **'N'** number of layers.

2) **Glorot-Normal is a :**
   A. Weights initialization technique.
   B. Optimizer Algorithm.

3) **Momentum in SGD optimizer is used to:**
   A. Increase the noise and slow down the convergence.
   B. Reduce the noise and smoothen the path for accelerated convergence.
   C. None of the above.

4) **Which activation function is the best suitable for multi-class classification use-cases?**
   A. Sigmoid.
   B. Softmax.
   C. Tanh.
   D. All of the above.

5) **The optimizer algorithm helps in:**
   A. Increasing the loss derived from the model.
   B. Reducing the loss derived from the model.

Fill in the spaces with appropriate answers

1) The various Weights initialization techniques are _____, _____, _____, _____.
2) Gradient Descent is an _____ algorithm which helps in minimizing the _____.
3) During back-propagation the weights updating calculation is based on _____.
4) ADAM optimizer is powerful because it combines both _____ and _____.
5) _____ decides the pace of the training in Gradient Descent optimizer.

True or False

1) **Do multilayered artificial neural networks comprise only 1 layer and a single neuron?**
   A. True
   B. False

2) **The cross-entropy loss function is mainly used for classification use-cases?**
   A. True
   B. False

3) **The goal of the Gradient Descent optimizer is to reach the global minimum?**
   A. True
   B. False

4) **Relu Activation function returns Zero (0) for a negative value of input?**
   A. True
   B. False

5) **All the weights are of the same value?**
   A. True
   B. False

Solutions for Assessment

Choose the appropriate options answers

1) B
2) A
3) B
4) B
5) B

Fill in the spaces with appropriate answers

1) He-uniform, He-Normal, Glorot Uniform, Glorot Normal.
2) Optimization, Loss.
3) Chain-Rule of differentiation.
4) Smoothening and RMSprop.
5) Learning_rate.

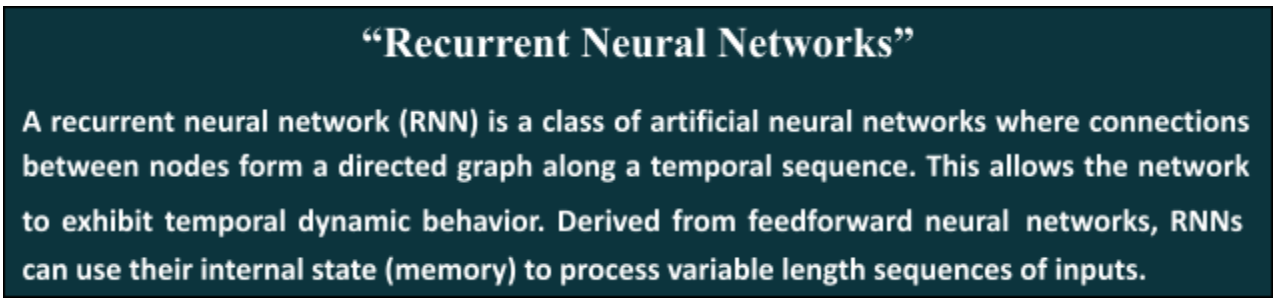
True or False

1) False.
2) True.
3) True.
4) True.
5) False.

CHAPTER 2: RECURRENT NEURAL NETWORKS.

INTRODUCTION:

Recurrent Neural networks are the most powerful neural networks, what makes them powerful is that RNN uses internal memory to compute the prediction/output. This type of neural network was created in the 1980s but it has shown its true potential in recent days.



The internal memory in the RNN allows the network to remember the important information from the input they receive, this results in increased precision of the model when compared to the other available types of neural networks. Thus, RNN is an expert in remembering every detail of information that flows through the network, it mainly remembers and considers the previous information which flows through the architecture.

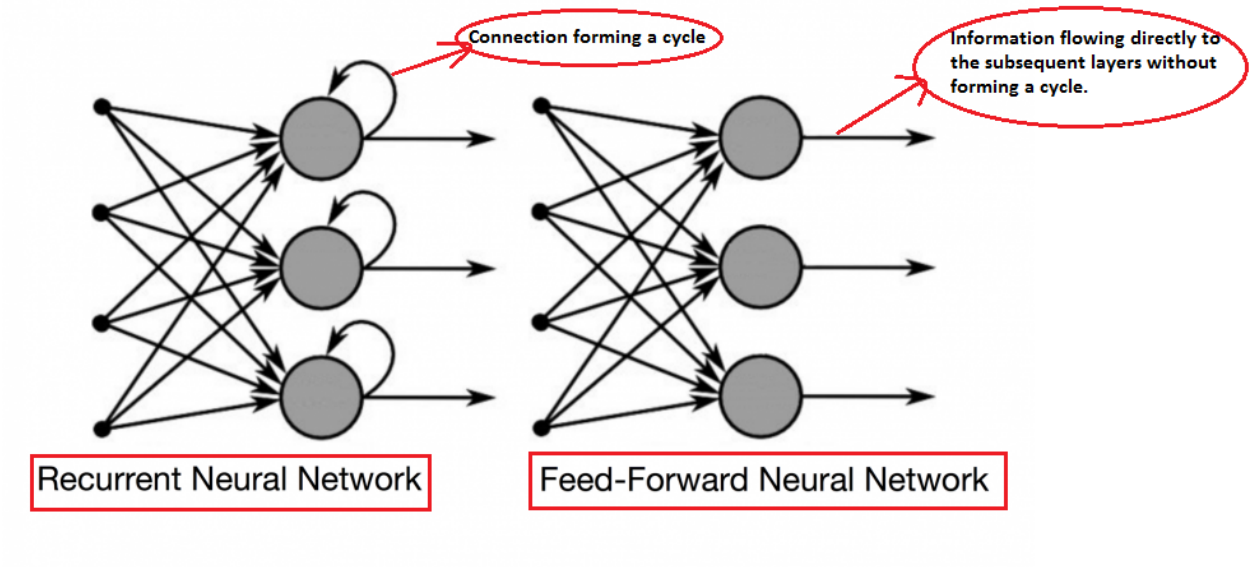
In Feedforward neural networks the connections between the nodes do not form a cycle. But in a Recurrent Neural network, the connections between the nodes will form a cycle, and they perform best when the input data is sequential. Going forward you will study more about sequential data.

This chapter assumes that you have a good understanding of the concepts about feed-forward multilayered artificial neural networks studied in the previous chapter, mainly the various layers in a neural network, various weights initialization techniques, how the values of a neuron are computed, activation functions, loss functions, optimizer algorithms and backpropagation of the neural network.

The architecture of an RNN (Recurrent Neural Network) is similar to that of a feed-forward multilayered artificial neural network, i.e. RNN also comprises of inputs, weights, hidden layers, activation functions, the flow of information from one layer to the other layers to derive the

prediction/output, computation of loss using the loss functions, most importantly the ultimate the goal is to minimize this loss with the help of backpropagation. However, the construction and flow of information of a Recurrent Neural Network are different in comparison to a feed-forward multilayered artificial neural network.

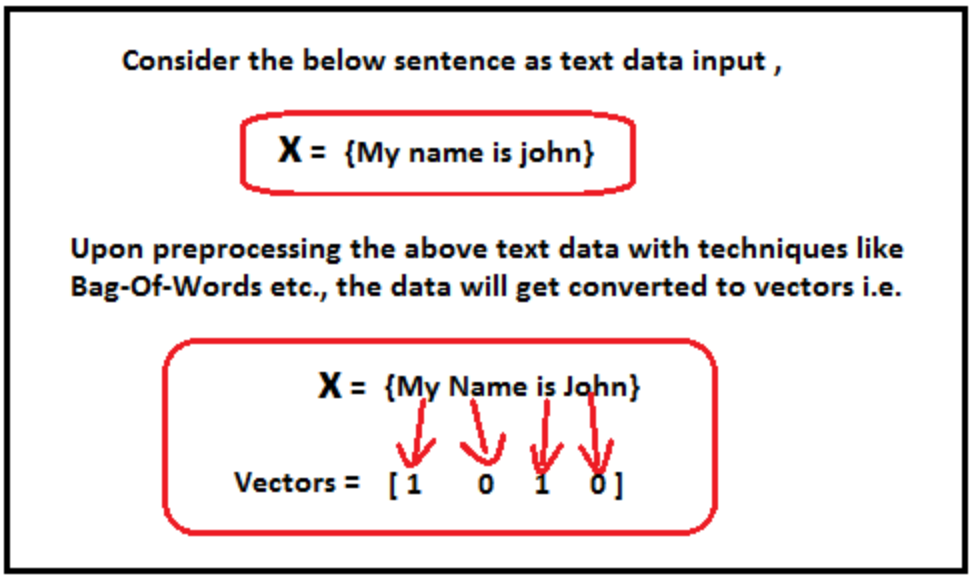
RNN architecture has increased complexity, they save the output of the preceding nodes and feed the result back into the model. I.e. the information is not just passed in one direction but passed in both directions. You can have a glance at the architecture of the feed-forward network and RNN to understand how the information would flow,



RECURRENT NEURAL NETWORKS USE-CASES:

RNN works best when the input is sequential data, Sequential data is a kind of data where the order/sequence of the data plays a vital role, for example, if you are processing text data where the sentence is a sequence, the other example of sequence data is time series forecasting.

Let us consider a simple problem of solving some textual data with Natural Language Processing, few of the available pre-processing techniques about the text are Bag-Of-Words (BOW), Word2VEC (You can study more about Bag-Of-Words (BOW), Word2VEC in the NLP book), these preprocessing techniques will help in converting the text data into vectors before running it through some machine learning algorithm and deriving an output, however during the process the sequence of information is discarded.



Here, the sequence in the text data is important, because if you jumble the sequence it will not form a proper sentence and when you deploy the same into a machine learning model it will hamper the output, thus RNN will help in retaining the sequential information.

Real-World applications built using RNN,

It is understood that Recurrent Neural Networks are powerful networks that use their internal memory to retain every detail of the input received, but the question is, are there any real-world applications that are built using Recurrent Neural Networks (RNN)? Yes, there are countless uses and applications which are built using RNN. Amazon-Alexa, Google Assistant, Google Image Search, google language translator, etc. are the most commonly used real-world examples of applications that are based on Recurrent Neural Networks.

In all the above-quoted applications sequence in the data is vital, consider Amazon-Alexa, when you try asking something, the application will return output in a proper sentence i.e. proper sequence of words. Similar to the other applications which are built using RNN, the Sequence in the data is vital for the model.

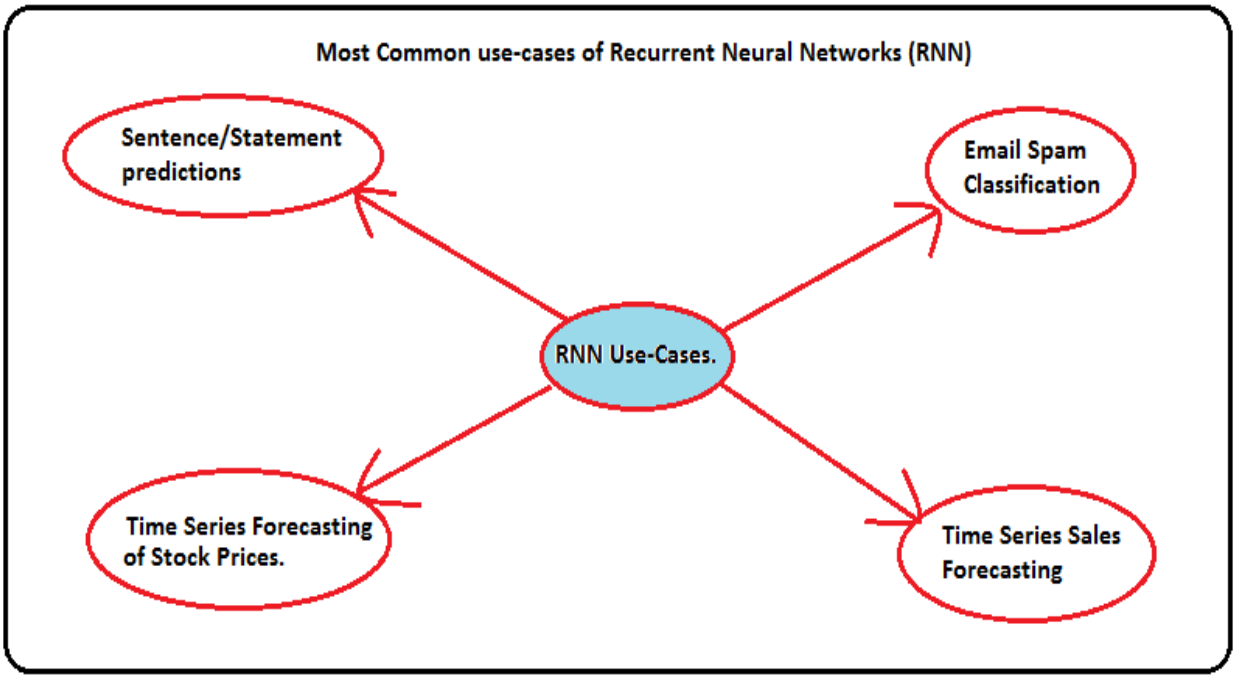
Apart from these examples of real-world applications based on Recurrent Neural Networks, they can also be used in Time-series forecasting use-cases. With machine learning algorithms you have models like ARIMA to perform the task, however, it would consider only a certain

window of preceding time to complete the model but RNN will consider the entire previous data to forecast.

In today's world, Recurrent Neural Networks are widely used for forecasting stock prices. Below is an illustration of time series forecasting,

Image Source: https://money.cnn.com/quote/forecast/forecast.html?symb=jpm

You will study the practical implementation of time-series forecasting in the subsequent chapters i.e. during LSTM Models. For time being understand that RNN can also be used for Time-Series forecasting.



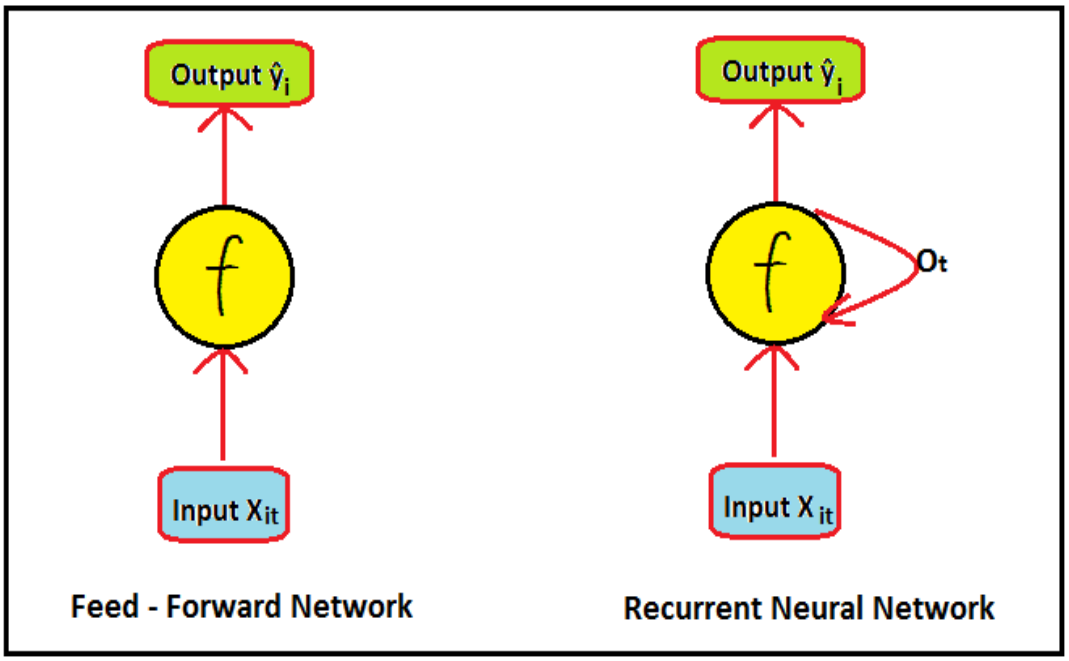
ARCHITECTURE OF RECURRENT NEURAL NETWORKS:

Forward Propagation with respect to time:

Earlier, you have studied the construction of a feed-forward multilayered artificial neural network along with the flow of information between layers to derive the output and minimize the loss, the overall structure and layers in a Recurrent Neural Network are similar to the feed-forward multilayered artificial neural networks i.e. recurrent neural networks will also have an input layer, hidden layers, output layer, loss functions and minimizing the loss, weights, activation functions. Sound understanding of the concepts in chapter 1 is a prerequisite before proceeding further.

Recurrent Neural Networks will also follow a sequence of forward-passing the information and back-propagating the network to update the weights and minimize the loss as it happens in a feed-forward multilayered artificial neural network.

Reiterating the flow of a feed-forward neural network and comparing it with the flow of a Recurrent Neural Network for a better understanding,

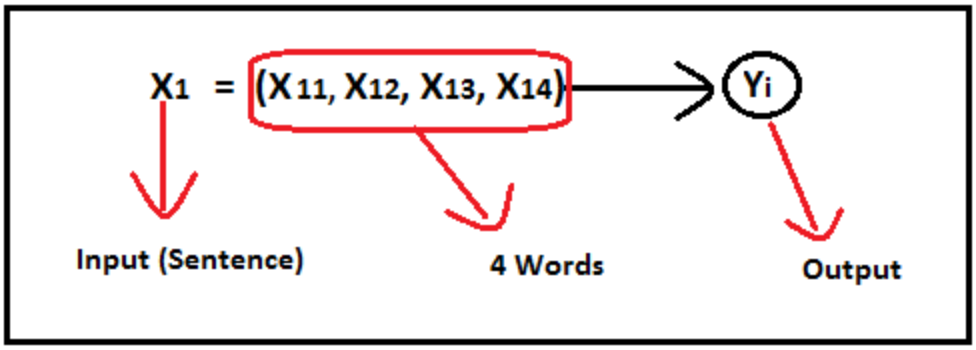


Where, t=time, Ot is the output with respect to time.

You can easily differentiate from the above image that the connection in the feed-forward network is not forming a cycle and the flow of information from the input layer to the output layer is direct, but the connection in the recurrent neural network is forming a cycle. The emphasis in the RNN is revolving around Ot i.e. Output with respect to time.

What is in a recurrent neural network?

In RNN, 't' is not the literal time from the clock, but it is a kind of footsteps that the algorithm takes to process the input. Let us understand this by assuming a crude use-case of performing sentiment analysis of sequential textual data as input,



Where **X1** is the input in the form of a textual sentence comprising of 4 Words.

Upon using RNN to perform this use-case, the network at,

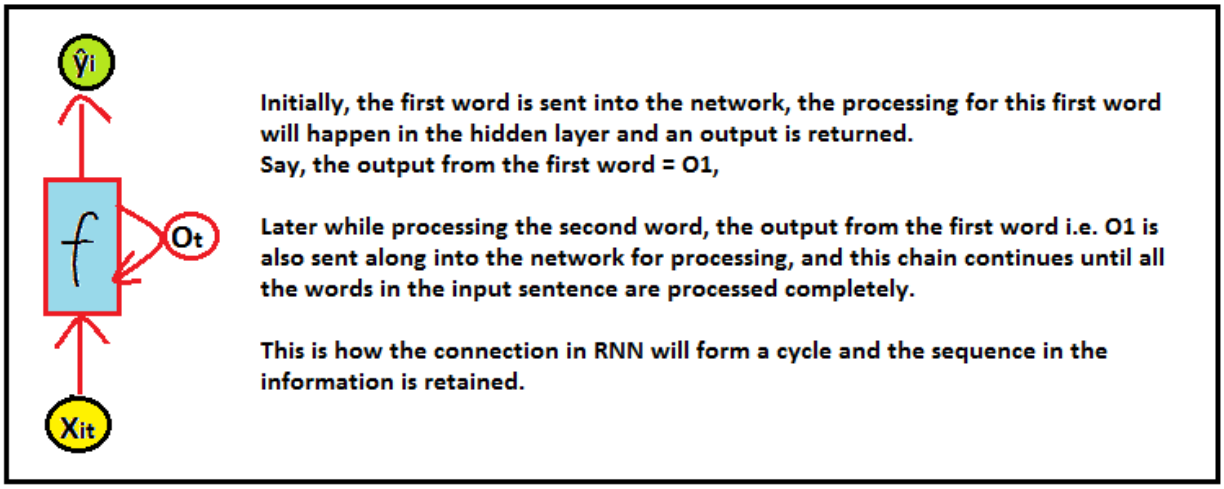
t will process the 1st word, i.e. X11

t+1 will process the 2nd word, i.e. X12

t+2 will process the 3rd word, i.e. X13

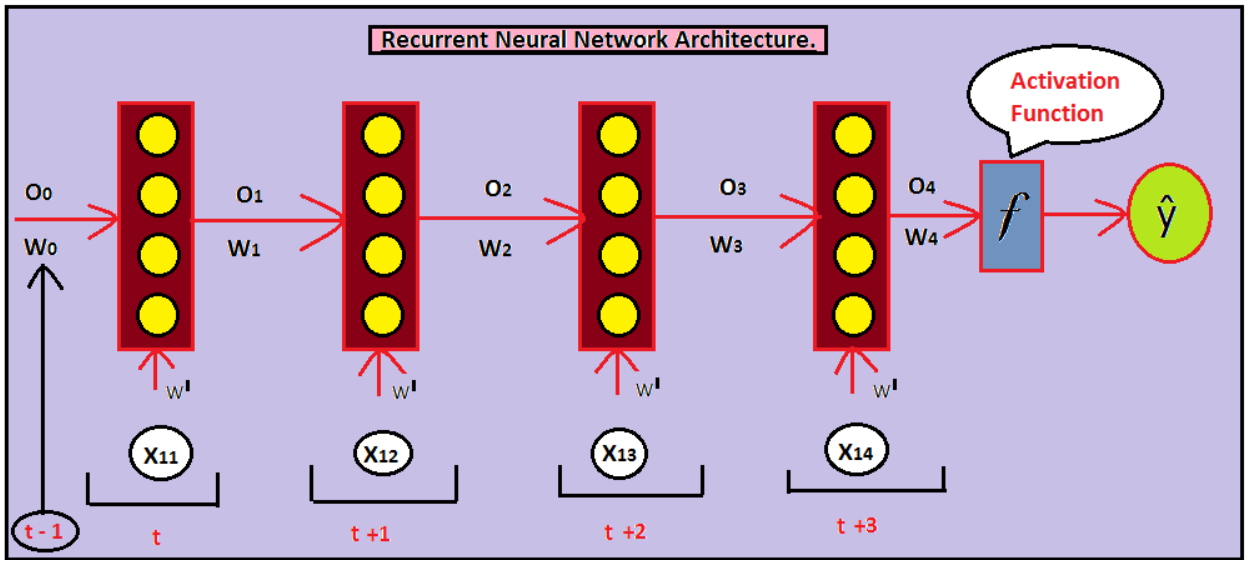
t+3 will process the 4th word, i.e. X14.

Note that you can have 'n' number of words in the input statement.



Therefore, in RNN the output of the previous times is used for processing the outputs of the subsequent times by feeding it along with the inputs, rather than directly passing the information from the input layer to the output layer.

The above image is a glimpse of the general overview of how the flow of information in RNN will form a cycle, now let us unfold the architecture, the full-length architecture of a Recurrent Neural Network would appear like below.



- For input features, we have used a sequential sentence with 4 words, where, X11 = First word, X12 = Second Word, X13 = Third word, X14 =Fourth Word, and these words are converted to vectors (using Bag-Of-Words/TFIDF, Word2VEC) prior passing them through the network. You can refer to the NLP Chapters for a detailed understanding of how the text is converted to vectors.

- While passing these Vectors representing the words into the network, weights get assigned and it is represented with $\mathbf{W^I}$ and this weight remains constant for t, t+1, t+2, t+3.
- Each word is processed individually,

  t will process the 1st word, i.e. X11, and return an output O1,

  t + 1 will process the 2nd word, i.e. X12, and return an output O2,

  t + 2 will process the 3rd word, i.e. X13, and return an output O3,

  t + 3 will process the 4th word, i.e. X14, and return an output O4.

- Weights i.e. W1, W2, W3, W4 are similar to that of a feed-forward neural network, They are initialized using weight initialization techniques and are updated through backpropagation to minimize the loss using an optimizer algorithm.

- You have studied that the connections in RNN will form a cycle and the output of the previous times is used for processing the inputs of the subsequent times, in the above image representing the RNN architecture you can observe that,

  O4 is dependent on X14 and O3,
  O3 is dependent on X13 and O2,
  O2 is dependent on X12 and O1.

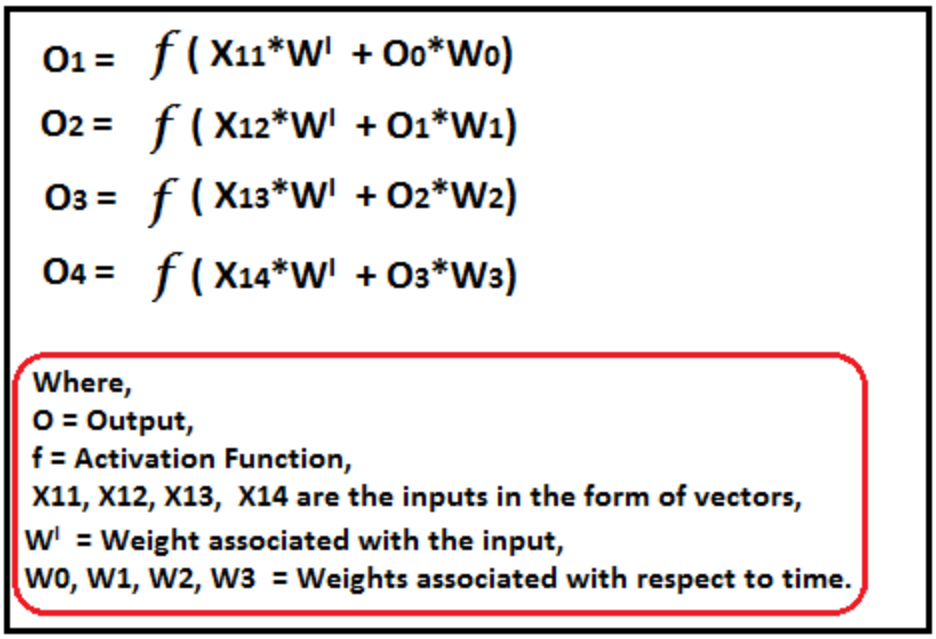
  However, O1 initially did not have any Output from the previous time/step as X11 is the first word at **t** which is passed into the network, Thus the network would initiate the values of **t-1** i.e. O_0, (O=Output) and W_0, (W=Weight) in the preceding connection, for time-being consider that these initialized values are Randomly initialized numerical values.

  It is important to initialize these values because the RNN model will use the preceding time's output to compute the output(s) of the subsequent times/steps.

  Similar to a Feed-Forward Neural Network, every neuron in RNN will have an activation function and Bias associated with it, their functionality and goal remain the same in both kinds of neural networks.

  (For more information on activation functions, refer to page # 25 from chapter 1).

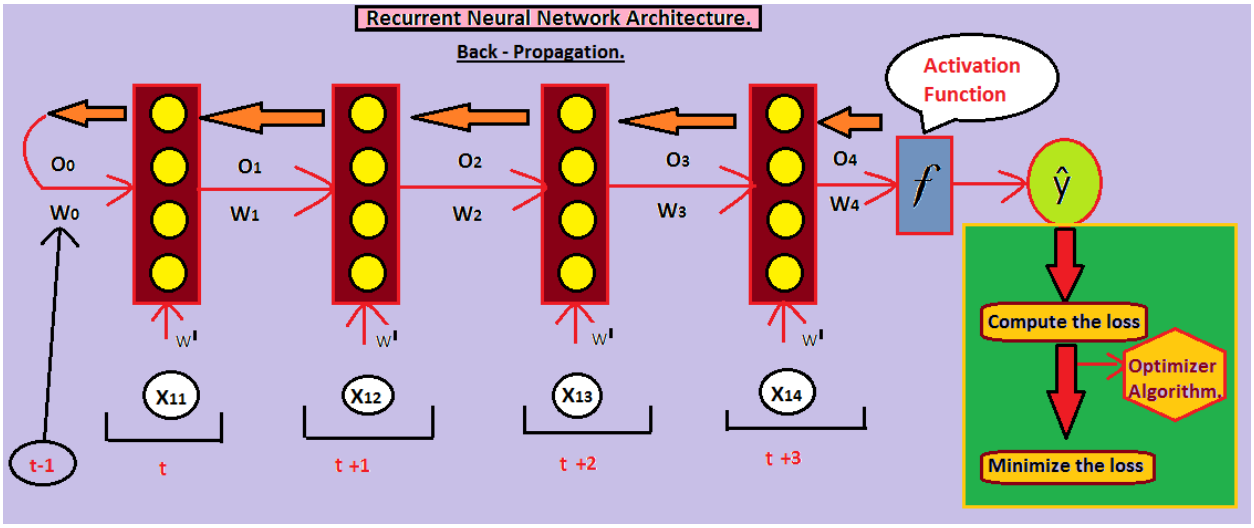
- Let us have a glance at how the values of O1, O2, O3, and O4 are computed,



Once the model returns the final output after the information is passed through the activation function in the final/output layer, the loss from the model is computed and the further goal of the model is to minimize the loss for optimized results. It is the same as it happens in the feed-forward neural network.

Back-Propagation of Recurrent Neural Network with respect to time:

The model will back-propagate to update the weights to reduce the loss derived from the model and the task is accomplished with the support of an optimizer algorithm like Gradient Descent, the goal of backpropagation is similar to that of in a feed-forward neural network. For a better understanding of the concept refer to the previous chapter. (Refer to page # 15 of Chapter 1).



Let us have a glance at how the updated weights are computed during the back-propagation of the neural network.

The weights are back-propagated using the chain rule of differentiation, reiterating the mathematical intuition about the Chain Rule of Differentiation from chapter1,

Chain Rule of Differentiation

$$\text{Multiply } \frac{dy}{dx} \text{ with } \frac{dz}{dz} \text{ we get,}$$

$$\left[\frac{dy}{dx} \times \frac{dz}{dz}\right] = \left[\frac{dy}{dz} \times \frac{dz}{dx}\right] = \frac{dy}{dx}$$

In simple words, Back-propagation is all about finding the derivatives of the existing/old weights and updating those weights such that the tuned weights will contribute towards minimizing the loss derived from the model, here, derivative refers to the slope. The model upon performing a certain number of epochs of tuning the weights it will finally be able to converge i.e. reach the global minimum, it is important to recognize that the training of the model is recurrent and it will continue until the model converges/reaches the global minima.

How are the updated weights computed?

$$\text{Weight}_{New} = w_{Old} - \eta \frac{dl}{dw_{Old}}$$

Where, η is the learning rate

The process will first start by finding the derivative of $\hat{y}$, the model calculates the derivative of loss with respect to derivative of $\hat{y}$ i.e. $\frac{dl}{d\hat{y}_i}$

Now, the weights will get updated starting from the end i.e. W4 and the process of weights updation will continue until the initial Weight at the beginning of the network is updated, this chain of training will continue until the model converges.

$$W4_{New} = W4_{Old} - \eta \frac{dl}{dw4_{Old}}$$

$$= \left[\frac{dl}{dw4_{Old}} = \frac{dl}{d\hat{y}_i} \times \frac{d\hat{y}_i}{dW4_{Old}} \right]$$

Where,

η = Learning Rate

$\frac{dl}{dw4_{Old}}$ = Slope of the weight

Let us now have a glance at how the weight W^l is updated during back-propagation.

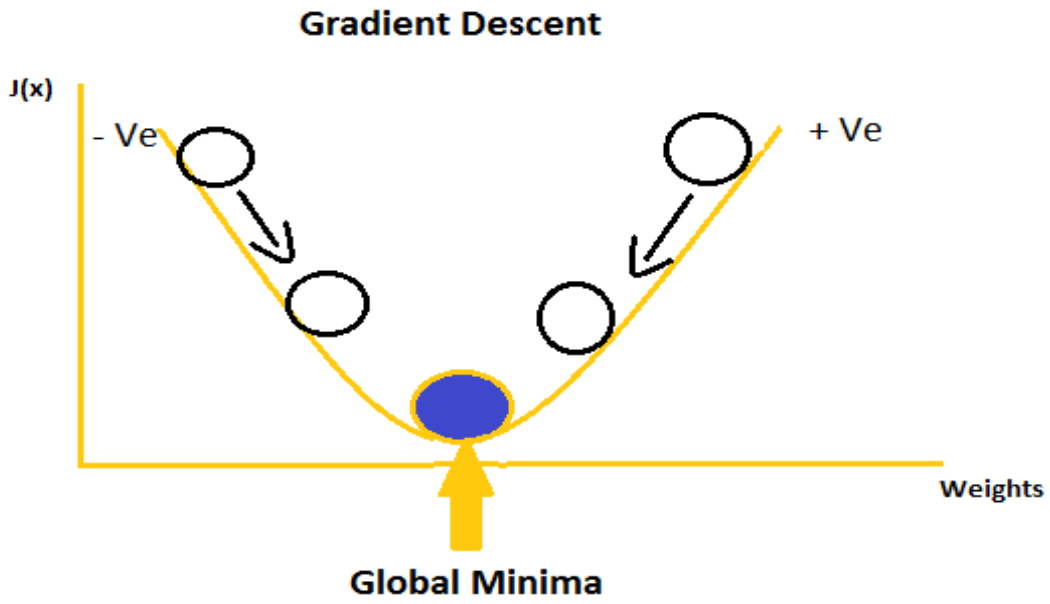
$$w^{l}_{New} = w^{l}_{Old} - \eta \frac{dl}{dw^{l}_{Old}}$$

$$= \left[\frac{dl}{dw^{l}_{Old}} = \frac{dl}{d\hat{y}_i} \times \frac{d\hat{y}_i}{dO_4} \times \frac{dO_4}{dw^{l}_{Old}} \right]$$

Similarly, the remaining weights in the network are updated using the chain rule of differentiation.

PROBLEM WITH SIMPLE RECURRENT NEURAL NETWORK:

You have studied the process of back-propagation where the weights get updated (I.e. the weights are tuned) to minimize the loss, to accomplish this task the model seeks the help of an optimizer algorithm like gradient descent. Reiterating the basic intuition of gradient descent algorithm from chapter1. (Refer to page 23 from chapter 1 for a detailed understanding of the concept).



J(x) = Loss

In the above image, you can observe that the loss function forms a convex curve, the gradients are being resampled in the opposite direction for the model to converge i.e. reach the global minima. The model would not converge if they are resampled in the same direction as the aim is to reach the global minima.

The main problem with RNN back-propagating the weights is that the model may encounter the problems of Vanishing Gradients and exploding gradients when the weights get updated.

Vanishing Gradients Problem:

Vanishing Gradient is a problem encountered during back-propagating and updating the weights in a neural network, the problem arises when the neural network's weights receive an update proportional to the partial derivative of the loss function for the current weight in each iteration of training, in simple words while tuning the weights at a certain point there might arise a situation where, $\mathbf{W_{new} \approx W_{Old}}$, **(W refers to weight**) it means there is no adequate movement for the gradients to reach the global minima and this would hamper the model before it reaches the global minimum.

Example:

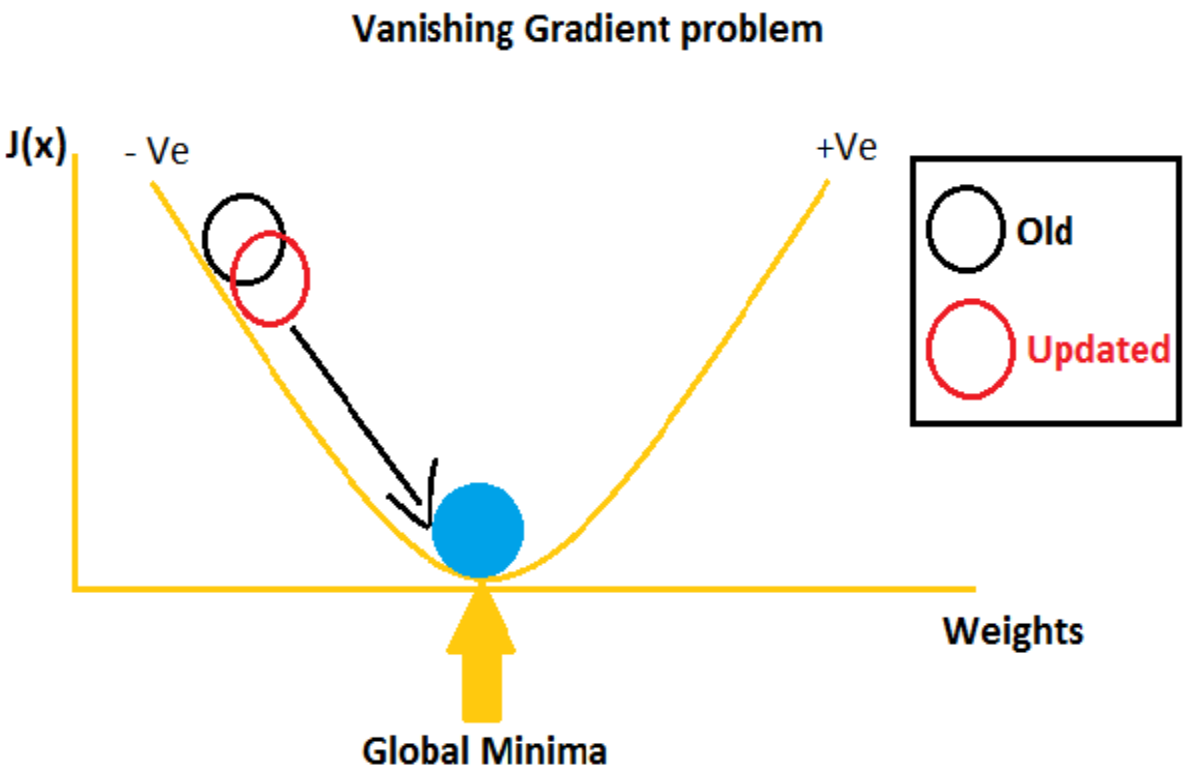
Say, $\mathbf{W_{Old}} = 2.5$ and after back-propagation the weight got updated to 2.4999 i.e. $\mathbf{W_{new}} = 2.4999$

At this point **2.4999 ≈ 2.5**, in this situation there is no adequate movement for the gradients to reach the global minima, this particular problem is called the “vanishing gradient problem”.

Reasons behind Vanishing Gradients problem:

- When the network is deep i.e. the number of layers is large and the weights are too small,
- As sigmoid function ranges between 0 & 1 when we use Sigmoid Activation Function the derivatives of sigmoid are too small leading to smaller weights
- Similar to the sigmoid activation function, using the Tan-H Activation function may also lead to the problem of vanishing gradients.

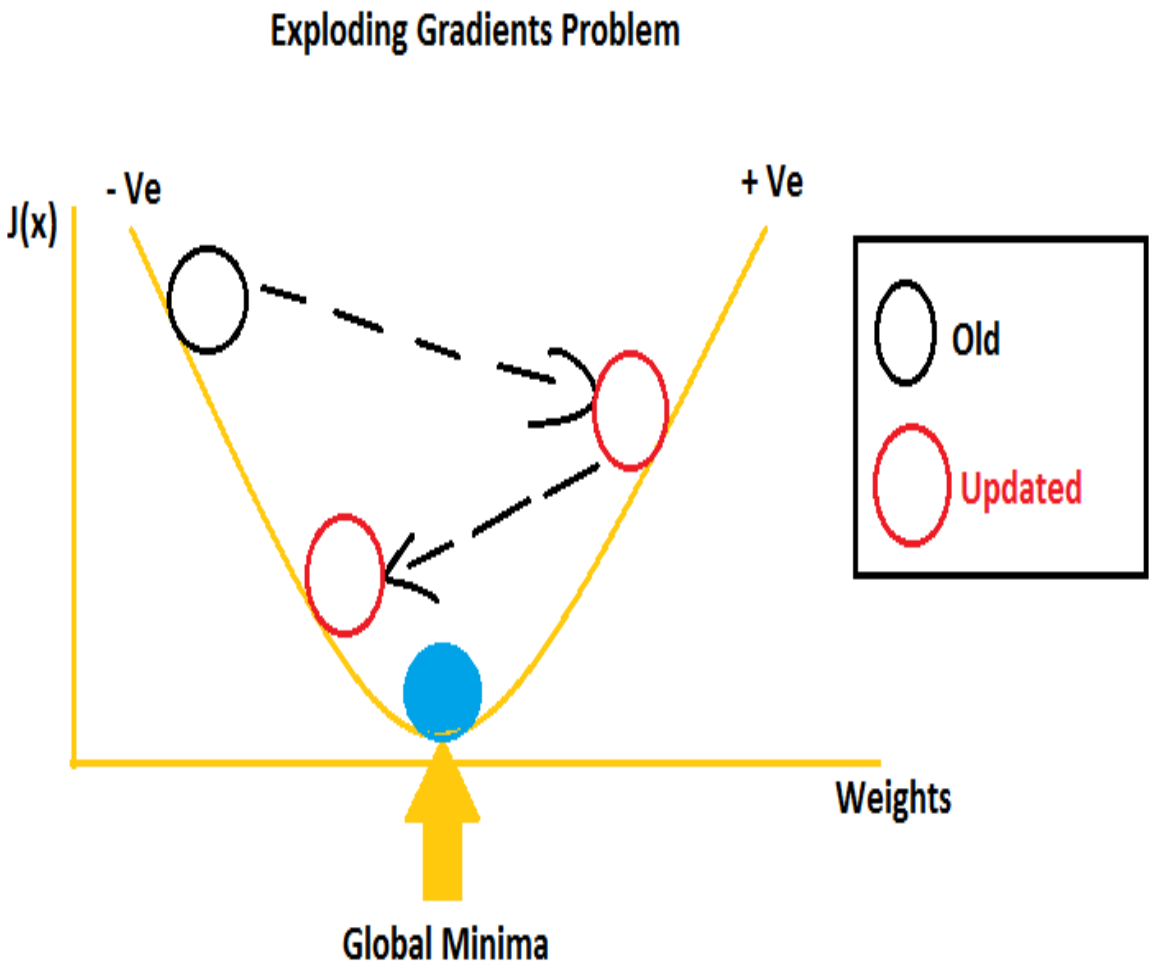
Let us now visualize what happens in the case of Vanishing Gradients,



In the above image, you can observe the updated gradient is approximately the same as the old gradient, and the global minima are far away from the updated weight, thus the model will not converge in this case.

The problem of Exploding Gradients:

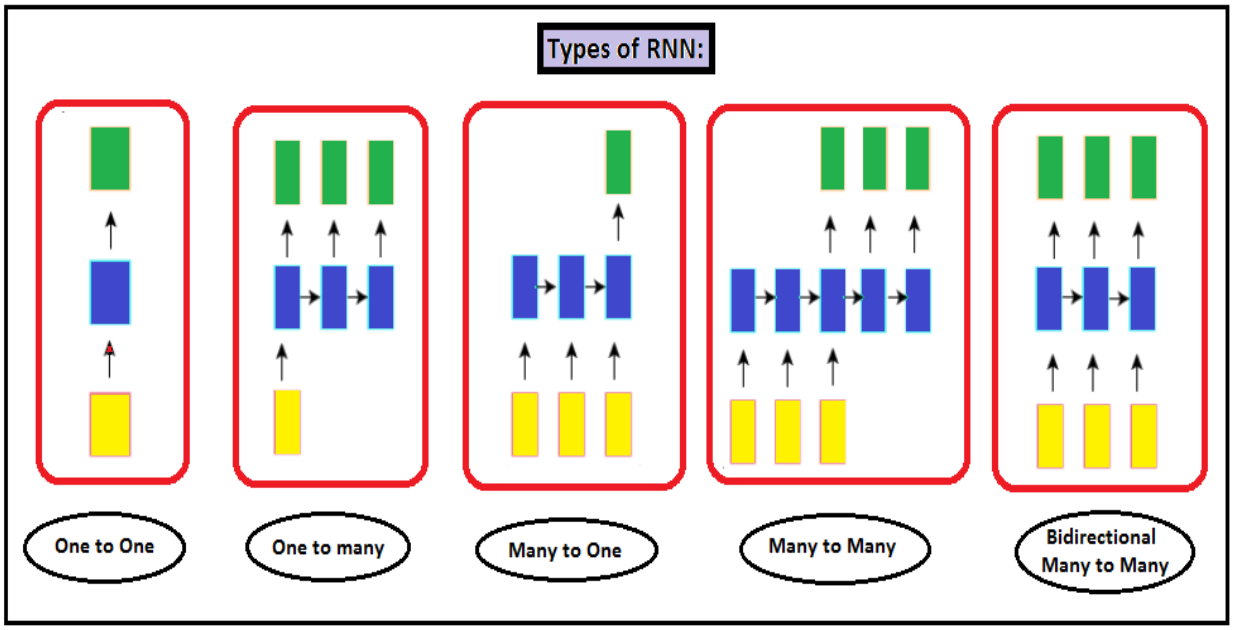
The problem of exploding gradients is reverse of the Vanishing Gradients problem, vanishing gradients problem is encountered when the updated weights are approximately equal to the existing weights, but Exploding gradients problem is encountered when the weights are too large the model would never converge as the gradients will take large steps and it will outrun the global minimum, it is important to understand that the vanishing gradients problem occurs mainly because of Sigmoid and Tanh activation functions, but the exploding gradients problem occurs because of large weights and not Sigmoid and Tanh activation functions.



In the above image you can observe that when the model encounters an exploding gradient problem, the weights do not move steadily towards the global minima as the gradients take large steps and bounce away from the global minimum. Thus when the problem of Exploding gradients is encountered the model would not converge.

As a simple Recurrent Neural Network may encounter the problem of Vanishing and exploding gradients we may consider a better alternative, i.e. **LSTM** model, you will study LSTM in detail in the next chapters.

TYPES OF RECURRENT NEURAL NETWORKS:



Unlike Feed-forward Neural Networks, RNNs can map a flexible variety of outputs based on inputs, however, it is important to remember that RNN takes Vectors as the input.

- One input to one output,
- One Input to many outputs,
- Many inputs to one output,
- Many inputs to many outputs,
- Bidirectional many inputs-to-many outputs.

Now, let us have a glance at the use-cases of each of the above-specified types of RNN.

One-to-one:

This type deals with a constant input and output size, they are independent of previous output.

It can be used in Image classification use-cases.

One -to-many:

This type of network deals with fixed input size and it returns a sequence of data as output.

It can be used in Image captioning use-cases. It can take a single image as input and returns a sequence of words.

Many-to-One:

This type of network takes a sequence of information as input and returns a fixed size of the output.

It can be used in Sentiment analysis use-cases, where the network takes a sequence of information as input and returns a fixed output i.e. if the expression of the input sentence is positive or negative.

Many-to-many:

This type of network takes a Sequence of information as input and recurrently processes the outputs as a Sequence of data.

It can be used in Language translation use-cases, it takes a sequence of data as input and returns a sequence of data as output.

Bidirectional many-to-many:

In the other types of RNN, the information flow is in one direction, however in a Bidirectional many-to-many type of network the information flow is synced in both directions, this enables the prediction of the future output in a sequence.

It can be used in Video-classification use-cases, where you will be able to label every frame of the video.

PRACTICAL IMPLEMENTATION OF SIMPLE RECURRENT NEURAL NETWORKS:

In the previous chapter you have studied the practical implementation of an artificial neural network, now let us study how the Recurrent Neural Network is built using python.

Similar to implementing an ANN, RNN will also use Tensorflow and Keras in python.

For time being let us consider a crude use-case of generating sequential data and implementing it into an RNN model as a part of the basic intuition of the concept, however in the next chapter you will be studying how the stock price forecasting of a particular stock is using LSTM.

Part1: Data Generation

Step 1:

Import the required packages.

```
# Import the required packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from keras.layers import Dense, SimpleRNN
from keras.models import Sequential
```

For a Feedforward network, you have stacked the layers of the network using Dense Layers, in RNN you will use SimpleRNN.

Step 2:

Generate the sequential data-set

```
# Generating the sequential dataset
n=10000
split=7000
t=np.arange(0,n)
x=np.sin(0.01 * t)+2*np.random.rand(n)
```

This piece of code will generate a sequence of numbers with 10000 records

Step 3:

Assign the generated data to a pandas data frame.

```
#Assigning the dataframe to the generated data
df=pd.DataFrame(x)
```

Step 4:

Now that the data frame is generated and assigned let us check the initial records of the data.

```
# Checking the initial records
df.head()
```

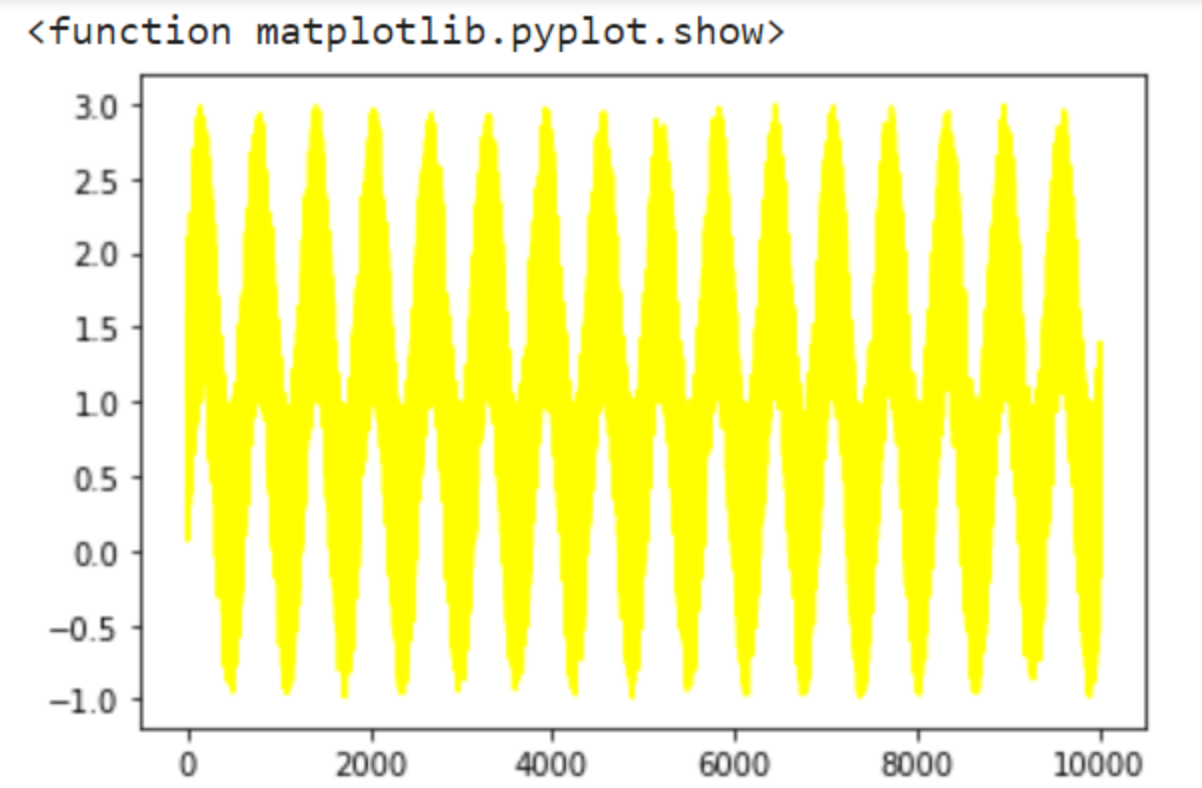
	0
0	1.023500
1	0.647122
2	1.608427
3	0.066706
4	1.416712

Step 5:

Visualize the data to have a glance at the generated d.

```
#Visualizing the data
plt.plot(df,color='yellow')
plt.show
```

This code will return the below chart,



As you have generated a sequential data with 10000 records, the scale on the X-axis of the above plot is having a maximum range of 10000.

Part2: Data Pre-processing and train-test split.

Step 6:

Assign a variable to the values within the data frame.

```
values=df.values
values
```

```
array([[1.02350045],
       [0.64712175],
       [1.60842702],
       ...,
       [1.04740894],
       [1.39546053],
       [0.66959744]])
```

Step 7:

Assign Train and test variables.

```
#train test split
train,test=values[0:split,:],values[split:n,:]
```

In step 2 you have generated a split of 7000 which is 70% of 10000, thus the split ratio will be 70:30

Understanding what is a step in RNN (Keras):

Before proceeding to the next step you need to understand that RNN needs a step value that contains 'n' number of elements as an input sequence, in python, we define it as 'step'.

Assume X Is a sequential data,

X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

If you select step = 1, the X input and Y prediction will be as follows:

X	Y
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10

If you select step = 3, the X input and Y prediction will be as follows:

X	Y
1, 2, 3	4
2, 3, 4	5
3, 4, 5	6
4, 5, 6	7
.....	
7, 8, 9	10

You can notice that when the step value is changing the X i.e. input values and Y i.e. the output values are changing.

Step 8:

Add a step value of 3 into train and test sets.

```
step=3
#adding step elements into train and test
train=np.append(train,np.repeat(train[-1],step))
test= np.append(test,np.repeat(train[-1],step))
```

Step 9:

The RNN network will require a 3-dimensional input of data, thus convert the data into a matrix before reshaping the input data. You can convert the data into a matrix by defining the below function.

```
#convert to matrix
def to_matrix(data,step):
  x,y=[],[]
  for i in range(len(data)-step):
    d=i+step
    x.append(data[i:d,])
    y.append(data[d,])
    return np.array(x),np.array(y)
```

Step 10:

Fit the above-defined function into the train and test, such that they will get converted into a matrix.

```
x_train,y_train=to_matrix(train,step)
x_test,y_test=to_matrix(test,step)
```

Step11:

As mentioned earlier to implement RNN using python, we will need 3-dimensional input data, so the time has arrived to reshape our input data.

```
# RNN needs 3 dimensions in input data, thus re-shape x_train and x_test to fit the RNN model
x_train=np.reshape(x_train,(x_train.shape[0],1,x_train.shape[1]))
x_test=np.reshape(x_test,(x_test.shape[0],1,x_test.shape[1]))
```

Step12:

Check if the data is reshaped into 3-dimensional data or not,

```
x_train.shape
x_test.shape
```

It will return the below output specifying the dimensions of the data,

```
(1, 1, 3)
```

You have successfully reshaped the data into 3-dimensions. Now, let us construct a SimpleRNN model with 64 neurons in the first layer and 8 neurons in the second layer, and one neuron in the output layer.

Part 3: Construction, prediction, and evaluation of the SimpleRNN model.

Step 13:

Initialize a Sequential model.

```
model=Sequential()
```

Step 14:

Constructing the model.

```
#Adding the input and the first Hidden Layers of the RNN
model.add(SimpleRNN(64,input_shape=(1,step),activation='relu',kernel_initializer='he_uniform', return_sequences=True))
# Adding the second hidden layer
model.add(SimpleRNN(8,activation='relu', kernel_initializer='he_uniform'))
# Adding the output layer
model.add(Dense(1))
```

'return_sequences =True' will enable a stack of the layers of the network.

Step 15:

Compile the model using the loss function as 'Mean squared error' and Adam optimizer,

```
# Compiling the model
model.compile(loss="mean_squared_error",optimizer='adam')
```

Step 16:

Check the summary to validate the model you created above.

```
#Model Summary
model.summary()
```

This code will return the summary of the constructed model,

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 1, 64)             4352
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 8)                 584
_________________________________________________________________
dense (Dense)                (None, 1)                 9
=================================================================
Total params: 4,945
Trainable params: 4,945
Non-trainable params: 0
```

You can observe that a sequential model is created and the model is having 3 layers.

Step 17:

Fit the model with training epochs of 100 and batch size of 50,

```
# Fitting the model
model.fit(x_train,y_train,epochs=100,batch_size=50)
```

The model will train for 100 epochs as you can observe below,

```
Epoch 94/100
1/1 [==============================] - 0s 13ms/step - loss: 1.1399e-10
Epoch 95/100
1/1 [==============================] - 0s 7ms/step - loss: 5.5712e-10
Epoch 96/100
1/1 [==============================] - 0s 7ms/step - loss: 1.1161e-09
Epoch 97/100
1/1 [==============================] - 0s 6ms/step - loss: 1.5883e-09
Epoch 98/100
1/1 [==============================] - 0s 6ms/step - loss: 1.8112e-09
Epoch 99/100
1/1 [==============================] - 0s 6ms/step - loss: 1.7802e-09
Epoch 100/100
1/1 [==============================] - 0s 7ms/step - loss: 1.4575e-09
<keras.callbacks.History at 0x7f2f4255b410>
```

Step 18:

Making the predictions

```
# Making the predictions
train_pred=model.predict(x_train)
test_pred=model.predict(x_test)
pred=np.concatenate((train_pred,test_pred))
```

Step 19:

Evaluate the Loss,

```
#Evaluation of loss
score = model.evaluate(x_train,y_train,verbose=0)
print(score)
```

Output:

```
1.0259237903653684e-09
```

Programming Assignment:

1) Build a simple RNN model with n=5000 and split=3000 and,
2) Stack another layer with 4 outputs to the existing model in step 14,

Hint for 1st assignment: In step 2 use the below values,

```
# Generating the sequential dataset
n=5000
split=3000
t=np.arange(0,n)
x=np.sin(0.01 * t)+2*np.random.rand(n)
```

Hint for 2nd assignment:

```
model=Sequential()
#Adding the input and the first Hidden Layers of the RNN
model.add(SimpleRNN(64,input_shape=(1,step),activation='relu',kernel_initializer='he_uniform',
                    return_sequences=True))
# Adding the second hidden layer
model.add(SimpleRNN(8,activation='relu', kernel_initializer='he_uniform', return_sequences=True))
# Adding the Third hidden layer
model.add(SimpleRNN(4,activation='relu', kernel_initializer='he_uniform'))
# Adding the output layer
model.add(Dense(1))
```

SUMMARY

You have studied the Recurrent Neural Networks, how the model is built, how the model is back-propagated, and how it differs from the Feed-forward networks. It is mainly used when the input data is sequential. Some of the real-world use-cases of RNN are Time-series forecasting, Speech recognition, Machine translation, Music composition, Handwriting recognition, Grammar learning.

Advantages of Recurrent Neural Network

- RNN can model a sequence of data so that each sample can be assumed to be dependent on previous ones.

Disadvantages of Recurrent Neural Network

- Vanishing Gradient and exploding Gradient problems.
- Training an RNN is a very difficult task.
- It cannot process very long sequences if using **Tanh** or **Relu** as activation functions.

ASSESSMENT

Choose the appropriate option:

1) **RNN is best used when the input data is:**
   A. Random alphabetical values from a feature.
   B. Sequential data.

2) **The inputs in a Recurrent Neural Network are in the form of:**
   A. Alphabets.
   B. Boolean.
   C. Vectors.

3) **Few Real-world examples of applications built using RNN are**
   A. Amazon-Alexa.
   B. Google Assistant.
   C. Google Image Search.
   D. All of the above.

4) **Which model can you consider using as an alternative to SimpleRNN, mainly due to the Vanishing and exploding gradients problem?**
   A. LSTM.
   B. Feed-forward multilayered network.
   C. Both the above.

5) **The weights in the network are tuned during,**
   A. Forward-pass.
   B. Back-propagation.

Fill in the spaces with appropriate answers

1) _____ and _____ gradients are the problems of a recurrent neural network.
2) The Connections in a recurrent neural network will form a _____.
3) The _____ rule of differentiation is used to update the weights in back-propagation of an RNN model.
4) To implement RNN with Keras the input data shall have _____ number of dimensions.
5) The various types of RNN are _____, _____, _____, _____, _____.

True or False

1) **In RNN the output of the previous connection is ignored while computing the outputs of the subsequent connections?**
   A. True.

B. False.

2) **In the RNN model, the process of forwarding and back-propagation happens with respect to time?**
   A. True.
   B. False.

3) **The problem of Vanishing Gradient is encountered because of the Sigmoid Activation function?**
   A. True.
   B. False.

4) **An RNN model is easy to train?**
   A. True.
   B. False.

5) **RNN model cannot solve long sequences of data using Tanh or Relu activation function?**
   A. True.
   B. False.

Solutions for Assessment

Choose the appropriate options answers

1) B
2) C
3) D
4) A
5) B

Fill in the spaces with appropriate answers

1) Vanishing and exploding gradients.
2) Cycle
3) Chain-Rule of differentiation.

4) Three
5) One to One, One to Many, Many to One, Many to Many, Bidirectional Many to Many.

True or False

1) False.
2) True.
3) True.
4) True.
5) False.

CHAPTER 3: LSTM (Long Short Term Memory).

INTRODUCTION:

LSTM is a special type of recurrent neural network architecture used in the field of deep learning, unlike the standard feedforward networks, LSTM has feedback connections. It can be used not only to process a single input data such as images but can also be used to process a complete sequence of data such as speech, video, etc., Time series predictions, Hand-writing recognition, Speech recognition are few examples of LSTM applications.

LSTM Networks are experts in learning long-term dependencies as they are explicitly designed to handle the long-term dependency problem. These networks are capable of remembering information for a long period of time. Let us understand more about long-term dependencies.

"LSTM"

Long Short-Term Memory (LSTM) networks are a modified version of recurrent neural networks, which makes it easier to remember past data in memory. The vanishing gradient problem of RNN is resolved here. LSTM is well-suited to classify, process and predict time series given time lags of unknown duration. It trains the model by using back-propagation.

THE PROBLEM OF LONG TERM DEPENDENCIES:

You have studied in the previous chapter that the connections in RNN will form a cycle, it uses the previous time's output to process the output(s) of the subsequent times. Therefore in few real-world use-cases to perform a task based on sequential data such as sentence prediction you will need the previous information to predict the next word.

For example, if you are trying to predict the last word in the following sentence, "The capital of the United States is **Washington**", Here you don't need any further context because it is obvious that the last word is Washington, In such cases the gap between the relevant information and the place that it is needed is small and RNN can perform the task without difficulty based on the previous information. However, while predicting sentences there might also be situations where further context is required, Consider predicting the last word from the following sentence, "Washington is the capital of the **United States**… the current president is **Joe Biden.**" Here the previous information suggests that the name of the president to be predicted is relevant to the context of the United States, for RNN it is possible that the gap between the relevant information and the point where it is needed to become very large. If the gap is very large the RNN model will not be able to learn to connect the information, thus leading to further difficulties in making the predictions. LSTM is a special kind of RNN that does not encounter the problem of long-term dependencies.

Reiterating the overview of RNN architecture where the connections form a cycle, i.e. the neural network would form a chain of repeating modules, the repeating module in standard RNN model will have a simple structure, such as a single Tanh layer.

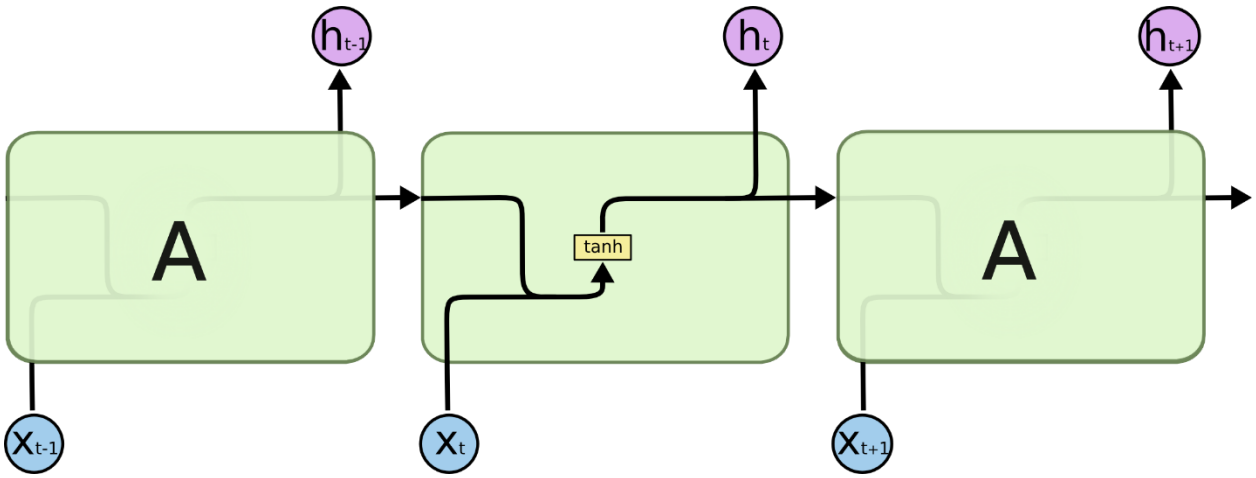


Image Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM models will also have a chain-like structure, however, unlike the standard RNN model the repeating module is a little bit complex, instead of having a single layer the LSTM models will have four interacting layers, namely:

- Memory Cell,
- Forget Gate,
- Input Gate,
- Output Gate.

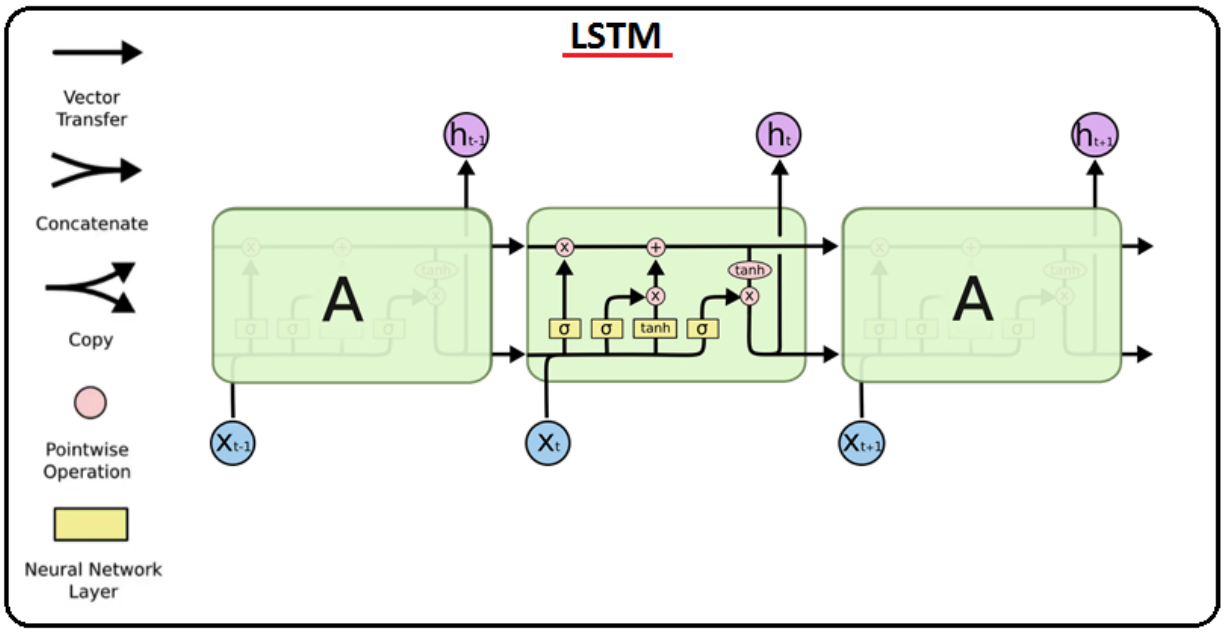
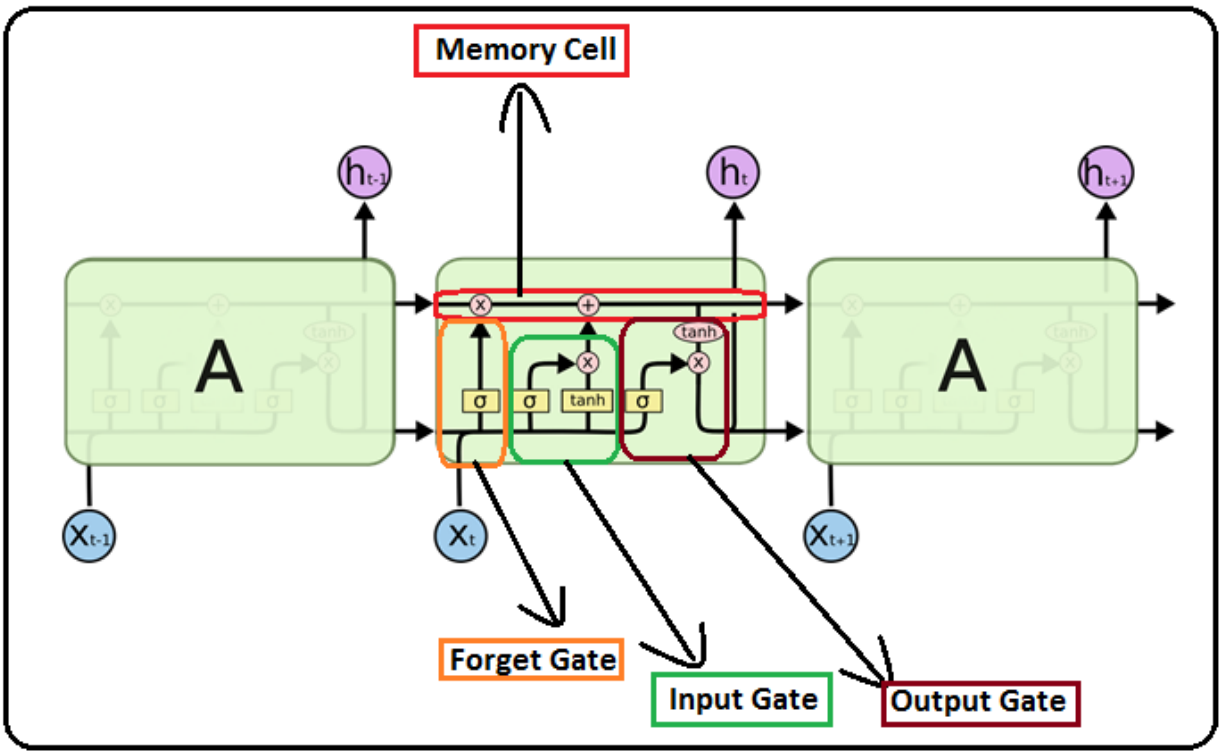


Image Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Let us label the architecture of LSTM with the 4 interacting layers,



Core Idea behind LSTM models:

LSTM model's function is to remember and forget the information while making predictions, the action of remembering and forgetting is based on the context of the information, let us reiterate the example of predicting the final word from the sentence, "Washington is the capital of the **United States**… the current president is **Joe Biden.**" The LSTM model helps in remembering the relevant context in the sentence based on the previous information, here the relevance to remember is Washington and United States, the neural network model shall be able to predict the president of United States, Now if you change the relevant context i.e. Washington to London and the United States to the United Kingdom the neural network shall be able to forget the previous relevant context and be able to make predictions based on the current relevant context.

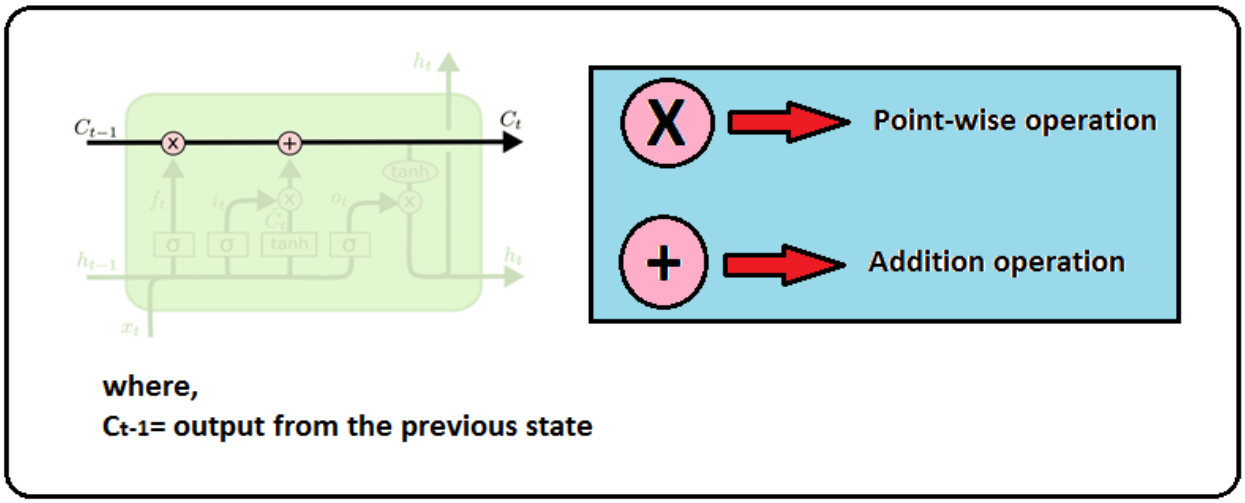
Assume the prediction without the forget action after the relevant context is changed to London and the United Kingdom, the model prediction would appear as follows, "London is the capital of the **United Kingdom**… the current president is **Joe Biden.**" This prediction is an absolute blunder, thus the forget action within the model plays a vital role in such cases.

To summarize, without the remember action the model will not be able to remember the relevant context to return the precise prediction, and without the forget action the model will return inappropriate predictions in case the relevant context is changed.

Let us have a look at each of these interacting layers and their importance in the model.

Memory Cell:

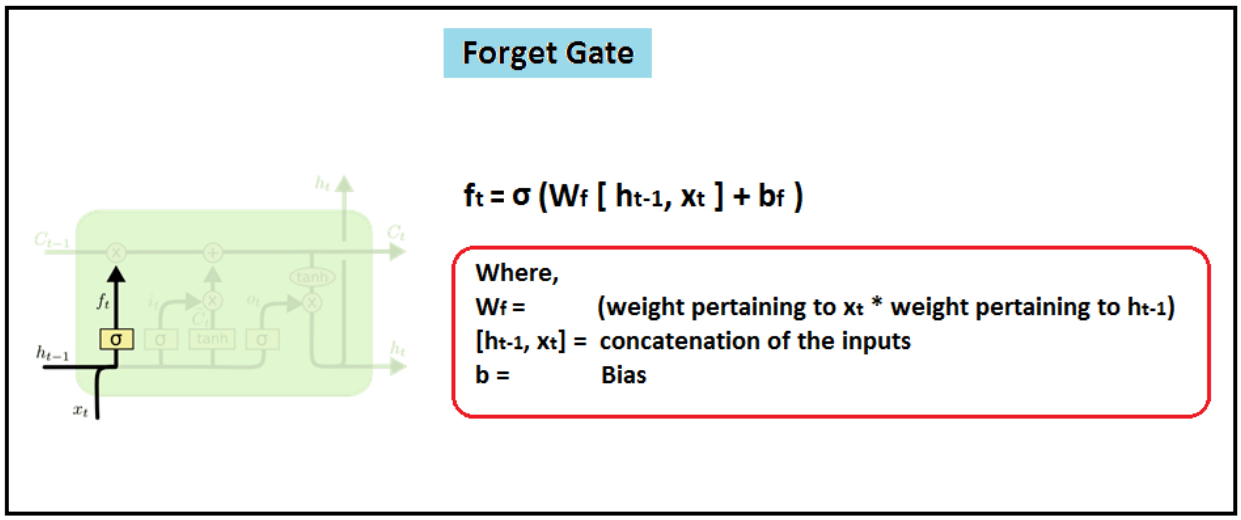
Consider the functionality of memory cell is similar to the brain, The Memory cell deals with forgetting some of the previous information when the relevant context is changed and it supports in remembering the new information of the changed relevant context.



The memory cell comprises 2 components the point-wise operation, and the addition operation.

- **point-wise operation:** In the previous chapter you have studied that RNN mainly deals with sequential data and the inputs are in the form of vectors and these vectors keep changing based on input data, the core idea behind the LSTM model is to remember and forget the information, the role of the point-wise operation in LSTM is that it supports in updating the vectors when the relevant context is changed and when the relevant context is unchanged the vectors remain unaffected. This is done by performing a simple point-wise multiplication between the vector of existing relevant context and the vector of the updated/changed context.
- **Addition operation**: The main functionality of LSTM is that it forgets the previous information and adds new information to the model when the relevant context of information is changed, the addition operation supports the model in adding new information when the relevant context is changed.

Forget Gate:



When we input the information and change the relevant context the LSTM model will forget some information and retain some information. The key functionality of the forget gate in the LSTM model is that it discovers the information which needs to be discarded by using the sigmoid function. The sigmoid function is the one that returns values between 0 & 1 irrespective of the input values.

In the above image, h_{t-1} is the input from the previous state and the current input is x_t, upon feeding these inputs, the sigmoid function will return outputs with values between 0 & 1, where 0 means omit and 1 means retain for each number in the cell state C_{t-1}

For example, let us assume h_{t-1} as the following sentence, “John and Michael are good at Soccer.” and,

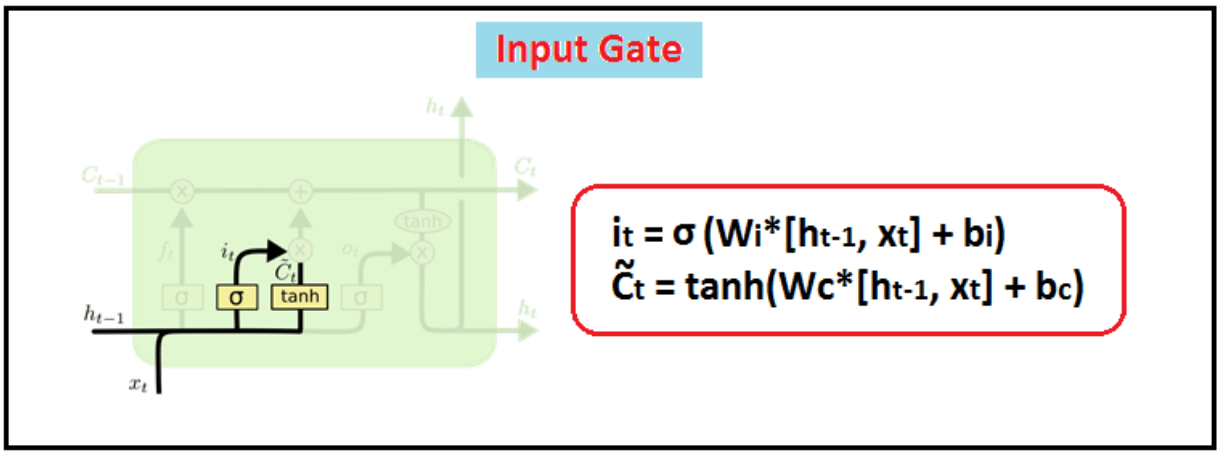
X_t as “Michael is good at Pop.”

- The Forget gate realizes that there might be a change in the relevant context after encountering its first full-stop.
- Then compares it with the current Input X_t
- It is really important to know that the next sentence talks about Michael and not John, Thus the information about John is forgotten.

The next step is to decide what new information is going to be stored in the cell state.

Seeing the above example, it is understood that the relevant context has changed and now the information of Michael needs to be retained/stored.

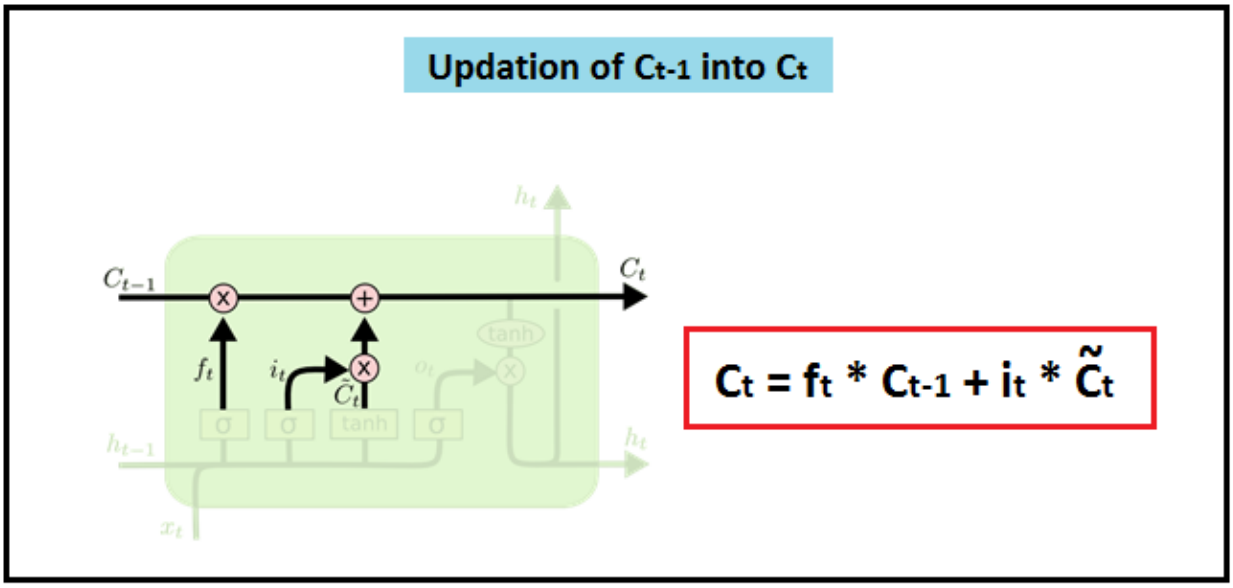
Input Gate:



The functionality of the Input gate is that it supports adding information with regards to the changed context.

The Input gate determines the value from the input to be used to modify the memory, this is done by the Sigmoid function that decides which values to let through 0, 1, and the Tanh function contributes the weightage to the values which are passed to decide their level of importance ranging from-1 to 1.

Now the model updates the old cell state i.e. C_{t-1} into the new cell state i.e. C_t

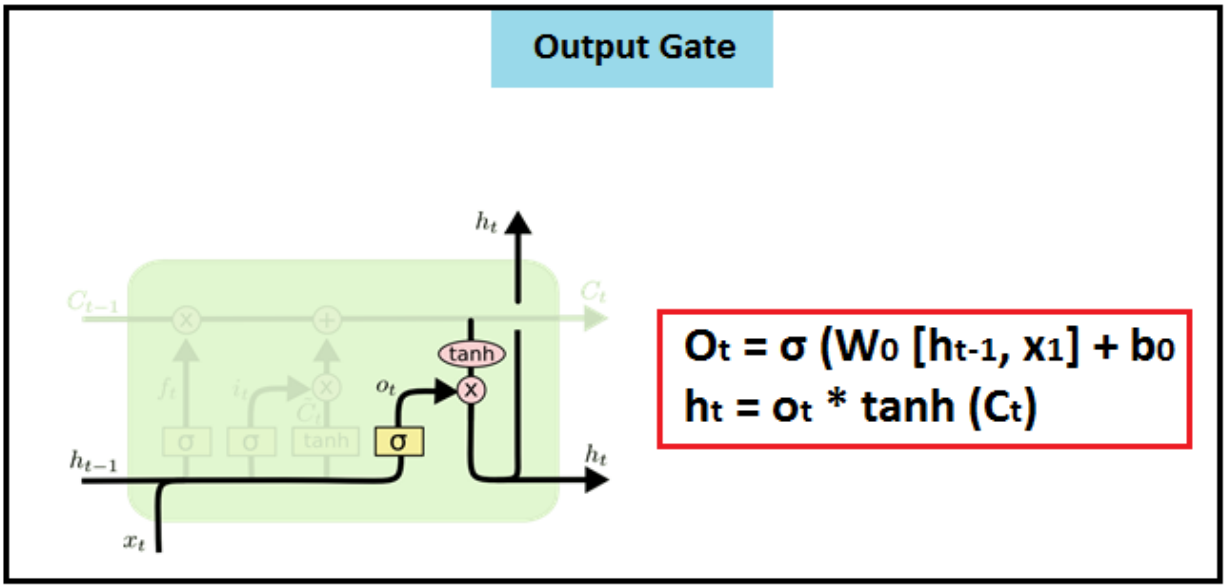


For example, consider the following as the input sentence, “Michael is good at pop singing, yesterday he told me that he is a state-level performer.”

- The input gate analyses and determines the important information.
- **Michael is good at pop singing, he is a state-level performer** is important.
- **Yesterday he told me that** is not important, hence the LSTM model forgets this portion of information and retains only the important portion of information.

Output Gate:

The final layer of the LSTM, the input, and the memory of the block is used to determine the output. Sigmoid function decides which values to let through 0, 1 and Tanh function contributes the weightage to the values which are passed to decide their level of importance ranging from-1 to 1 and multiplied with the output returned by the sigmoid function.



For example, consider the statement, Michael is good at Pop singing, he is a State level performer, _____is the winner of last year Pop singing competition held in the university.

- There could be several choices to fill the blank. However, this final gate will fill it with Michael.

Let us now study how to do the next word prediction practically with Python Keras.

PRACTICAL IMPLEMENTATION OF LSTM USING PYTHON-KERAS:

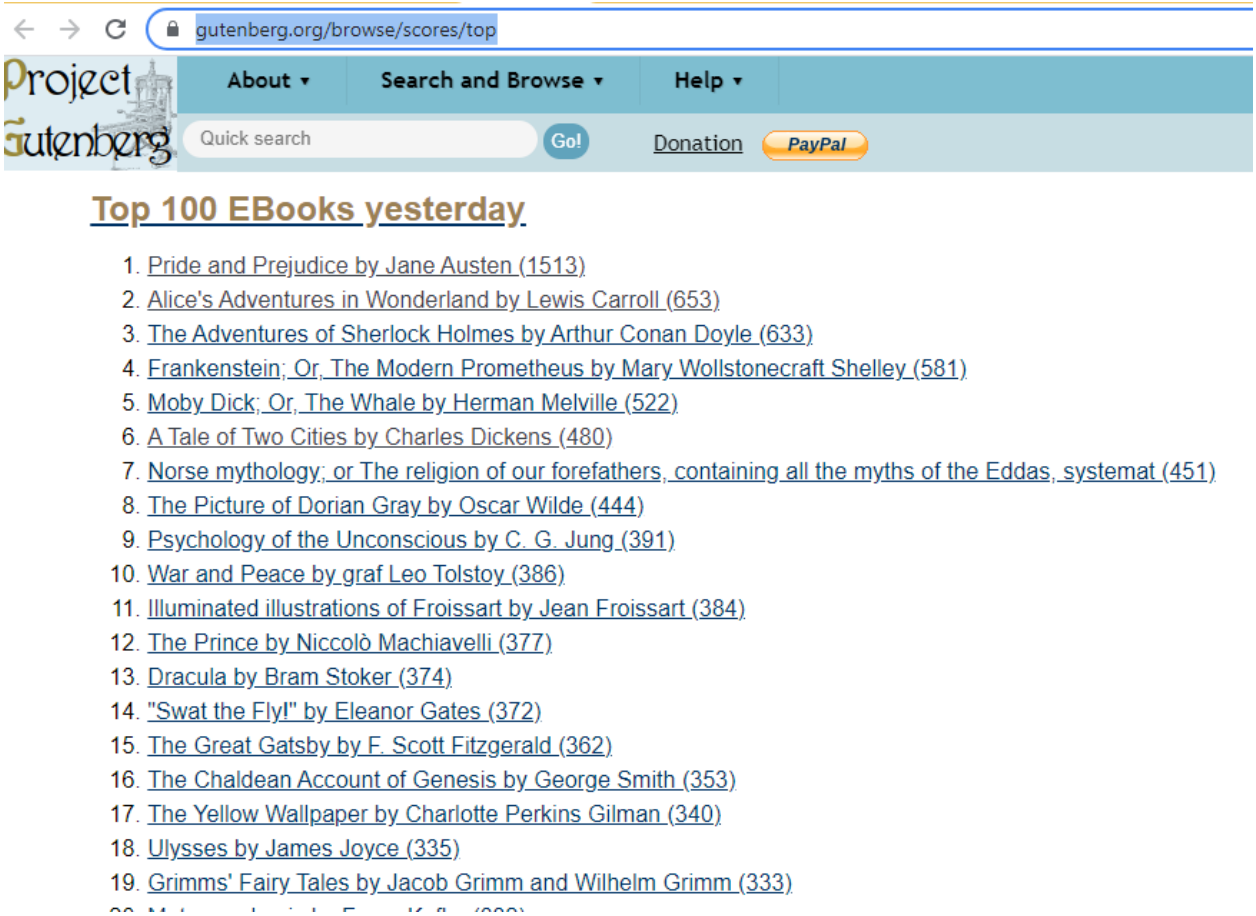
Task: You will use sequential text data and convert the text data into vectors and feed these vectors into the RNN – LSTM network and build a neural network model which will use the preceding 3 words to predict the next word.

Downloading the Dataset:

You can download the dataset from the below URL.

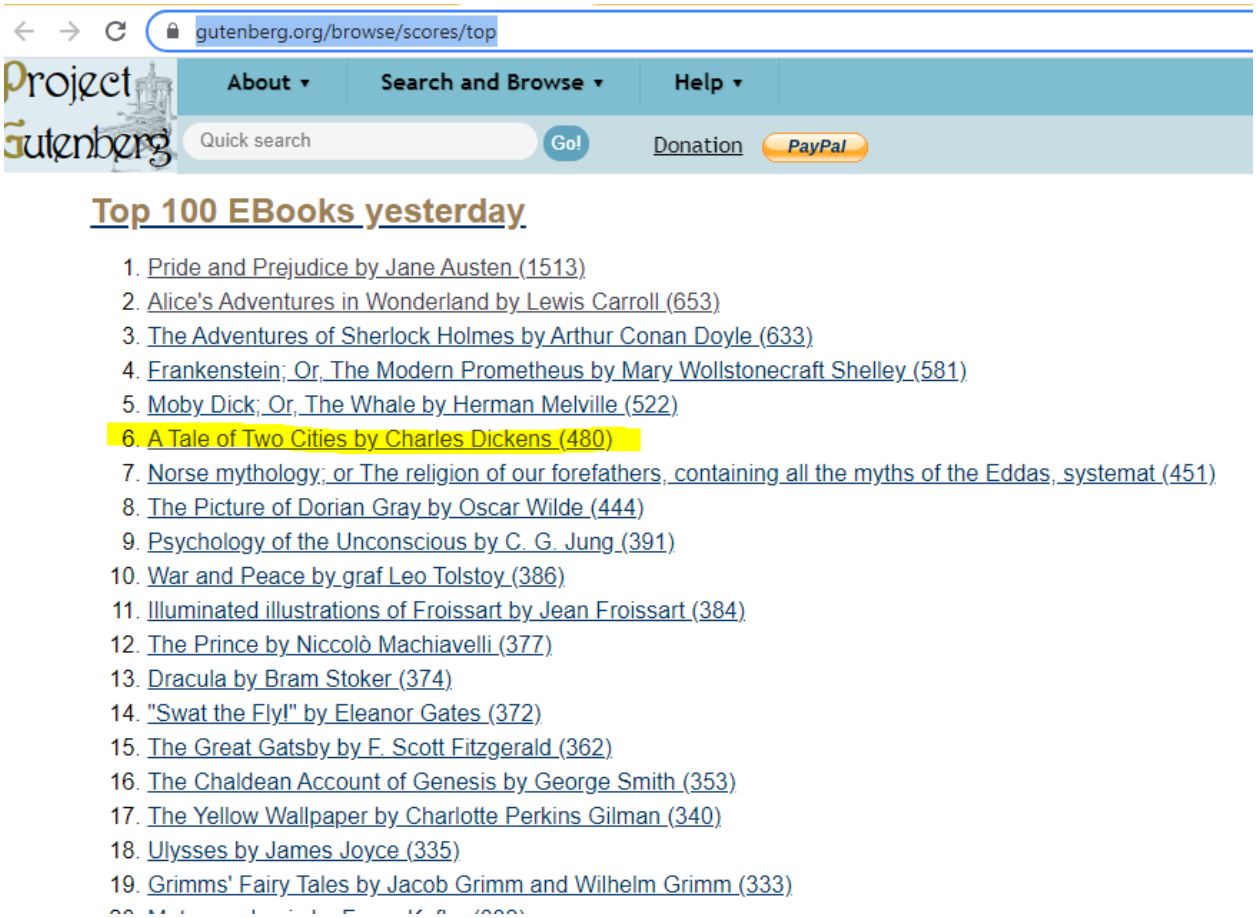
https://www.gutenberg.org/browse/scores/top

Once you click the URL and open the webpage and scroll down a little, you will find the below screen,

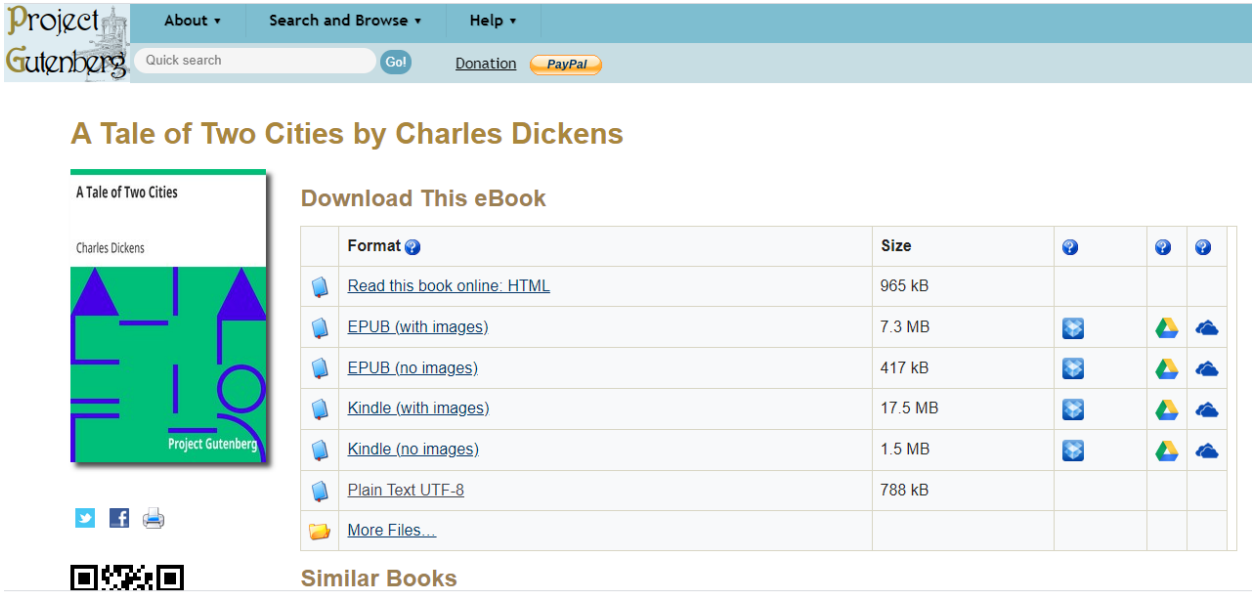


There are multiple text documents available and you can choose any of them to practice, we are using **“A Tale of Two Cities by Charles Dickens”** for now.

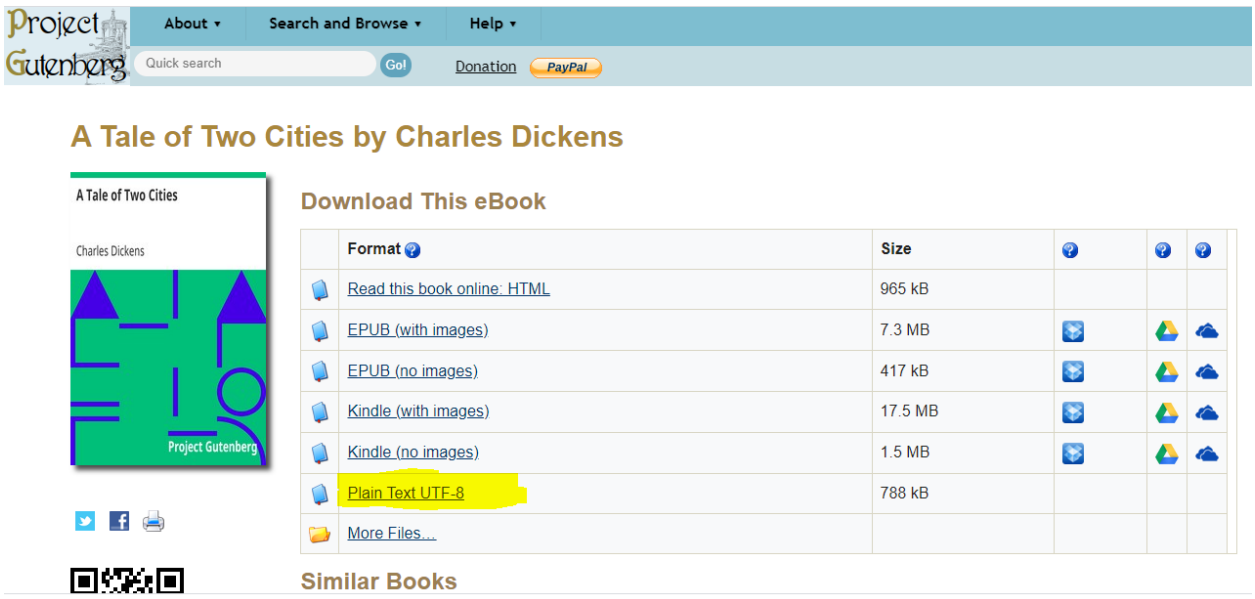
Next, click on **“A Tale of Two Cities by Charles Dickens”,**



Upon clicking on **“A Tale of Two Cities by Charles Dickens”,** it will lead you to the below page,



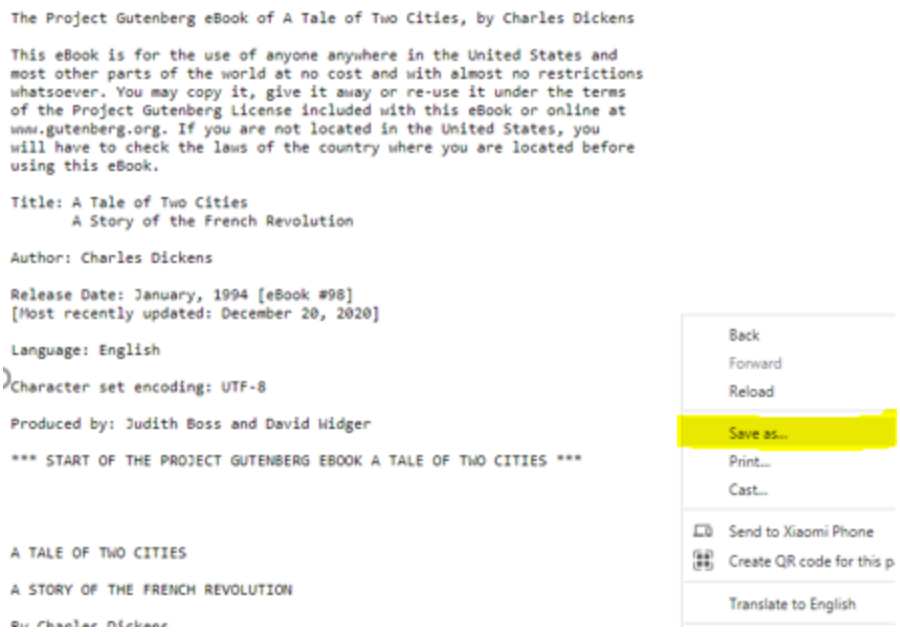
Now, click on "**Plain Text UTF -8",**



After clicking on "Plain Text UTF – 8", you will find the text document opened as below,

```
The Project Gutenberg eBook of A Tale of Two Cities, by Charles Dickens

This eBook is for the use of anyone anywhere in the United States and
most other parts of the world at no cost and with almost no restrictions
whatsoever. You may copy it, give it away or re-use it under the terms
of the Project Gutenberg License included with this eBook or online at
www.gutenberg.org. If you are not located in the United States, you
will have to check the laws of the country where you are located before
using this eBook.

Title: A Tale of Two Cities
       A Story of the French Revolution

Author: Charles Dickens

Release Date: January, 1994 [eBook #98]
[Most recently updated: December 20, 2020]

Language: English

Character set encoding: UTF-8

Produced by: Judith Boss and David Widger

*** START OF THE PROJECT GUTENBERG EBOOK A TALE OF TWO CITIES ***



A TALE OF TWO CITIES

A STORY OF THE FRENCH REVOLUTION

By Charles Dickens


CONTENTS
```

You can Right Click and save the text document into your local drive, you will use this file for practice.



The sequential text data is ready now, let us start implementing step by step using Google Colab. (You may also use your local machine if you are working with a high-end GPU)

Part1: Load the required Packages and dataset.

Step 1:

Load the packages,

```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam
import pickle
import numpy as np
import os
```

Step 2:

Load the Sequential text data which was downloaded earlier into your local drive. i.e. "A Tale of Two Cities by Charles Dickens"

```
# Load the text file into colab workspace
from google.colab import files
uploaded = files.upload()
```

Upon executing the above code you will find the below button i.e. "Choose Files" appearing on the screen, you can click on it to upload the Text file/Data from your local drive.

```
Choose Files  A tale of two cities.txt
```

- **A tale of two cities.txt**(text/plain) - 807231 bytes, last modified: 7/26/2021 - 100% done

```
Saving A tale of two cities.txt to A tale of two cities (3).txt
```

Part 2: Data Preprocessing

You have already studied that RNN will take inputs in the form of vectors, thus we need to convert the text data into Vectors, to do that we first need to eliminate unwanted stuff from the textual dataset.

You will do the pre-processing step-by-step,

- **Open the text file in reading mode,**
- **Store the file in the form of a list,**
- **Convert the list into a string,**
- **Replace unnecessary stuff with space,**
- **Finally, remove the unwanted spaces.**

Step 3:

Open File in reading mode,

```
# Open the file in read mode
file = open("A tale of two cities.txt", "r", encoding = "utf8")
```

Step 4:

Store file in the form of a list,

```
# store file in list
lines = []
for i in file:
    lines.append(i)
```

Step 5:

Covert list to String,

```
# Convert list to string
data = ""
for i in lines:
  data = ' '. join(lines)
```

Step 6:

Replace unnecessary stuff with spaces,

```
#replace unnecessary stuff with space
data = data.replace('\n', '').replace('\r', '').replace('\ufeff', '').replace('“','').replace('”','')
```

Step 7:

Remove unnecessary spaces,

```
#remove unnecessary spaces
data = data.split()
data = ' '.join(data)
```

Step 8:

Let us check the initial 500 fields from the pre-processed data,

```
# Validate the preprocessed data by checking the initial 500 fields
data[:500]
```

You can see that the unwanted stuff is eliminated,

```
'The Project Gutenberg eBook of A Tale of Two Cities, by Charles Dickens This eBook is for the use of anyone anywhere in the U
nited States and most other parts of the world at no cost and with almost no restrictions whatsoever. You may copy it, give it
away or re-use it under the terms of the Project Gutenberg License included with this eBook or online at www.gutenberg.org. If
you are not located in the United States, you will have to check the laws of the country where you are located before usi'
```

Step 9:

Check the length of the data, to know the number of words in the dataset.

```
# Check the length of the data
len(data)
```

```
767189
```

Part 3: Tokenization of data – Converting text to sequences

Step 10:

You already studied that you need vectors to feed into the network, the input data is in the form of text, you will need to convert them into vectors now before building the LSTM model.

With Keras "Tokenizer" you can easily convert the sequential text data into vectors, you will do it step-by-step.

- **Initiate a tokenizer object,**
- **Save the tokenizer object into a pickle file,**
- **Finally, convert text data into vectors.**

```
# Initiate a tokenizer object
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data]) # Fit the preprocessed data into the above initiated Tokenizer object

# save the tokenizer into a pickle file
pickle.dump(tokenizer, open('token.pkl', 'wb'))

# Convert the textual data into vectors i.e. neumerical values
sequence_data = tokenizer.texts_to_sequences([data])[0]
```

While saving the tokenizer object, **'wb'** means write binary used to handle the file, **token.pkl** is giving the file name.

'texts_to_sequences' converts text data into vectors, i.e. numerical sequences.

Step 11:

Now check if the Text is converted to vectors or not by calling the first 15 fields,

```
sequence_data[:15]
```

This line of code will return the below output,

```
[1, 209, 192, 1027, 3, 5, 1907, 3, 77, 1908, 32, 189, 3242, 31, 1027]
```

Each number represents a word.

Step 12:

Let us check the length of the vectors now,

```
# Check the length of the sequential data
len(sequence_data)
```

```
140776
```

The length of the sequential data is lesser than the actual data because each unique value will get a numerical representation, thus all the repeated values will get the label of the same numerical value, also white spaces and punctuations are omitted.

Step 13:

Check how many unique words are present in the data,

```
# let us check how many unique words are present in the data
vocab_size = len(tokenizer.word_index) + 1  # use +1 because 0 will be used for padding
print(vocab_size)
```

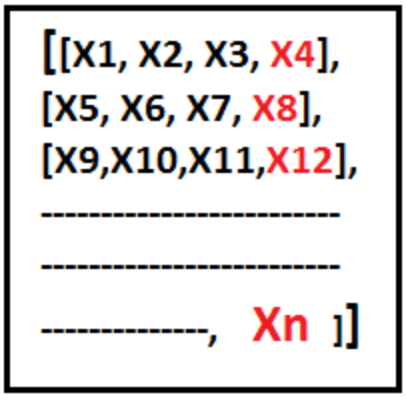
This piece of code will return the total number of unique words in the dataset,

```
10202
```

Step 14:

The goal is to predict the next word when the preceding three words are given as the input.

To achieve the goal you will need the vectors in the below form of a matrix with 4 columns in each row where the first 3 values are the input and the fourth value is the output in each row.



You can do it with the help of the below chunk of code,

```
# Initiate a blank list
sequences = []

# Iterate the sequence data into the above blank list
for i in range(3, len(sequence_data)):
    words = sequence_data[i-3:i+1] # Will use previous 3 words to predict the next word
    sequences.append(words)
   # Print the length of sequences
print("The Length of sequences are: ", len(sequences))
sequences = np.array(sequences) # Convert the sequences into array
sequences[:10] # print the initial 10 sequences
```

It will return the below output,

```
The Length of sequences are:  140773
array([[   1,  209,  192, 1027],
       [ 209,  192, 1027,    3],
       [ 192, 1027,    3,    5],
       [1027,    3,    5, 1907],
       [   3,    5, 1907,    3],
       [   5, 1907,    3,   77],
       [1907,    3,   77, 1908],
       [   3,   77, 1908,   32],
       [  77, 1908,   32,  189],
       [1908,   32,  189, 3242]])
```

In the above matrix of sequences, the first 3 words are the inputs and the fourth word is the output in each line.

Step 15:

Separate the dependent and independent data from the above step, as the first 3 words are the independent features and the 4th word is the dependent feature.

```
# Split the dependent and independent features
# initiate empty lists for x and y features
X = []
y = []
# Append the first 3 words into the x feature and the last word into the y feature
for i in sequences:
    X.append(i[0:3])
    y.append(i[3])
# Convert the above appended lists into arrays
X = np.array(X)
y = np.array(y)
```

Step 16:

Convert the vectors into a binary class matrix and check the first 5 fields,

```
# Convert the vectors into binary class matrix
y = to_categorical(y, num_classes=vocab_size)
# print the first 5 fields
y[:5]
```

It will return the first 5 fields of the categorical variable.

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

Part 4: Build the model

Step 17:

Initialize the neural network model and create the layers of the neural network.

```
# Initiate a Sequential Deep Network
model = Sequential()
# Create an embedding layer which takes 3 arguments as input
model.add(Embedding(vocab_size, 10, input_length=3))
# Vocab_size = input dim which is the size of vocabulary in the text data,
# 10 = Output dim which is the size of the vector space in which words will be embedded
# add an LSTM layer
model.add(LSTM(1000, return_sequences=True))
# 1000 is the dimensionality given to the output space,
# Return_sequences=True is to create next LSTM layer
# add the second LSTM layer
model.add(LSTM(1000))
# add a Dense Layer
model.add(Dense(1000, activation="relu"))
# add a second Dense Layer
model.add(Dense(vocab_size, activation="softmax"))
```

Step 18:

Check the layers created in the above step, you can have the overview by executing the below line of code,

```
# Check the layers created above by printing the summary of the model
model.summary()
```

Output:

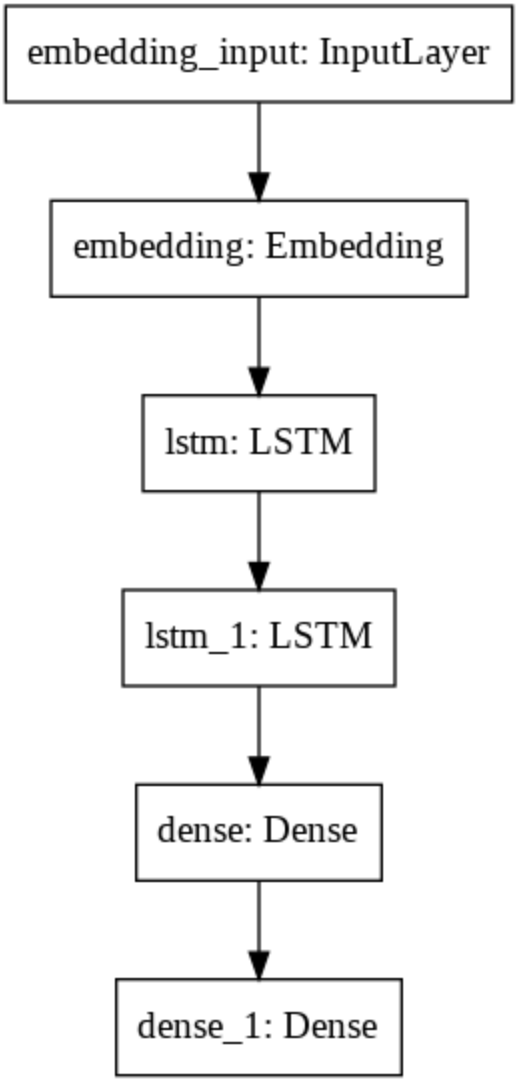
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 3, 10)             102020
_________________________________________________________________
lstm (LSTM)                  (None, 3, 1000)           4044000
_________________________________________________________________
lstm_1 (LSTM)                (None, 1000)              8004000
_________________________________________________________________
dense (Dense)                (None, 1000)              1001000
_________________________________________________________________
dense_1 (Dense)              (None, 10202)             10212202
=================================================================
Total params: 23,363,222
Trainable params: 23,363,222
Non-trainable params: 0
```

Step 19:

You can visualize the outline of the layers in the model using the below line of code,

```
# Vizualise the model
from tensorflow import keras
from keras.utils.vis_utils import plot_model

keras.utils.plot_model(model, to_file='plot.png', show_layer_names=True)
```

Output:



Step 20:

Compile and fit the model,

```
# Compile and fit the model i.e train the model
from tensorflow.keras.callbacks import ModelCheckpoint

checkpoint = ModelCheckpoint("next_words.h5", monitor='loss', verbose=1, save_best_only=True)
model.compile(loss="categorical_crossentropy", optimizer=Adam(learning_rate=0.001))
model.fit(X, y, epochs=70, batch_size=64, callbacks=[checkpoint])
```

The optimizer used is Adam, the learning rate is 0.001, the loss function used is categorical cross-entropy, the number of epochs is 70, and the batch size is 64.

Step 21:

Define a simple function to predict the next word,

```
from tensorflow.keras.models import load_model
import numpy as np
import pickle
```

```
# Load the model and tokenizer
model = load_model('next_words.h5')
tokenizer = pickle.load(open('token.pkl', 'rb'))

def Predict_Next_Words(model, tokenizer, text):

  sequence = tokenizer.texts_to_sequences([text])
  sequence = np.array(sequence)
  preds = np.argmax(model.predict(sequence))
  predicted_word = ""

  for key, value in tokenizer.word_index.items():
      if value == preds:
          predicted_word = key
          break

  print(predicted_word)
  return predicted_word
```

Step 22:

Create a loop that creates a blank box, where you can input 3 words and the created model will predict the next word,

```
while(True):
  text = input("Enter your line: ")

  if text == "0":
      print("Execution completed.....")
      break

  else:
      try:
          text = text.split(" ")
          text = text[-3:]
          print(text)

          Predict_Next_Words(model, tokenizer, text)

      except Exception as e:
        print("Error occurred: ",e)
        continue
```

Upon executing the above loop, Python will return the below output,

You can now key in any three words in a sequence from the original data downloaded and the model will output the 4th word for you,

```
Enter your line: yet received through
['yet', 'received', 'through']
any
Enter your line: body burned alive
['body', 'burned', 'alive']
because
Enter your line: the worst of
['the', 'worst', 'of']
the

Enter your line: wonderful fact to
['wonderful', 'fact', 'to']
reflect
Enter your line: door of the
['door', 'of', 'the']
doctor’s
Enter your line: wine, and had
['wine,', 'and', 'had']
a
Enter your line: wine and had
['wine', 'and', 'had']
stained
Enter your line:
```

PROGRAMMING ASSIGNMENT:

Download the Pride and Prejudice text file and create an LSTM model which will take 3 sequential words as input and predict the next word.

Hint: You can replicate all the steps shown in the above practical implementation, except for steps 2 and 3, where you will need to replace **“A Tale of Two Cities by Charles Dickens”** with **“Pride and Prejudice by Jane Austen”.**

SUMMARY:

LSTM – Long Short Term Memory is a special type of Recurrent Neural network, It uses its internal memory to remember the previous data and it makes robust predictions, it works best in the use-cases where the input is sequential data especially, Word predictions, Time-series forecasting. LSTM Model is a chain-like structure, its architecture is comprised of 4 modules i.e. Memory Cell, Forget Gate, Input Gate, and the Output Gate. These modules mainly depend on the context of information, these 4 modules function in such a manner that when the context of information is changed the network forgets a portion of previous information and adds the changed/new information to the model. It is similar to the functioning that of a human brain.

ASSESSMENT

Choose the appropriate option:

1) LSTM models are dependent on :

A. Context of information.
B. Sequential Data as input.
C. Both of the above.

2) How many Modules are present in LSTM Architecture?

A. 4
B. 6
C. 8
D. 10

3) LSTM is a special kind of:

A. Feed-Forward Network.
B. Recurrent Neural Network.
C. Both of the above.

4) The Forget Gate Discovers the:

A. Information to be discarded.
B. Information not to be discarded.

5) LSTM Model escapes:

A. The Problem of vanishing Gradient.
B. The Problem of Long Term Dependencies.
C. Both of the above.

Fill in the Spaces:

1) In the problem of long term dependencies, the gap between the relevant information and the point where it is needed is _____.
2) The LSTM architecture mainly comprises of _____, _____, _____, _____.
3) _____ And _____ are the components of the Memory Cell.
4) _____ supports in adding information with regards to the changed context.
5) LSTM is a special kind of _____ Neural Networks.

True or False:

1) **To implement an LSTM model using python you need to install and import Keras.**

A. True
B. False

2) **In LSTM models, the input data is in the form of Vectors**

A. True
B. False

3) **Tanh function decides which values to let through 0, 1**

A. True
B. False

4) **The Keras-Tokenizer helps in converting Text to Sequences.**

A. True
B. False

5) **While making word predictions using LSTM, it is recommended to retain unwanted spaces in the data.**

A. True
B. False

Solutions for Assessment

Choose the appropriate options answers

1) C
2) A
3) B
4) A
5) C

Fill in the spaces with appropriate answers

1) Large.
2) Memory Cell, Forget Gate, Input Gate, Output Gate.
3) Point-wise operation and addition operation.
4) Input-Gate.
5) Recurrent.

True or False

1) True.
2) True.
3) False.
4) True.
5) False.

CHAPTER 4:

PRACTICAL IMPLEMENTATION OF LSTM (LONG SHORT TERM MEMORY) TIME SERIES USE-CASE.

Until Now, you have studied about Perceptron, Feed-Forward Neural Networks, Recurrent Neural Networks and RNN –LSTM, You have also studied that RNN – LSTM network uses its internal memory to dynamically produce outputs when the input data is sequential information, in the previous chapter you had a glance at how to practically work on word predictions using LSTM network in python, now the time has arrived to study more about LSTM practical implementation use-cases.

In this chapter you will explicitly deal with time series forecasting of stock prices using LSTM models. Upon completing this chapter you will be able to forecast the stock prices of any stock based on their previous day's prices, you can bifurcate this use-case into:

- Univariate forecasting.
- Multivariate forecasting.

UNIVARIATE FORECASTING:

Under Univariate forecasting you will be using a single feature to forecast the prices for the subsequent days. In simple words it is the extrapolation of values based on previous prices.

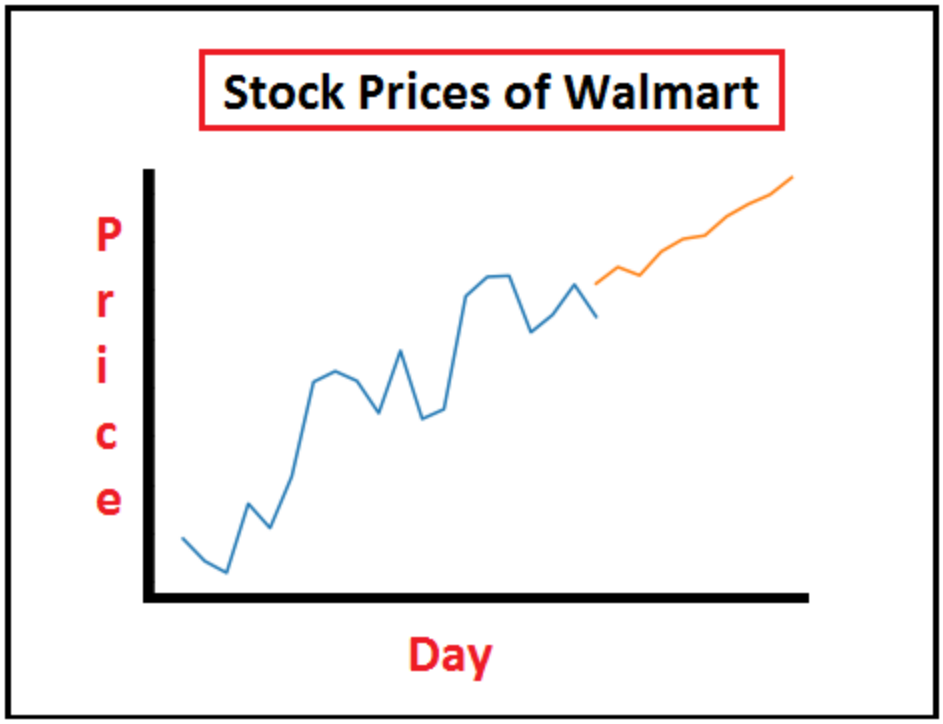
MULTIVARIATE FORECASTING:

Under Multivariate forecasting you will be suing multiple features to forecast the stock prices for the subsequent days.

PRECISE UNDERSTANDING OF TIME SERIES FORECASTING OF STOCK PRICES:

Suppose, consider the previous 20 days stock prices of a particular stock you are interested in, and then use LSTM model to forecast its prices for the next 10 days,

You might be interested in an output like something below,

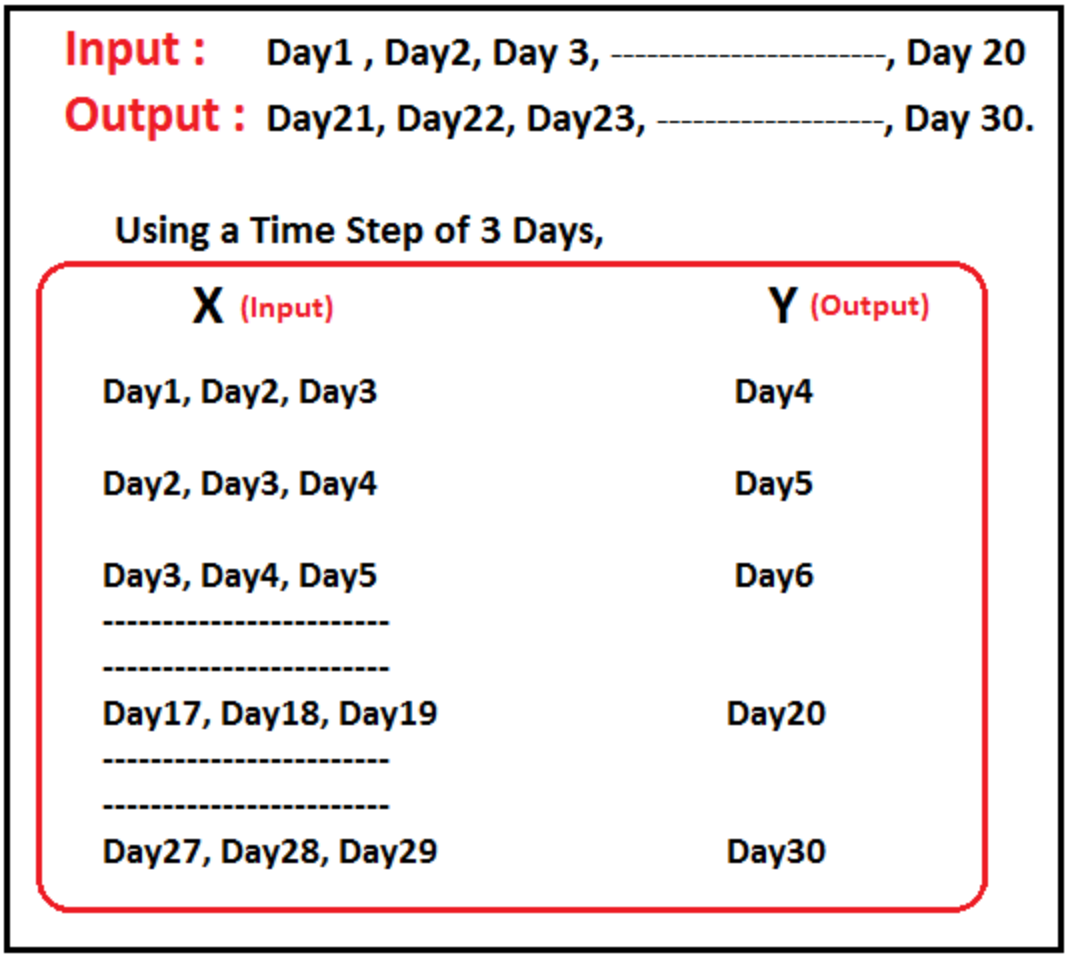


In the above image, you can see that the stock prices forecast of Walmart is done using the LSTM model based on the past prices, you can observe that the line chart has got 2 colors to it i.e. Blue and Orange, the blue portion of the line in the past 20 days closing prices and the orange portion of the line is the forecasted closing prices for the next 10 days.

METHODOLOGY BEHIND THE PRE-PROCESSING OF INFORMATION DURING LSTM TIME-SERIES FORECASTING:

By now you might have understood that the LSTM model uses a sequence of information as input i.e., stock prices of **Day1, Day2, Day3,** ----------**, Day 20** are a sequence of information. From theory, you know that the LSTM model uses the previous **time step** information to return the output when the input is a sequence of information, in python we can set the time step manually based on our use-case and requirement, Let us understand more about it, suppose you want to forecast the stock prices using a time step of 3 days, the model will use the preceding 3 days prices as input to return the price of the 4^{th} day and this chain continues until it reaches the final day where the forecast ends, In our use-case, the final day will be day 30 i.e., 20days prices are the input and 10 days is the output to be forecasted, thus day 30 will become the final day.

Let us see how it is done when you use a time step of 3 days,



To explain the information in the image more clearly, consider the first record from the above image, the stock prices of Day1, Day2, Day3 will be the input to derive the stock price on Day4 and this process continues until the final day i.e. day 30.

Now, let us start by forecasting the stock prices of Walmart considering a Univariate Feature.

IMPLEMENTING UNIVARIATE FORECASTING OF STOCK PRICES USING PYTHON:

Problem Statement: Consider the Stock Prices of Walmart between Jul.06.2021 to Aug.02.2021 as per stock prices sourced from Nasdaq.com, you can download the data from the below URL,

https://www.nasdaq.com/market-activity/stocks/wmt/historical

Once you download the data, extract the information from the Close/Last column, while extracting the closing prices of the stock it is important to ensure that the prices are arranged in chronological order, i.e. oldest to newest, if missed to arrange, the model will become void as we need the sequence in the information.

Date	Close/Last	Volume	Open	High	Low
8/2/2021	$142.22	6652841	$142.83	$142.87	$141.67
7/30/2021	$142.55	5444061	$141.20	$142.96	$141.20
7/29/2021	$142.24	3687689	$142.64	$142.85	$142.02
7/28/2021	$142.06	4690487	$142.48	$143.25	$141.66
7/27/2021	$142.64	5131905	$143	$143	$141.96
7/26/2021	$142.63	6172445	$142.36	$143.86	$141.46
7/23/2021	$142.43	5315836	$141.52	$142.72	$141.01
7/22/2021	$141.27	4337765	$141.13	$142.12	$140.80
7/21/2021	$141.17	6197299	$142.50	$142.54	$140.71
7/20/2021	$141.87	6340100	$140.97	$142.47	$140.78
7/19/2021	$141.23	9127720	$141.41	$142.98	$139.77
7/16/2021	$141.56	6031805	$141.70	$142.13	$140.95
7/15/2021	$141.66	5991612	$142	$142.64	$141.09
7/14/2021	$141.55	6270671	$140.71	$141.80	$139.83
7/13/2021	$140.58	6315868	$139.99	$141.18	$139.94
7/12/2021	$140.05	6504976	$140.69	$140.79	$139.33
7/9/2021	$140.30	5061763	$140.41	$140.84	$139.93
7/8/2021	$139.59	7089047	$138.34	$140.65	$138.22
7/7/2021	$139.71	8535819	$139.76	$141.19	$139.36
7/6/2021	$139.94	6093058	$139.90	$140.75	$139.52

Providing the above-referenced data below, arranged in the required order for your ease.

139.94, 139.71, 139.59, 140.3, 140.05, 140.58, 141.55, 141.66, 141.56, 141.23, 141.87, 141.17, 141.27, 142.43, 142.63, 142.64, 142.06, 142.24, 142.55, 142.22

If you are not keen on downloading the data from the referenced URL, you can simply use the above numbers to build the model which is exactly sourced from the URL.

Now that you have the data for 20 days, let's build an LSTM model which forecasts the close price for the next 10 Days using python.

PART 1: IMPORT AND LOAD REQUIRED PACKAGES AND LIBRARIES.

Step 1:

Import the necessary packages, you will mainly need Tensorflow, Numpy, Keras, and matplotlib.

```
import numpy as np
import tensorflow
import keras
```

```
tensorflow.__version__ # Check the Tensorflow Version
```

```
'2.5.0'
```

```
# Load the packages required to bulid the neural network
from keras.models import Sequential
from keras.layers import Dense, LSTM, Flatten
```

PART 2: DATA INPUT AND PRE-PROCESSING

Step 2:

Load the Data and assign a label to it.

```
univariate_data=[139.94, 139.71, 139.59, 140.3, 140.05, 140.58, 141.55, 141.66, 141.56,
                 141.23, 141.87, 141.17, 141.27, 142.43, 142.63, 142.64, 142.06, 142.24,
                 142.55, 142.22]
```

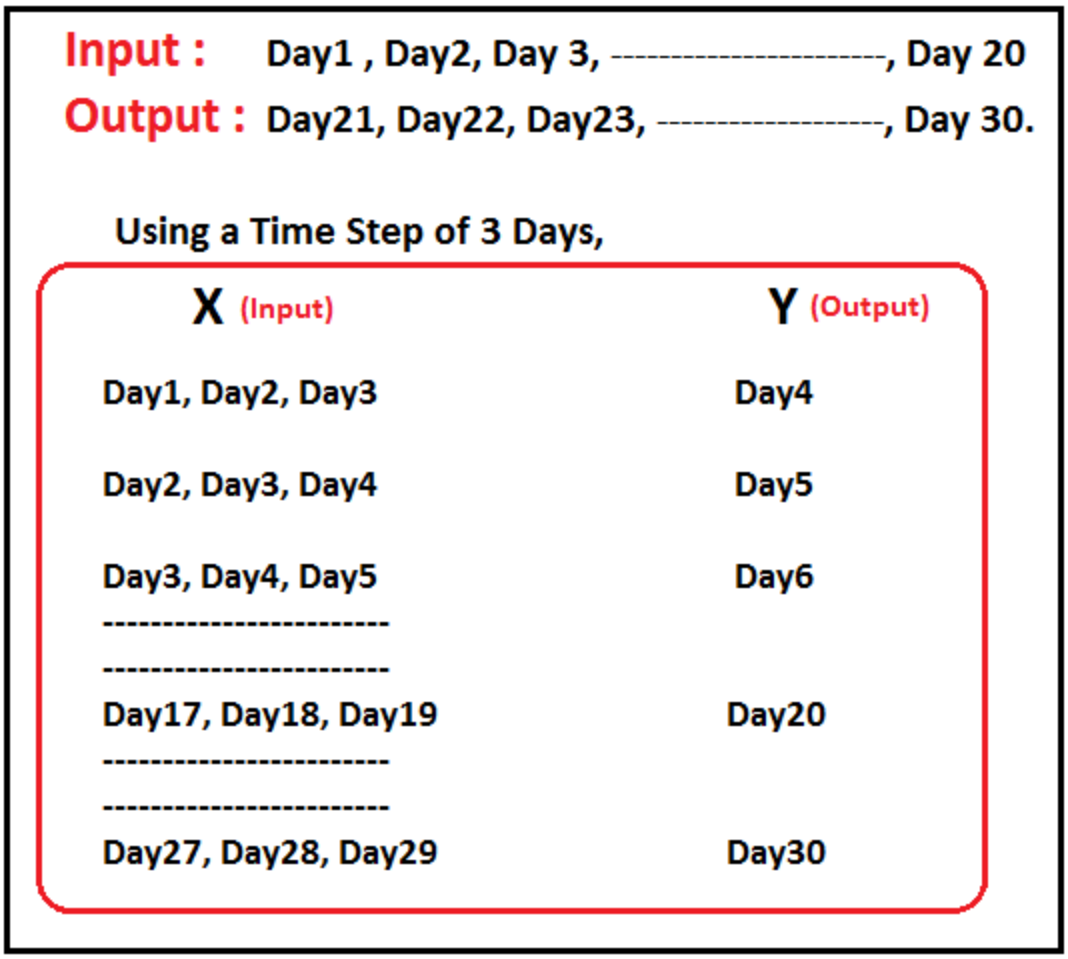
Step 3:

Create a Time Step of 3 Days, such that the model will use the preceding 3 days' information from the sequence as input to return the next day's output.

```
# Create the number of time steps i.e Based on previous 3 days data
# the model shall predict the next day's closing price
# Therefore, the time step to be created is 3

time_steps =3
```

Step 4:

Create Dependent and Independent Features, Remember the split needed shall be based on the Time Step of 3 Days, Reiterating the below image, Where X is the Independent Feature and Y is the Dependent Feature.



You can do the splitting of Dependent and Independent Features in the next step, however, in the current step you will only be preparing the Independent and dependent features by simply defining a function as below.

```
# preparing dependent and independent features
def prepare_data(univariate_data, n_features):
  x, y =[],[]
  for i in range(len(univariate_data)):
    # find the end of this pattern
    end_ix = i + n_features
    # check if we are beyond the sequence
    if end_ix > len(univariate_data)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = univariate_data[i:end_ix], univariate_data[end_ix]
    x.append(seq_x)
    y.append(seq_y)
  return np.array(x), np.array(y)
```

Step 5:

Split the data into Dependent and Independent Features.

```
# split the data into samples
x, y = prepare_data(univariate_data, time_steps)
```

Step 6:

Check if the data is Split as per the need or not

```
print(x),print(y)
```

This Line of Code will return the below output,

```
[[139.94 139.71 139.59]
 [139.71 139.59 140.3 ]
 [139.59 140.3  140.05]
 [140.3  140.05 140.58]
 [140.05 140.58 141.55]
 [140.58 141.55 141.66]
 [141.55 141.66 141.56]
 [141.66 141.56 141.23]
 [141.56 141.23 141.87]
 [141.23 141.87 141.17]
 [141.87 141.17 141.27]
 [141.17 141.27 142.43]
 [141.27 142.43 142.63]
 [142.43 142.63 142.64]
 [142.63 142.64 142.06]
 [142.64 142.06 142.24]
 [142.06 142.24 142.55]]
[140.3  140.05 140.58 141.55 141.66 141.56 141.23 141.87 141.17 141.27
 142.43 142.63 142.64 142.06 142.24 142.55 142.22]
```

You have successfully split the data as needed.

Step 7:

Check the shape of the data

```
x.shape
```

```
(17, 3)
```

Step 8:

To proceed further, we need a 3 Dimensional Data input, let us reshape it from [samples, time steps] into [samples, time steps, features] using the below line of code.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
x = x.reshape((x.shape[0], x.shape[1], n_features))
```

PART 3: BUILDING THE MODEL

Step 9:

Define and build the LSTM model.

```
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', return_sequences=True, input_shape=(time_steps, n_features)))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(x, y, epochs=100, verbose=1)
```

Step 10:

Visualize the loss based on the above fit model,

```
# For Vizualising the loss, assigning the model a variable
final_model = model.fit(x, y, epochs=100, verbose=0)
```

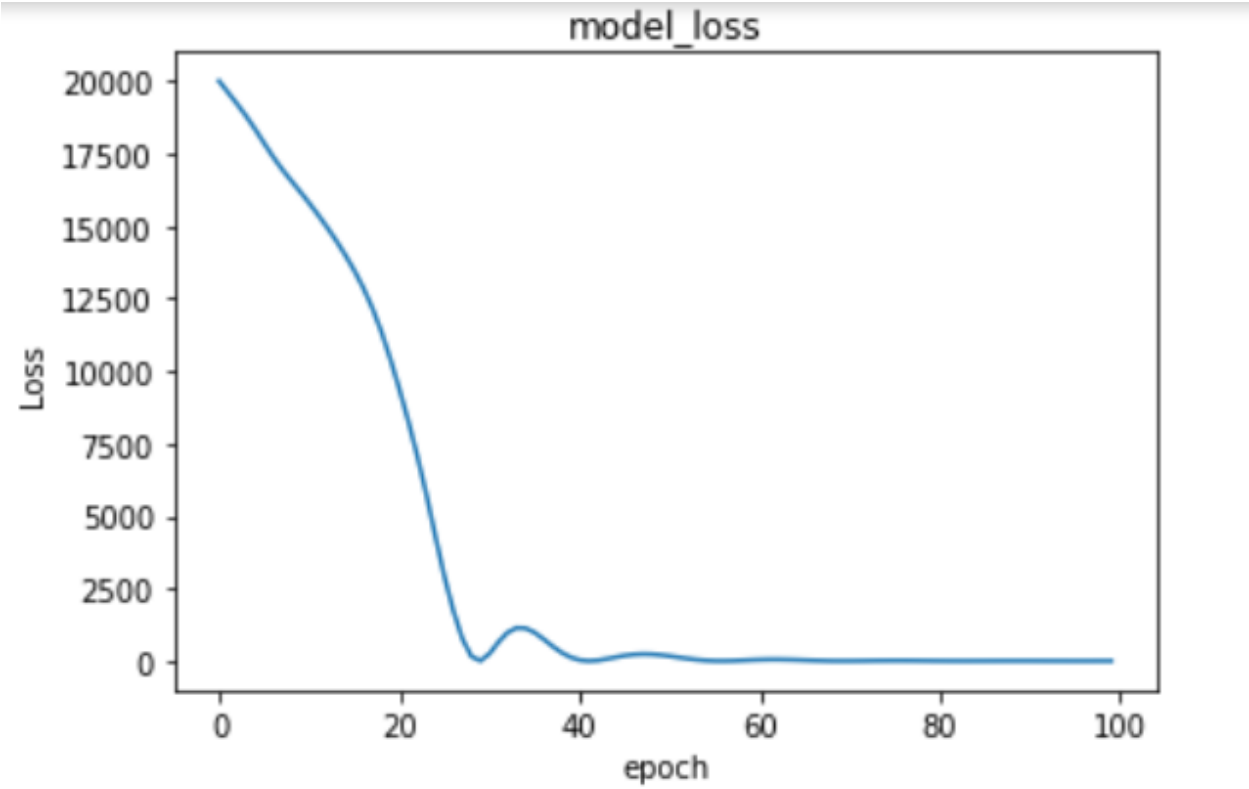
Check the keys present in final_model,

```
final_model.history.keys()
```

```
dict_keys(['loss'])
```

```
import matplotlib.pyplot as plt

plt.plot(final_model.history['loss'])
plt.title('model_loss')
plt.ylabel('Loss')
plt.xlabel('epoch')
```

Output from Step 10:



PART 4: FORECASTING THE VALUES FOR THE NEXT 10 DAYS.

Step 11:

Demonstrate prediction for the next 10 days.

```
x_input = np.array([142.24, 142.55, 142.22])
temp_input=list(x_input)
lst_output=[]
i=0
while(i<10):
```

```
  if(len(temp_input)>3):
      x_input=np.array(temp_input[1:])
      print("{} day input {}".format(i,x_input))
      #print(x_input)
      x_input = x_input.reshape((1, time_steps, n_features))
      #print(x_input)
      yhat = model.predict(x_input, verbose=0)
      print("{} day output {}".format(i,yhat))
      temp_input.append(yhat[0][0])
      temp_input=temp_input[1:]
      #print(temp_input)
      lst_output.append(yhat[0][0])
      i=i+1
```

```
else:
    x_input = x_input.reshape((1, time_steps, n_features))
    yhat = model.predict(x_input, verbose=0)
    print(yhat[0])
    temp_input.append(yhat[0][0])
    lst_output.append(yhat[0][0])
    i=i+1
```

Step 12:

Print the result

```
print(lst_output)
```

Output from Step 12:

```
[142.56053]
1 day input [142.55       142.22       142.56053162]
1 day output [[142.73035]]
2 day input [142.22       142.56053162 142.73034668]
2 day output [[142.64426]]
3 day input [142.56053 142.73035 142.64426]
3 day output [[142.8893]]
4 day input [142.73035 142.64426 142.8893 ]
4 day output [[143.01833]]
5 day input [142.64426 142.8893  143.01833]
5 day output [[143.05328]]
6 day input [142.8893  143.01833 143.05328]
6 day output [[143.24922]]
7 day input [143.01833 143.05328 143.24922]
7 day output [[143.37595]]
8 day input [143.05328 143.24922 143.37595]
8 day output [[143.47383]]
9 day input [143.24922 143.37595 143.47383]
9 day output [[143.64964]]
[142.56053, 142.73035, 142.64426, 142.8893, 143.01833, 143.05328, 143.24922, 143.37595, 143.47383, 143.64964]
```

Step 13:

Print the forecasted values,

```
lst_output
```

```
[142.56053,
 142.73035,
 142.64426,
 142.8893,
 143.01833,
 143.05328,
 143.24922,
 143.37595,
 143.47383,
 143.64964]
```

The values in the above list are the forecasted values of close price for the next 10 days.

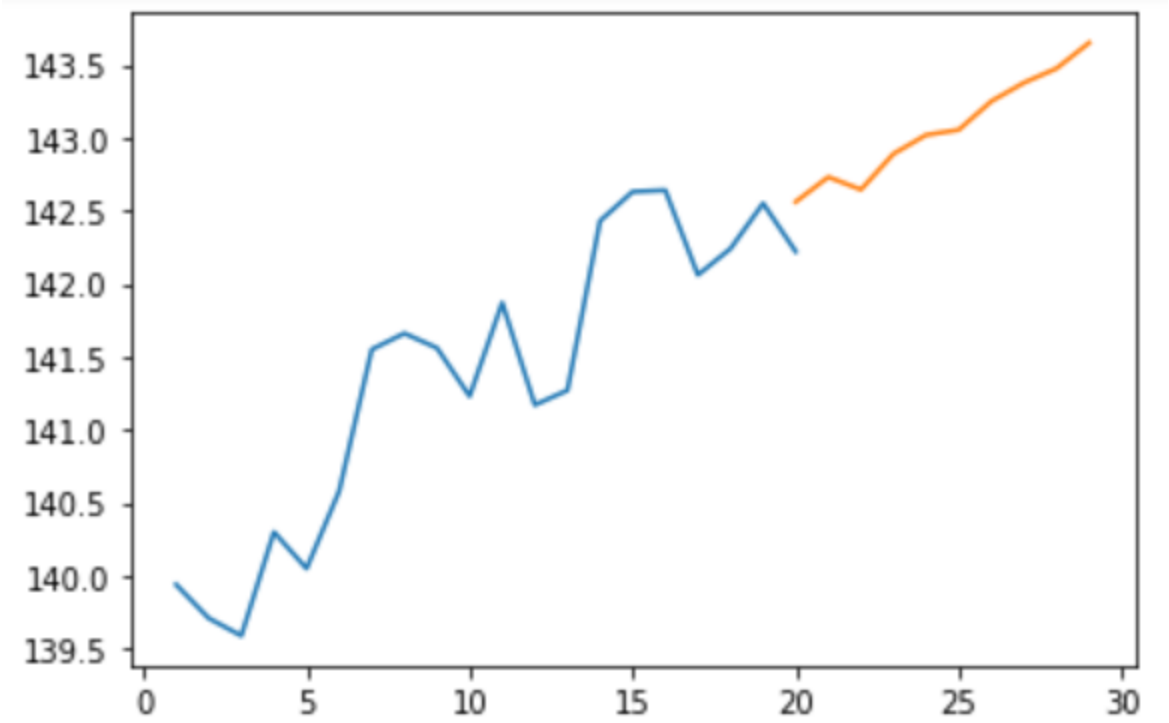
PART 5: VISUALIZING THE OUTPUT

Step 14:

```
day_new=np.arange(1,21)
day_pred=np.arange(20,30)
```

```
plt.plot(day_new,univariate_data)
plt.plot(day_pred,lst_output)
```

Final Output:



You have completed the implementation of Univariate Forecasting of stock prices, you can select any other stock of your choice and replicate all of these steps to forecast the prices. Similarly, instead of forecasting the Closing price, you can do it for the open price, high, Low as well.

IMPLEMENTING MULTIVARIATE FORECASTING OF STOCK PRICES USING PYTHON:

You have seen the practical implementation of Univariate Stock price forecasting on a beginner level, now you will study Multivariate stock price forecasting at a more advanced level, let's take it forward step by step from stock prices to be used as input, building the LSTM model and deriving the forecast.

Problem statement: Pick any stock data of your choice (choosing **GameStop Corporation Common Stock -GME** for this use-case implementation), based on the past 5 Years' data, Forecast the next 30 Days Stock 'Close' price, use a time step of 150 days.

You can download the stock data from the below URL:

https://www.nasdaq.com/market-activity/stocks/gme/historical

Once you open the URL, Select '5Y' as shown in the below image,

GME > GME HISTORICAL DATA

GME Historical Data

1M 6M YTD 1Y 5Y MAX DOWNLOAD DATA

Date	Close/Last	Volume	Open	High	Low
08/06/2021	$151.77	1,352,689	$154.59	$156.5	$150.31
08/05/2021	$153.44	2,412,123	$148	$157.6	$145.22
08/04/2021	$146.8	2,393,484	$152.73	$157.79	$145.691
08/03/2021	$152.75	14,450,620	$156.74	$158.89	$148.2
08/02/2021	$157.65	2,532,931	$162	$163.5899	$155.06
07/30/2021	$161.12	2,377,118	$165	$167.22	$158.87
07/29/2021	$164.86	2,239,995	$170.6	$173.89	$164.51
07/28/2021	$169.12	4,100,845	$175.72	$177.12	$164.27
07/27/2021	$178.54	1,214,775	$183	$185	$176.6601
07/26/2021	$183.94	1,260,593	$180.36	$186.04	$178.76
07/23/2021	$180.36	1,316,168	$181	$181.6	$173.84
		1,412,988	$185.3	$187.69	$176.15

After selecting ‘5Y’ click on download data, such that the data will be downloaded into your local drive in .CSV format.

GME > GME HISTORICAL DATA

GME Historical Data

1M 6M YTD 1Y 5Y MAX

DOWNLOAD DATA

Date	Close/Last	Volume	Open	High	Low
08/06/2021	$151.77	1,352,689	$154.59	$156.5	$150.31
08/05/2021	$153.44	2,412,123	$148	$157.6	$145.22
08/04/2021	$146.8	2,393,484	$152.73	$157.79	$145.691
08/03/2021	$152.75	14,450,620	$156.74	$158.89	$148.2
08/02/2021	$157.65	2,532,931	$162	$163.5899	$155.06
07/30/2021	$161.12	2,377,118	$165	$167.22	$158.87
07/29/2021	$164.86	2,239,995	$170.6	$173.89	$164.51
07/28/2021	$169.12	4,100,845	$175.72	$177.12	$164.27
07/27/2021	$178.54	1,214,775	$183	$185	$176.6601
07/26/2021	$183.94	1,260,593	$180.36	$186.04	$178.76
07/23/2021	$180.36	1,316,168	$181	$181.6	$173.84
		1,412,988	$185.3	$187.69	$176.15

Once the data is downloaded, ensure it is sorted in chronological order, because when you forecast the data you will need the oldest data in the beginning and the latest data in the end, if you miss sorting the data the entire forecast will become void as it will give a wrong interpretation.

Upon sorting the oldest records shall be in the top,

Date	Close	Volume	Open	High	Low
8/9/2016	$29.63	2247634	$29.76	$30.22	$29.50
8/10/2016	$30.15	1750279	$29.60	$30.23	$29.60
8/11/2016	$31.11	2464282	$30.53	$31.17	$30.43
8/12/2016	$31.35	2146046	$31	$31.73	$30.82
8/15/2016	$31.73	1430096	$31.31	$32.08	$31.31
8/16/2016	$31.17	1586892	$31.66	$31.95	$31.16
8/17/2016	$30.82	1762462	$30.87	$30.94	$30.50
8/18/2016	$31.17	1580156	$30.93	$31.19	$30.57
8/19/2016	$31.72	2164838	$31.06	$31.92	$30.89
8/22/2016	$31.46	1671844	$32	$32.10	$31.26
8/23/2016	$32.09	2839347	$32.03	$32.67	$31.83
8/24/2016	$31.71	2713812	$32.13	$32.41	$31.62
8/25/2016	$32.16	6091551	$31.67	$32.25	$31.28
8/26/2016	$28.74	14492510	$30.22	$30.51	$28.69
8/29/2016	$28.97	4405544	$28.75	$28.97	$28.34
8/30/2016	$28.49	2859089	$28.89	$28.96	$28.27
8/31/2016	$28.39	2827455	$28.48	$28.95	$28.31
9/1/2016	$28.03	2094881	$28.47	$28.58	$27.97
9/2/2016	$28.39	1690754	$28.21	$28.48	$28.04
9/6/2016	$28.37	1944117	$28.38	$28.67	$28.25
9/7/2016	$28.13	1516245	$28.03	$28.23	$27.89
9/8/2016	$27.78	1728754	$27.97	$28.11	$27.75

The latest entries/records shall be at the bottom,

7/15/2021	$166.82	4298589	$160	$171.99	$158.01
7/16/2021	$169.04	3287929	$170.15	$179.47	$166.30
7/19/2021	$173.49	2436934	$163.30	$176	$161.22
7/20/2021	$191.18	3101090	$173.90	$193.64	$172.42
7/21/2021	$185.81	2229184	$187.79	$195.51	$182.11
7/22/2021	$178.85	1412988	$185.30	$187.69	$176.15
7/23/2021	$180.36	1316168	$181	$181.60	$173.84
7/26/2021	$183.94	1260593	$180.36	$186.04	$178.76
7/27/2021	$178.54	1214775	$183	$185	$176.66
7/28/2021	$169.12	4100845	$175.72	$177.12	$164.27
7/29/2021	$164.86	2239995	$170.60	$173.89	$164.51
7/30/2021	$161.12	2377118	$165	$167.22	$158.87
8/2/2021	$157.65	2532931	$162	$163.59	$155.06
8/3/2021	$152.75	14450620	$156.74	$158.89	$148.20
8/4/2021	$146.80	2393484	$152.73	$157.79	$145.69
8/5/2021	$153.44	2412123	$148	$157.60	$145.22
8/6/2021	$151.77	1352689	$154.59	$156.50	$150.31

As the data is ready now, let's start implementing the model with python,

PART 1: LOAD THE REQUIRED PACKAGES AND DATASET

Step 1:

Import the required packages,

```
# Importing the required packages
import numpy as np
import pandas as pd
import tensorflow as tf
import keras
```

Step 2:

Load the downloaded GME stock prices dataset into python, and print the top rows to have a glance at the records,

```
# Loading the Dataset
df=pd.read_csv('/content/drive/MyDrive/GME.csv')
df.head() # View the first 5 records of the dataset
```

	Date	Close	Volume	Open	High	Low
0	8/9/2016	$29.63	2247634	$29.76	$30.22	$29.50
1	8/10/2016	$30.15	1750279	$29.60	$30.23	$29.60
2	8/11/2016	$31.11	2464282	$30.53	$31.17	$30.43
3	8/12/2016	$31.35	2146046	$31	$31.73	$30.82
4	8/15/2016	$31.73	1430096	$31.31	$32.08	$31.31

Note: The original downloaded column label for 'Close' was 'Close/Last', as a part of ethical practice and ease of work, to remove the special characters from the feature name, changed it to 'Close'.

Step 3:

Check the summary of the data, ensure there are no null values,

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1258 entries, 0 to 1257
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   Date    1258 non-null   object
 1   Close   1258 non-null   object
 2   Volume  1258 non-null   int64 
 3   Open    1258 non-null   object
 4   High    1258 non-null   object
 5   Low     1258 non-null   object
dtypes: int64(1), object(5)
memory usage: 59.1+ KB
```

As per the summary in the above image, there are 1258 records in the dataset and there are no null values, thus we can proceed further.

Step 4:

As per the project goal, you need to forecast the Close price of the stock, let us do the preliminary preprocessing of the values in this feature, i.e. you need to remove the special characters '$' and ',' before proceeding further,

You will do this in steps:

- Ensure the Data type of the 'Close' feature into a string,
- Remove the $ symbol,
- Remove the commas,
- Finally, convert the data type into a float before proceeding further.

Ensure the data type as a string object,

```
# Ensure the Data type of Close as String
df['Close']=df['Close'].astype(str)
```

Remove the $ symbol,

```
# Removing the $ Symbol from the Close Column
df['Close']=df['Close'].str[1:]
```

Remove commas,

```
# Removing Commas from the Close/Last Column
df['Close'] = df['Close'].str.replace(',','')
```

Validate if $ and commas are removed from the 'Close' feature or not, using Head and Tail Methods.

```
df.head()
```

	Date	Close	Volume	Open	High	Low
0	8/9/2016	29.63	2247634	$29.76	$30.22	$29.50
1	8/10/2016	30.15	1750279	$29.60	$30.23	$29.60
2	8/11/2016	31.11	2464282	$30.53	$31.17	$30.43
3	8/12/2016	31.35	2146046	$31	$31.73	$30.82
4	8/15/2016	31.73	1430096	$31.31	$32.08	$31.31

```
df.tail() # View if the Commas are removed
```

	Date	Close	Volume	Open	High	Low
1253	8/2/2021	157.65	2532931	$162	$163.59	$155.06
1254	8/3/2021	152.75	14450620	$156.74	$158.89	$148.20
1255	8/4/2021	146.80	2393484	$152.73	$157.79	$145.69
1256	8/5/2021	153.44	2412123	$148	$157.60	$145.22
1257	8/6/2021	151.77	1352689	$154.59	$156.50	$150.31

You can notice that the $ and ‘,’ symbols are successfully eliminated from the data.

Convert the data type into ‘float’ before proceeding further,

```
# Convert the required column into Float datatype
df['Close']=df['Close'].astype(float)
```

The preliminary preprocessing of the ‘Close’ feature is completed.

Step 5:

Reset the index for the ‘Close’ feature, so that it will handle the case of any multi-index (in the current data which you are using does not have any multi-index), including this step here only for your future references. Just in case, you find any input data with multi-index this step will help, for more information on this you can refer to the below URL,

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.reset_index.html

```
df1=df.reset_index()['Close']
df1
```

```
0        29.63
1        30.15
2        31.11
3        31.35
4        31.73
         ...
1253    157.65
1254    152.75
1255    146.80
1256    153.44
1257    151.77
Name: Close, Length: 1258, dtype: float64
```

PART 2: DATA PRE-PROCESSING

Step 6:

Scale the values, as the LSTM model will work best when the input values are scaled, you will use MinMaxScaler and transform the values,

```
from sklearn.preprocessing import MinMaxScaler
scaler=MinMaxScaler(feature_range=(0,1))
df1=scaler.fit_transform(np.array(df1).reshape(-1,1))
```

Validate if the values are scaled or not,

```
print(df1)
```

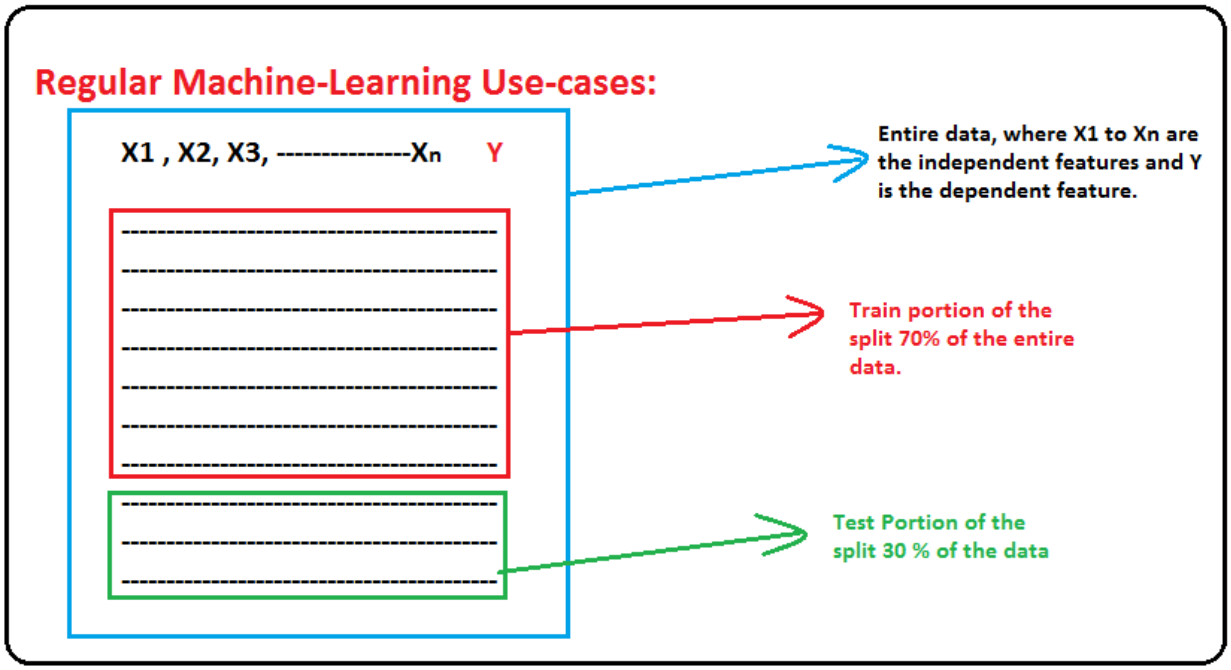
```
[[0.07783354]
 [0.07934206]
 [0.08212701]
 ...
 [0.41774245]
 [0.43700502]
 [0.43216037]]
```

Step 7:

Split the data into train and test.

Before keying in the piece of code to split the data, you need to understand the difference between how the train-test data is split in the regular machine learning use-cases and Time-series forecasting use-case.

In regular machine-learning use-cases, assume a **Train: Test** split of **70: 30**, upon splitting the original dataset will be divided into 2 chunks as shown below,

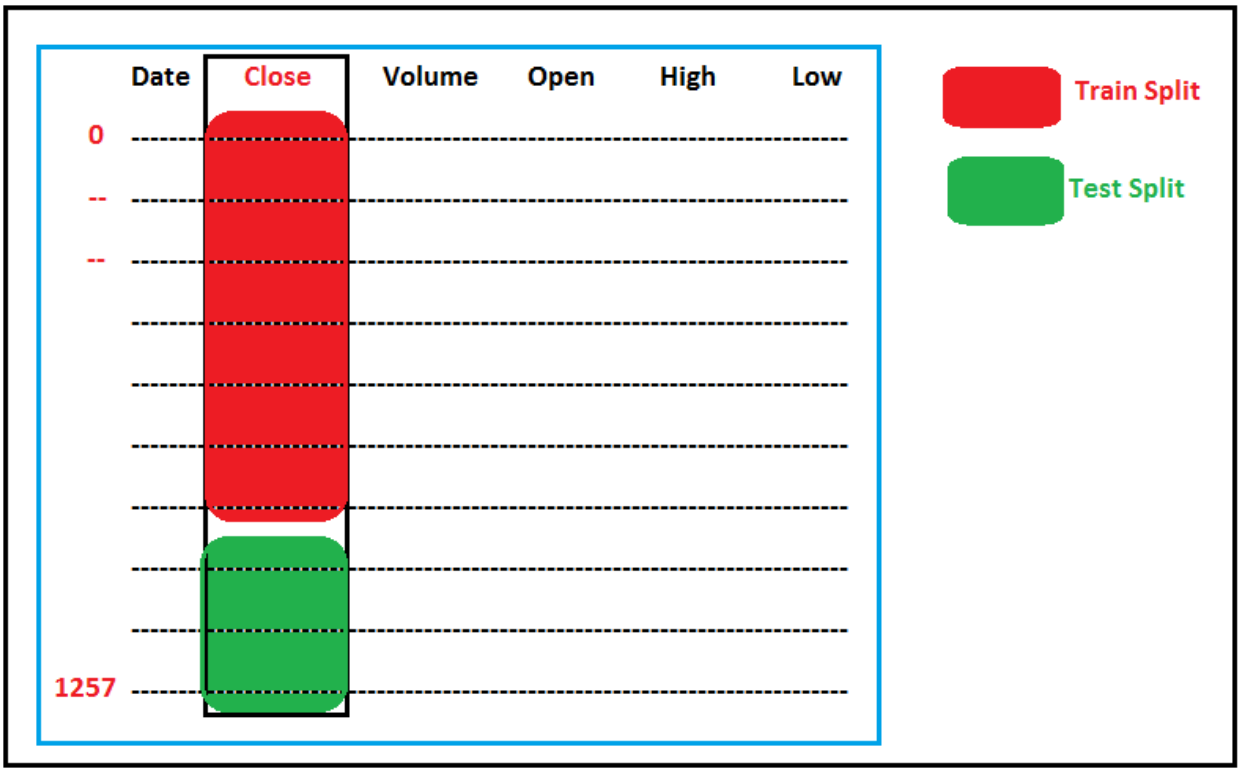


However, note that the split, in reality, happens randomly, not necessarily the first 70 % of the records are Training data and the last 30 % records are the test data, the above image is only an

illustration to make you understand that the train-test split will divide the entire data into 2 chunks.

How is the train test splitting done in time-series forecasting use-case?

In the use-cases of time-series forecasting or cases where the input data is sequential information, the entire data is not split, rather you would pick the feature which you are intending to forecast and then split that particular feature into train and test, in this particular use case you will be picking and splitting the 'Close' feature from the original dataset as shown below,



Use the below code to split the data into train and test,

```
# splitting dataset into train and test split
training_size=int(len(df1)*0.65) # Defining the training data size
test_size=len(df1)-training_size # Test data =  Total length of the data - length of training data
train_data,test_data=df1[0:training_size,:],df1[training_size:len(df1),:1]
```

Validate the split by having a glance at the train and test data sizes created in the above code,

```
training_size,test_size
```

```
(817, 441)
```

You can interpret it as the 0 to 817 indexed data is the Training portion and the remaining values are the Test portion of the data.

Step 8:

See if the data is in the form of a matrix or an array, you need the data in the form of a matrix to proceed further.

```
train_data
```

```
array([[0.01576515],
       [0.0163461 ],
       [0.01642909],
       [0.01731876],
       [0.01775696],
       [0.01640585],
       [0.01492859],
       [0.01512445],
       [0.01506802],
       [0.01269445],
       [0.01341482],
       [0.01440076],
       [0.01267453],
       [0.01332851],
```

Step 9:

As the training data is in the form of an array you need to convert it in the form of a matrix,

```
import numpy
# convert an array of values into a dataset matrix
def create_dataset(dataset, time_step=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-time_step-1):
    a = dataset[i:(i+time_step), 0]   ###i=0, 0,1,2--- 150 as the time step is 150
    dataX.append(a)
    dataY.append(dataset[i + time_step, 0])
  return numpy.array(dataX), numpy.array(dataY)
```

Step 10:

Reshape the data into the form of x=t, t+1, t+2, t+3, and y=t+4 based on time steps of 100 using the below code,

```
# reshape the data
time_step = 150
X_train, y_train = create_dataset(train_data, time_step)
X_test, ytest = create_dataset(test_data, time_step)
```

Step 11:

Check if the data is reshaped based on time-steps of 150 or not,

```
print(X_train)
```

```
[[0.07783354 0.07934206 0.08212701 ... 0.06492414 0.06205216 0.06312553]
 [0.07934206 0.08212701 0.08282324 ... 0.06205216 0.06312553 0.06344463]
 [0.08212701 0.08282324 0.08392562 ... 0.06312553 0.06344463 0.06225523]
 ...
 [0.02135128 0.02196049 0.02117722 ... 0.00983435 0.00873198 0.0084999 ]
 [0.02196049 0.02117722 0.01981376 ... 0.00873198 0.0084999  0.00765861]
 [0.02117722 0.01981376 0.02097415 ... 0.0084999  0.00765861 0.00881901]]
```

```
print(X_train.shape), print(y_train.shape)
```

```
(666, 150)
(666,)
(None, None)
```

You can notice that X_train has got 150 features and y_train has got only 1 feature and the total number of records is 666.

Similarly, check for the test data,

```
print(X_test.shape), print(ytest.shape)
```

```
(290, 150)
(290,)
(None, None)
```

You can notice that X_test has got 150 features and y_test has got only 1 feature and the total number of records is 290.

PART 3: MODEL CREATION AND MODEL EVALUATION

Step 12:

Convert the data into 3 dimensional as LSTM will accept a 3 Dimensional data as input, so let's reshape the data into 3 dimensions by adding a value of 1 in the third dimension as shown below,

```
# reshape input to be [samples, time steps, features] which is required for LSTM
X_train =X_train.reshape(X_train.shape[0],X_train.shape[1] , 1)
X_test = X_test.reshape(X_test.shape[0],X_test.shape[1] , 1)
```

Step 13:

Load the required packages to build the LSTM model,

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
```

Step 14:

Create the LSTM network by stacking multiple layers of LSTM, use 'return_sequences = True' to stack the layers.

```
model=Sequential() # Initializing the sequential model
model.add(LSTM(200,return_sequences=True,input_shape=(150,1))) # 150 = time_step and 1 = 3rd dimension
model.add(LSTM(200,return_sequences=True)) # adding the second LSTM layer
model.add(LSTM(100)) # Adding the third LSTM Layer
model.add(Dense(1)) # adding the final/Output layer
model.compile(loss='mean_squared_error',optimizer='adam')
```

Step 15:

Check the model summary, i.e. the overview of layers created in the above step,

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 150, 200)          161600
_________________________________________________________________
lstm_1 (LSTM)                (None, 150, 200)          320800
_________________________________________________________________
lstm_2 (LSTM)                (None, 100)               120400
_________________________________________________________________
dense (Dense)                (None, 1)                 101
=================================================================
Total params: 602,901
Trainable params: 602,901
Non-trainable params: 0
_________________________________________________________________
```

Step 16:

Fit the model,

```
model.fit(X_train,y_train,validation_data=(X_test,ytest),epochs=150,batch_size=64,verbose=1)
```

Step 17:

Predict the model,

```
# predict the model
train_predict=model.predict(X_train)
test_predict=model.predict(X_test)
```

Step 18:

Transform the values to their original form,

As you have scaled the values before building and fitting the model using MinMaxScaler, you now need to transform the values back into their original form as it is not feasible to measure the performance metrics i.e. RMSE with the scaled-down values.

You can do this by applying the inverse transformation

```
#Transformback to original form
train_predict=scaler.inverse_transform(train_predict)
test_predict=scaler.inverse_transform(test_predict)
```

Step 19:

Calculate RMSE metrics for train data,

```
# RMSE performance metrics for training data
import math
from sklearn.metrics import mean_squared_error
math.sqrt(mean_squared_error(y_train,train_predict))
```

```
15.362150741704122
```

Step 20:

Calculate RMSE metrics for test data,

```
# Test Data RMSE
math.sqrt(mean_squared_error(ytest,test_predict))
```

```
117.86893458040785
```

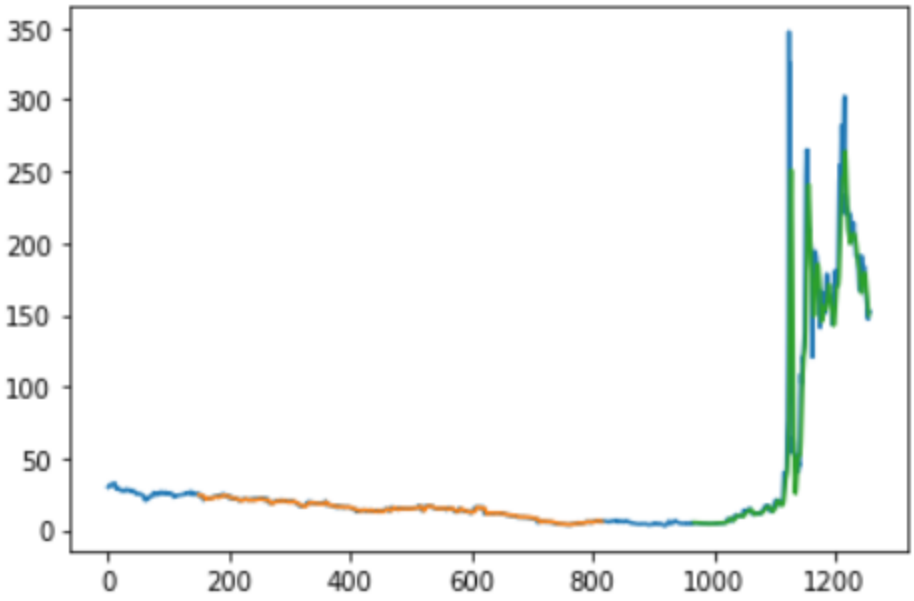
Note that the RMSE values under the price forecasting use-cases will differ based on the value of the stock price,

Step 21:

Plot the graph which shows the line chart of original data, train predict and test predict

```
# Plot the original, Train and Test data
import matplotlib.pyplot as plt

# shift train predictions for plotting
look_back=150 # resembles the time-step of 150
trainPredictPlot = numpy.empty_like(df1)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(train_predict)+look_back, :] = train_predict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(df1)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(train_predict)+(look_back*2)+1:len(df1)-1, :] = test_predict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(df1))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Output:



You can interpret the above chart as, the Blue Line as the original data points, i.e., the original values from the 'Close' feature, Orange line as the Training-predicted data, and the green line as the Test-predicted data.

PART 4: FORECASTING THE NEXT 30 DAYS CLOSING PRICES, BASED ON THE PAST 5 YEARS DATA AND TIME STEP OF 150 DAYS.

Step 22:

From the test dataset, the last day is Aug-06-2021, thus the forecasting needs to be done for the next 30 days starting Aug-07-2021 based on the time step of 150 days you need to predict the value of Close price of Aug-07-2021 and so on for the next 30 days' time frame.

From Step 7, you already know that the Test data length is 441, Thus to predict the data for Aug-07-2021, using the time step of 150 you need to use 291 as shown below and label it as x_input.

```
# Creating x_input variable for the test data
x_input=test_data[291:].reshape(1,-1) # 441 -150 = 291
x_input.shape
```

```
(1, 150)
```

Step 23:

Convert all the values of above created 'x_input' into a list before proceeding further,

```
# Convert the values of X_input into a list and label it as temp_input
temp_input=list(x_input)
temp_input=temp_input[0].tolist()
```

Step 24:

Validate if the values are converted to list or not,

```
temp_input
```

```
[0.04191929447941749,
 0.04226741318789708,
 0.045139392532853706,
 0.04432711554640132,
 0.043195729743842656,
 0.049722955527835,
 0.049751965420208294,
 0.08296829218763599,
 0.10765571059731369,
 0.0948623480606887,
 0.10606016651678223,
 0.10536392909982303,
 0.11670679701778308,
 0.18047054045429492,
 0.2146441936700415,
 0.42116561747555914,
 0.5408894433001654,
 0.5142293522091034,
 0.47814104609671904,
 0.475820254706855,
 0.4822604508137275,
 0.49516985291984567,
 0.5464883525282123,
 0.5309100403237504,
 0.5107191552319341,
 0.5150996489803024,
 0.5254851904499435,
 0.5098198485683618,
 0.4824925299527139,
 0.4701343158016884,
 0.45928461605407445,
 0.4492181834005396,
 0.4350033613762293,
 0.41774245017550987,
 0.4370050187113806,
 0.43216036668503965]
```

You can notice that the values are enclosed within [ ], resembling a list of values.

Step 25:

Demonstrate the prediction for the next 30 days,

```
# demonstrate prediction for next 30 days
from numpy import array

lst_output=[]
n_steps=150
i=0
while(i<30):

    if(len(temp_input)>150):
        #print(temp_input)
        x_input=np.array(temp_input[1:])
        print("{} day input {}".format(i,x_input))
        x_input=x_input.reshape(1,-1)
        x_input = x_input.reshape((1, n_steps, 1))
        #print(x_input)
        yhat = model.predict(x_input, verbose=0)
        print("{} day output {}".format(i,yhat))
        temp_input.extend(yhat[0].tolist())
        temp_input=temp_input[1:]
        #print(temp_input)
        lst_output.extend(yhat.tolist())
        i=i+1
```

```
    else:
        x_input = x_input.reshape((1, n_steps,1))
        yhat = model.predict(x_input, verbose=0)
        print(yhat[0])
        temp_input.extend(yhat[0].tolist())
        print(len(temp_input))
        lst_output.extend(yhat.tolist())
        i=i+1


print(lst_output)
```

In the output, you can see that the values of each day are predicted by using the previous 150 days' data as input.

Step 26:

Check the forecasted stock Close prices for the next 30 days,

```
# Lets check the predicted Close prices for next 30 days
scaler.inverse_transform(lst_output)
```

Output:

```
array([[146.07873553],
       [145.29003433],
       [144.36081687],
       [143.37944258],
       [142.42262113],
       [141.51093995],
       [140.63265681],
       [139.76639326],
       [138.89204474],
       [137.99424259],
       [137.06276503],
       [136.09185909],
       [135.07960369],
       [134.02701586],
       [132.93751658],
       [131.81619105],
       [130.66926481],
       [129.50354896],
       [128.32601898],
       [127.14343461],
       [125.96208301],
       [124.78753225],
       [123.62457988],
       [122.47701671],
       [121.34786303],
       [120.23913239],
       [119.15215003],
       [118.08738852],
       [117.04482729],
       [116.02384998]])
```

Step 27:

Create variables to plot a graph of the forecasted values,

```
day_new=np.arange(1,151) # 150 Days time step
day_pred=np.arange(151,181) # to make a forecast of 30Days i.e 151 to 180 days
```

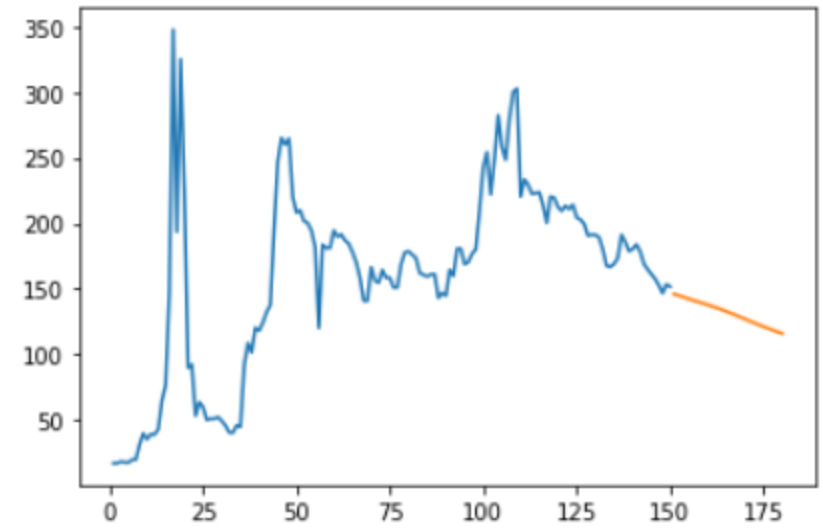
Step 28:

Plot the forecast,

```
plt.plot(day_new,scaler.inverse_transform(df1[1108:])) # 1108 = 1258 - 150 (time-step)
plt.plot(day_pred,scaler.inverse_transform(lst_output))
```

Output:

```
[<matplotlib.lines.Line2D at 0x7f49da03d390>]
```



Interpretation: The Orange portion of the line is the forecast made using the LSTM model for the next 30 days.

SUMMARY:

You have seen the practical implementation of how to forecast Stock prices with univariate data and Multi-variate data, you can follow the same sequence of steps to forecast the prices of any stock of your choice, in this chapter you have seen the forecasting for the 'Close' prices of the stock, however, you can implement the model on the other features as well, such as High, Low, Open, Volumes.

Before concluding, it is important to understand that the forecasting shown in this chapter is purely based on the past available stock data and the theory of RNN-LSTM models, however, in reality, the stock prices may be influenced by other multiple market factors.

PROGRAMMING ASSIGNMENT:

Using the same stock data i.e. GME (5 Years Historical), do the next 30 days forecasting for the 'Low' feature from the original dataset.

Hint: You can replicate all the steps shown above, the only change will be instead of using the 'Close' feature you have to use the 'Last' feature from the data.

CHAPTER 5:

PRACTICAL IMPLEMENTATION OF LSTM (LONG-SHORT TERM MEMORY) TIME SERIES USE-CASE FOR CRYPTO-CURRENCIES.

This Chapter is similar and runs parallel to the previous chapter, the only difference between these two chapters is that in the previous chapter you have experimented with the theory of LSTM time series forecasting on Stocks, here you will be experimenting with the same theory on Crypto-currencies. Upon completing this chapter you will be able to forecast the prices of various cryptocurrencies based on their previous day(s) prices, you can bifurcate this use-case into:

- Univariate forecasting.
- Multivariate forecasting.

UNIVARIATE FORECASTING:

Under Univariate forecasting, you will be using a single feature to forecast the prices for the subsequent days. In simple words, it is the extrapolation of values based on previous prices.

MULTIVARIATE FORECASTING:

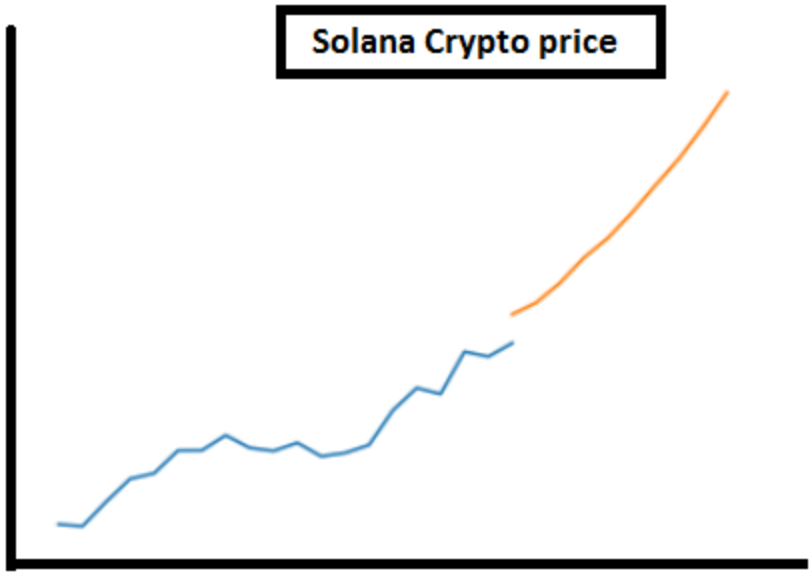
Under Multivariate forecasting, you will be using multiple features to forecast the prices for the subsequent days.

NOTE: Before proceeding further you need to understand that the experiments shown in this chapter are based on the theory of RNN LSTM time-series forecasting, however in the real world, the financial instrument prices may depend on various market and economic factors, thus it is recommended that you do not depend on this theory alone to make decisions about your financial investments.

PRECISE UNDERSTANDING OF TIME SERIES FORECASTING OF CRYPTO-CURRENCY PRICES:

Suppose, consider the previous 20 days Crypto prices of a particular cryptocurrency you are interested in, and then use the LSTM model to forecast its prices for the next 10 days,

You might be interested in an output like something below,



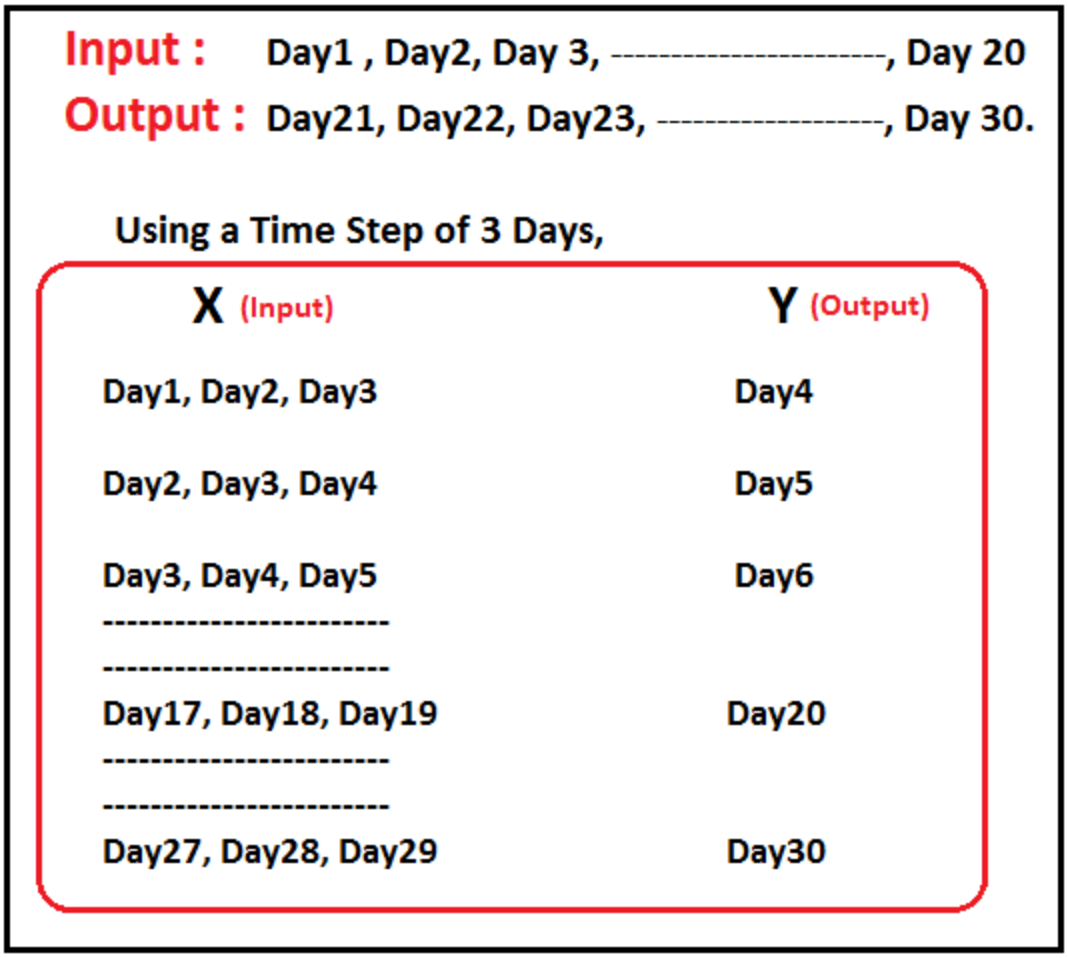
In the above image, you can see that the price forecast of Solana Crypto is done using the LSTM model based on the past prices, you can observe that the line chart has got 2 colors to it i.e. Blue and Orange, the blue portion of the line in the past 20 days closing prices and the orange portion of the line is the forecasted closing prices for the next 10 days.

METHODOLOGY BEHIND THE PRE-PROCESSING OF INFORMATION DURING LSTM TIME-SERIES FORECASTING:

By now you might have understood that the LSTM model uses a sequence of information as input i.e., Cryptocurrency prices of **Day1, Day2, Day3,** ----------**, Day 20** are a sequence of information. From theory, you know that the LSTM model uses the previous **time step** information to return the output when the input is a sequence of information, in python we can set the time step manually based on our use-case and requirement, i.e. Time step is a hyper-parameter, Let us understand more about it, suppose you want to forecast the crypto

prices using a time step of 3 days, the model will use the preceding 3 days prices as input to return the price of the 4th day and this chain continues until it reaches the final day where the forecast ends, In our use-case, the final day will be day 30 i.e., 20days prices are the input and 10 days is the output to be forecasted, thus day 30 will become the final day.

Let us see how it is done when you use a time step of 3 days,



To explain the information in the image more clearly, consider the first record from the above image, the crypto prices of Day1, Day2, Day3 will be the input to derive the price on Day4 and this process continues until the final day i.e. day 30.

Now, let us start by forecasting the prices of Solana Crypto considering a Univariate Feature.

IMPLEMENTING UNIVARIATE FORECASTING OF CRYPTO PRICES USING PYTHON:

Problem Statement: Consider the Prices of Solana between Aug.13.2021 to Sep.01.2021 as per the prices sourced from investing.com, you can download the data from the below URL,

https://in.investing.com/crypto/solana/historical-data

Once you download the data, extract the information from the Price column, while extracting the prices of the Crypto, it is important to ensure that the prices are arranged in chronological order, i.e. oldest to newest, if missed to arrange, the model will become void as we need the sequence in the information.

Date	Price	Open	High	Low	Vol.	Change %
13-Aug-21	44.86	41.12	44.91	40.7	2.28M	9.22%
14-Aug-21	44.14	44.84	44.86	42.77	1.55M	-1.59%
15-Aug-21	53.48	44.15	54.63	43.37	6.26M	21.15%
16-Aug-21	62.12	53.65	68.95	52.4	15.01M	16.16%
17-Aug-21	64.16	62.02	75.04	59.13	14.96M	3.28%
18-Aug-21	72.8	63.92	80.26	59.83	15.02M	13.47%
19-Aug-21	72.85	72.94	75.25	68.52	7.82M	0.06%
20-Aug-21	78.59	72.8	79.95	70.92	5.71M	7.88%
21-Aug-21	73.83	78.68	81.81	72.68	5.13M	-6.05%
22-Aug-21	72.64	73.83	77.5	71.41	2.86M	-1.61%
23-Aug-21	75.75	72.74	76.59	71.57	4.15M	4.28%
24-Aug-21	70.63	75.63	79.34	68.62	6.79M	-6.76%
25-Aug-21	71.9	70.52	72.7	66.28	5.07M	1.81%
26-Aug-21	74.88	72.01	78.15	66.28	6.56M	4.14%
27-Aug-21	87.99	74.86	88.88	72.9	8.80M	17.51%
28-Aug-21	96.51	87.98	97.81	85.63	5.76M	9.68%
29-Aug-21	94.2	96.25	97.77	90.81	4.17M	-2.39%
30-Aug-21	110.3	94	116.4	93.68	11.11M	17.04%
31-Aug-21	108.4	109.9	130	103.4	16.51M	-1.70%
1-Sep-21	113.5	108.1	114.7	106	15.55M	4.70%

Providing the above-referenced data below, arranged in the required order for your ease.

44.858, 44.144, 53.48, 62.122, 64.159, 72.801, 72.845, 78.587, 73.83, 72.639, 75.749, 70.626, 71.904, 74.88, 87.99, 96.51, 94.2, 110.25, 108.38, 113.47

If you are not keen on downloading the data from the referenced URL, you can simply use the above numbers to build the model which is exactly sourced from the URL.

Now that you have the data for 20 days, let's build an LSTM model which forecasts the close price for the next 10 Days using python.

PART 1: IMPORT AND LOAD REQUIRED PACKAGES AND LIBRARIES.

Step 1:

Import the necessary packages, you will mainly need Tensorflow, Numpy, Keras, and matplotlib.

```
import numpy as np
import tensorflow
import keras
```

```
tensorflow.__version__ # Check the Tensorflow Version
```

```
'2.5.0'
```

```
# Load the packages required to bulid the neural network
from keras.models import Sequential
from keras.layers import Dense, LSTM, Flatten
```

PART 2: DATA INPUT AND PRE-PROCESSING

Step 2:

Load the Data and assign a label to it.

```
univariate_data=[44.858, 44.144, 53.48, 62.122, 64.159, 72.801, 72.845, 78.587, 73.83, 72.639,
                 75.749, 70.626, 71.904, 74.88, 87.99, 96.51, 94.2, 110.25, 108.38, 113.47]
```

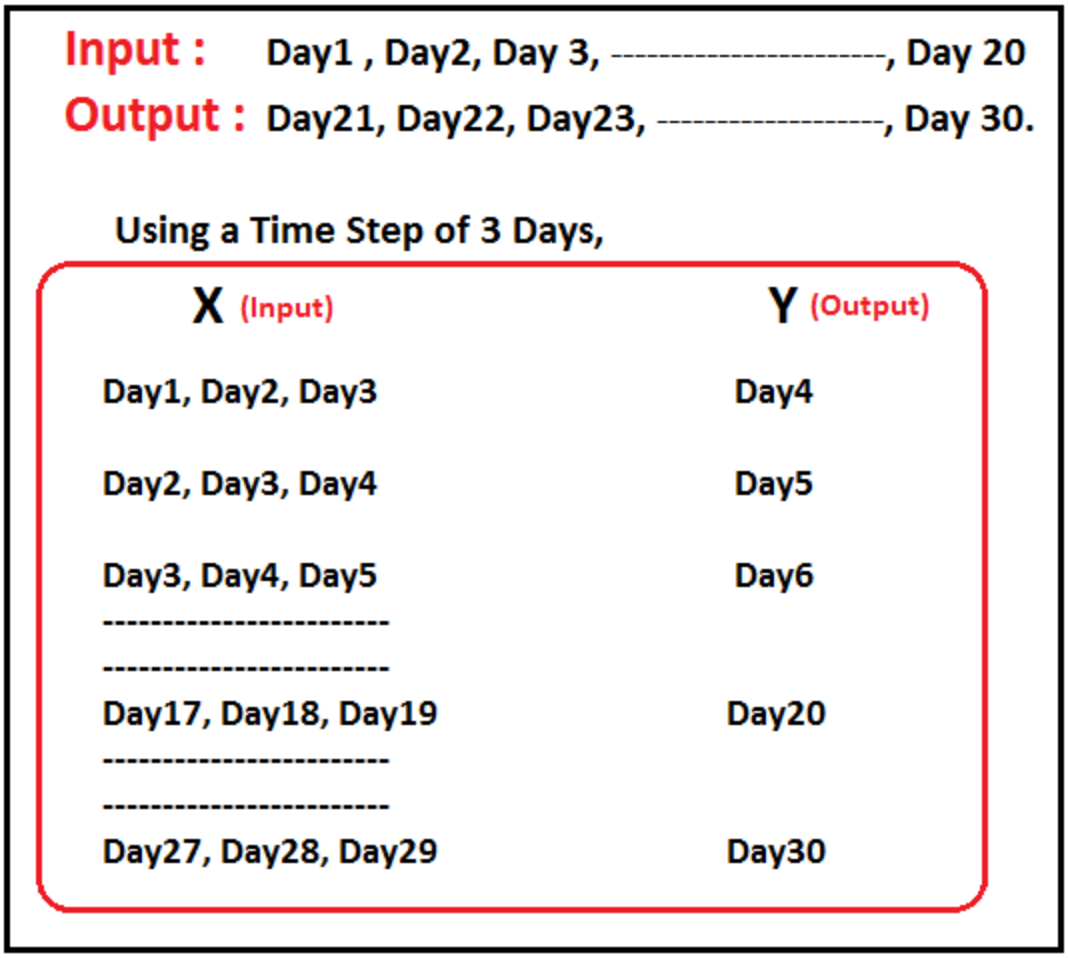
Step 3:

Create a Time Step of 3 Days, such that the model will use the preceding 3 days' information from the sequence as input to return the next day's output.

```
# Create the number of time steps i.e Based on previous 3 days data
# the model shall predict the next day's closing price
# Therefore, the time step to be created is 3

time_steps =3
```

Step 4:

Create Dependent and Independent Features, Remember the split needed shall be based on the Time Step of 3 Days, Reiterating the below image, Where X is the Independent Feature and Y is the Dependent Feature.



You can do the splitting of Dependent and Independent Features in the next step, however, in the current step you will only be preparing the Independent and dependent features by simply defining a function as below.

```
# preparing dependent and independent features
def prepare_data(univariate_data, n_features):
  x, y =[],[]
  for i in range(len(univariate_data)):
    # find the end of this pattern
    end_ix = i + n_features
    # check if we are beyond the sequence
    if end_ix > len(univariate_data)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = univariate_data[i:end_ix], univariate_data[end_ix]
    x.append(seq_x)
    y.append(seq_y)
  return np.array(x), np.array(y)
```

Step 5:

Split the data into Dependent and Independent Features.

```
# split the data into samples
x, y = prepare_data(univariate_data, time_steps)
```

Step 6:

Check if the data is Split as per the need or not

```
print(x),print(y)
```

This Line of Code will return the below output,

```
[[ 44.858  44.144  53.48 ]
 [ 44.144  53.48   62.122]
 [ 53.48   62.122  64.159]
 [ 62.122  64.159  72.801]
 [ 64.159  72.801  72.845]
 [ 72.801  72.845  78.587]
 [ 72.845  78.587  73.83 ]
 [ 78.587  73.83   72.639]
 [ 73.83   72.639  75.749]
 [ 72.639  75.749  70.626]
 [ 75.749  70.626  71.904]
 [ 70.626  71.904  74.88 ]
 [ 71.904  74.88   87.99 ]
 [ 74.88   87.99   96.51 ]
 [ 87.99   96.51   94.2  ]
 [ 96.51   94.2   110.25 ]
 [ 94.2   110.25  108.38 ]]
[ 62.122  64.159  72.801  72.845  78.587  73.83   72.639  75.749  70.626
  71.904  74.88   87.99   96.51   94.2   110.25  108.38  113.47 ]
(None, None)
```

You have successfully split the data as needed.

Step 7:

Check the shape of the data

```
x.shape
```

```
(17, 3)
```

Step 8:

To proceed further, we need a 3 Dimensional Data input, let us reshape it from [samples, time steps] into [samples, time steps, features] using the below line of code.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
x = x.reshape((x.shape[0], x.shape[1], n_features))
```

PART 3: BUILDING THE MODEL

Step 9:

Define and build the LSTM model.

```
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', return_sequences=True, input_shape=(time_steps, n_features)))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(x, y, epochs=100, verbose=1)
```

Step 10:

Visualize the loss based on the above fit model,

```
# For Vizualising the loss, assigning the model a variable
final_model = model.fit(x, y, epochs=100, verbose=0)
```

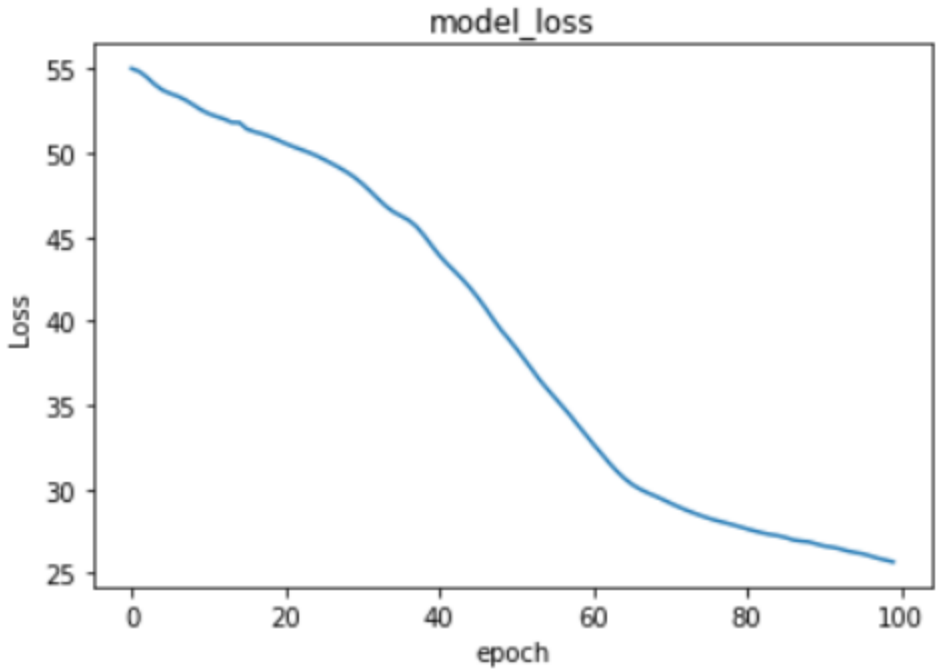
Check the keys present in final_model,

```
final_model.history.keys()
```

```
dict_keys(['loss'])
```

```
import matplotlib.pyplot as plt

plt.plot(final_model.history['loss'])
plt.title('model_loss')
plt.ylabel('Loss')
plt.xlabel('epoch')
```

Output from Step 10:



PART 4: FORECASTING THE VALUES FOR THE NEXT 10 DAYS.

Step 11:

Demonstrate prediction for the next 10 days.

```
# demonstrate prediction for next 10 days
x_input = np.array([110.25, 108.38, 113.47])
temp_input=list(x_input)
lst_output=[]
i=0
while(i<10):
```

```
  if(len(temp_input)>3):
      x_input=np.array(temp_input[1:])
      print("{} day input {}".format(i,x_input))
      #print(x_input)
      x_input = x_input.reshape((1, time_steps, n_features))
      #print(x_input)
      yhat = model.predict(x_input, verbose=0)
      print("{} day output {}".format(i,yhat))
      temp_input.append(yhat[0][0])
      temp_input=temp_input[1:]
      #print(temp_input)
      lst_output.append(yhat[0][0])
      i=i+1
```

```
else:
    x_input = x_input.reshape((1, time_steps, n_features))
    yhat = model.predict(x_input, verbose=0)
    print(yhat[0])
    temp_input.append(yhat[0][0])
    lst_output.append(yhat[0][0])
    i=i+1
```

Step 12:

Print the result

```
print(lst_output)
```

Output from Step 12:

```
[124.36959]
1 day input [108.38       113.47       124.36959076]
1 day output [[128.78783]]
2 day input [113.47       124.36959076 128.78782654]
2 day output [[136.26967]]
3 day input [124.36959 128.78783 136.26967]
3 day output [[145.82053]]
4 day input [128.78783 136.26967 145.82053]
4 day output [[153.21259]]
5 day input [136.26967 145.82053 153.21259]
5 day output [[162.68582]]
6 day input [145.82053 153.21259 162.68582]
6 day output [[173.28693]]
7 day input [153.21259 162.68582 173.28693]
7 day output [[183.49394]]
8 day input [162.68582 173.28693 183.49394]
8 day output [[195.42117]]
9 day input [173.28693 183.49394 195.42117]
9 day output [[208.36455]]
[124.36959, 128.78783, 136.26967, 145.82053, 153.21259, 162.68582, 173.28693, 183.49394, 195.42117, 208.36455]
```

Step 13:

Print the forecasted values,

```
lst_output
```

```
[124.36959,
 128.78783,
 136.26967,
 145.82053,
 153.21259,
 162.68582,
 173.28693,
 183.49394,
 195.42117,
 208.36455]
```

The values in the above list are the forecasted values of close price for the next 10 days.

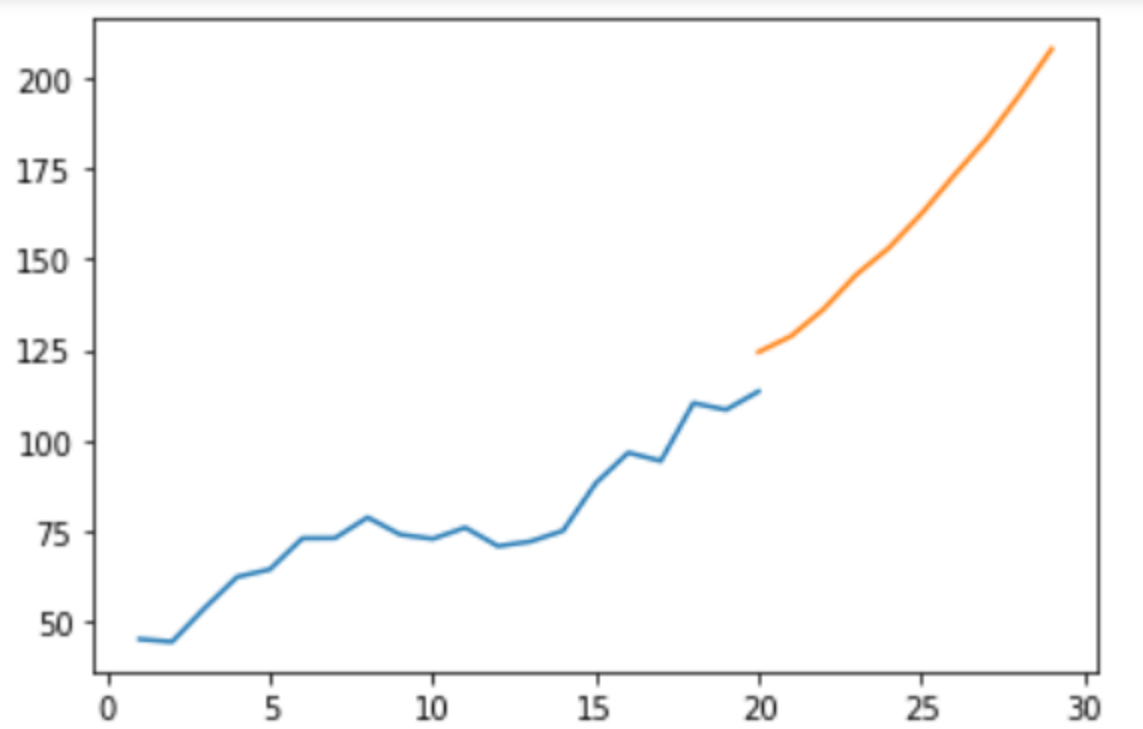
PART 5: VISUALIZING THE OUTPUT

Step 14:

```
day_new=np.arange(1,21)
day_pred=np.arange(20,30)
```

```
plt.plot(day_new,univariate_data)
plt.plot(day_pred,lst_output)
```

Final Output:



You have completed the implementation of Univariate Forecasting of Cryptocurrency prices, you can select any other Crypto-currency of your choice and replicate all of these steps to forecast the prices. Similarly, instead of forecasting the Closing price, you can do it for the open price, high, Low as well.

IMPLEMENTING MULTIVARIATE FORECASTING OF CRYPTO-CURRENCY PRICES USING PYTHON:

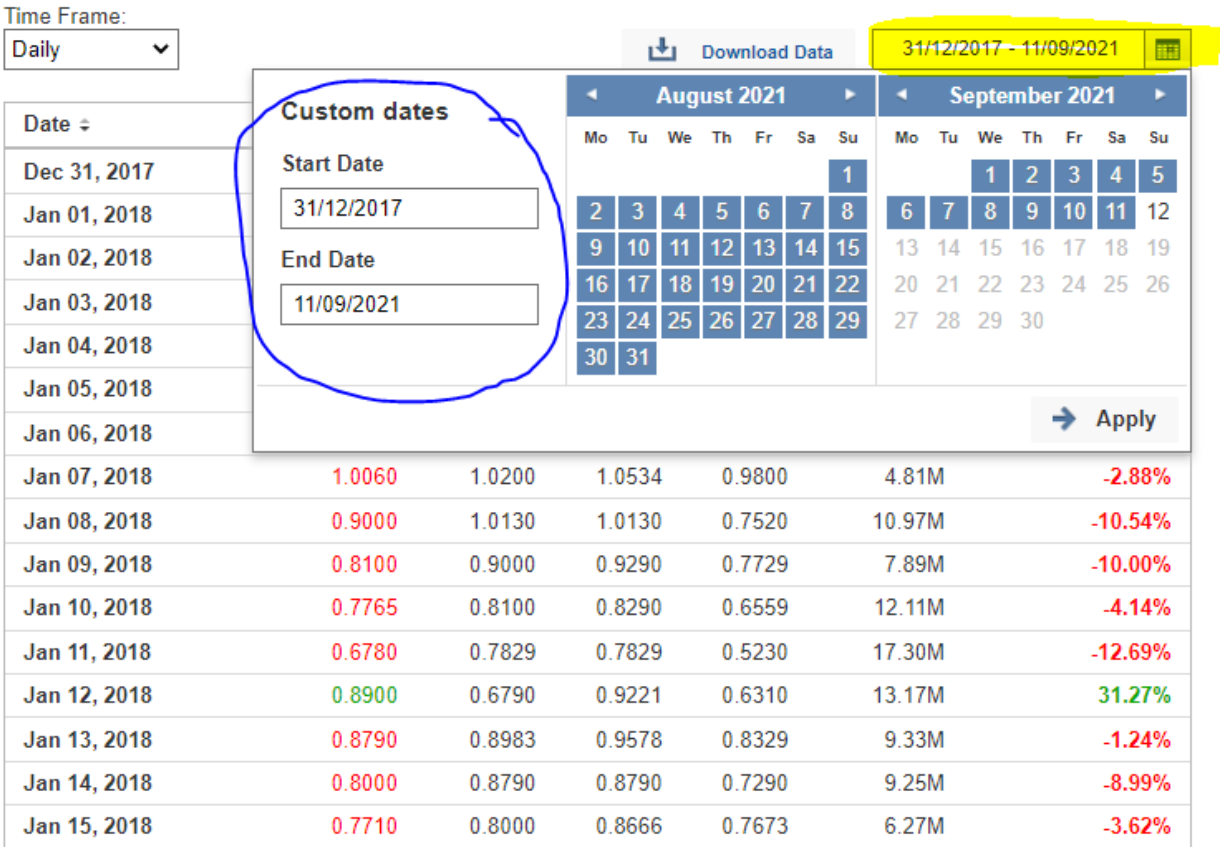
You have seen the practical implementation of Univariate Cryptocurrency price forecasting on a beginner level, now you will study Multivariate cryptocurrency price forecasting at a more advanced level, let's take it forward step by step from the cryptocurrency prices to be used as input, building the LSTM model and deriving the forecast.

Problem statement: Pick any crypto currency's historical data of your choice (choosing **CARDANO** for this use-case implementation), based on the data between Dec.31.2017 to Sep.11.2021, Forecast the next 30 Days price, using a time step of 150 days.

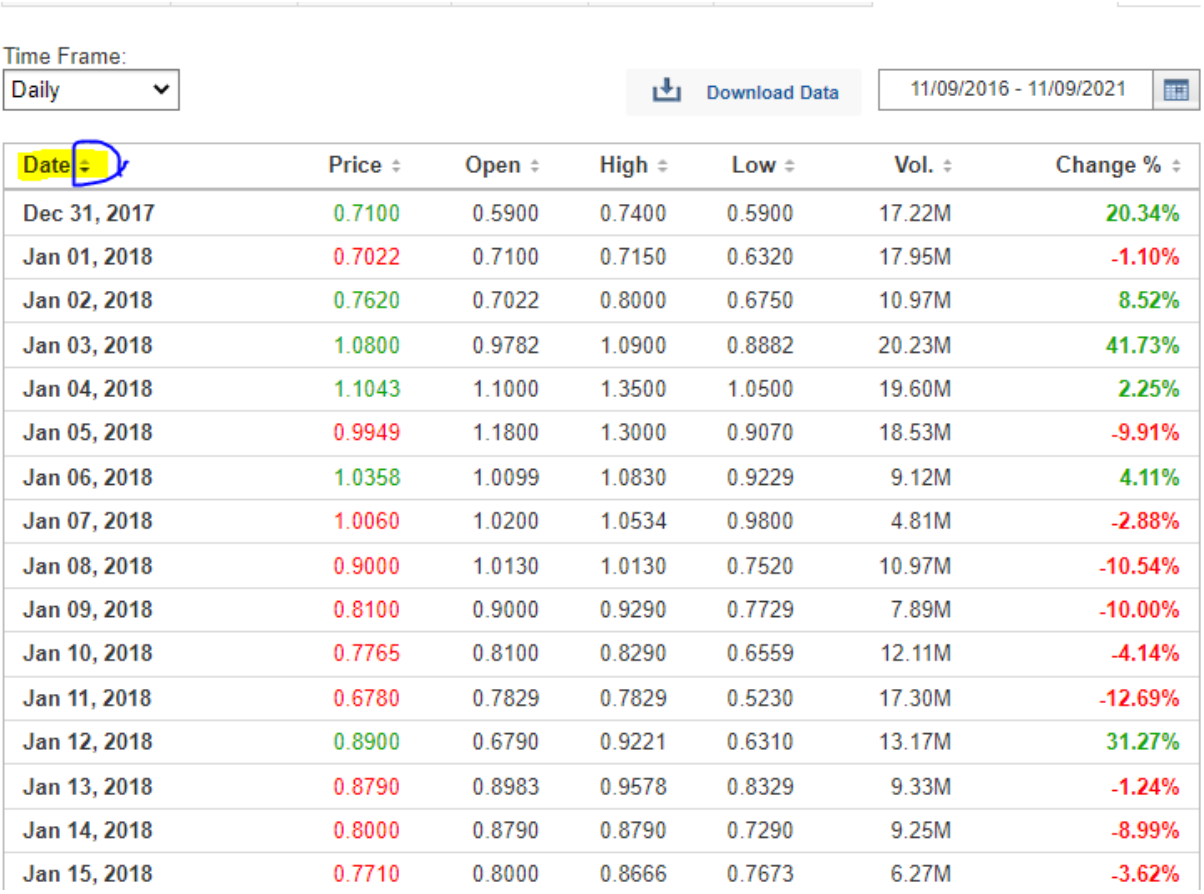
You can download the data from the below URL:

https://in.investing.com/crypto/cardano/historical-data

Once you open the URL, Select the custom dates with Daily Time-Frame, as shown in the below image,



After selecting the custom date range, sort the data from oldest to newest by clicking on the below-highlighted area



Date	Price	Open	High	Low	Vol.	Change %
Dec 31, 2017	0.7100	0.5900	0.7400	0.5900	17.22M	20.34%
Jan 01, 2018	0.7022	0.7100	0.7150	0.6320	17.95M	-1.10%
Jan 02, 2018	0.7620	0.7022	0.8000	0.6750	10.97M	8.52%
Jan 03, 2018	1.0800	0.9782	1.0900	0.8882	20.23M	41.73%
Jan 04, 2018	1.1043	1.1000	1.3500	1.0500	19.60M	2.25%
Jan 05, 2018	0.9949	1.1800	1.3000	0.9070	18.53M	-9.91%
Jan 06, 2018	1.0358	1.0099	1.0830	0.9229	9.12M	4.11%
Jan 07, 2018	1.0060	1.0200	1.0534	0.9800	4.81M	-2.88%
Jan 08, 2018	0.9000	1.0130	1.0130	0.7520	10.97M	-10.54%
Jan 09, 2018	0.8100	0.9000	0.9290	0.7729	7.89M	-10.00%
Jan 10, 2018	0.7765	0.8100	0.8290	0.6559	12.11M	-4.14%
Jan 11, 2018	0.6780	0.7829	0.7829	0.5230	17.30M	-12.69%
Jan 12, 2018	0.8900	0.6790	0.9221	0.6310	13.17M	31.27%
Jan 13, 2018	0.8790	0.8983	0.9578	0.8329	9.33M	-1.24%
Jan 14, 2018	0.8000	0.8790	0.8790	0.7290	9.25M	-8.99%
Jan 15, 2018	0.7710	0.8000	0.8666	0.7673	6.27M	-3.62%

When you forecast the data you will need the oldest data in the beginning and the latest data in the end, if you miss sorting the data the entire forecast will become void as it will give a wrong interpretation. In simple words, upon sorting, the oldest records shall be at the top, and the latest records shall be at the bottom.

Now Click on Download Data and save a **.CSV** copy of the selected data into your local machine so that you can use it for further analysis and forecasting.

Time Frame: Daily

Download Data

11/09/2016 - 11/09/2021

Date	Price	Open	High	Low	Vol.	Change %
Dec 31, 2017	0.7100	0.5900	0.7400	0.5900	17.22M	20.34%
Jan 01, 2018	0.7022	0.7100	0.7150	0.6320	17.95M	-1.10%
Jan 02, 2018	0.7620	0.7022	0.8000	0.6750	10.97M	8.52%
Jan 03, 2018	1.0800	0.9782	1.0900	0.8882	20.23M	41.73%
Jan 04, 2018	1.1043	1.1000	1.3500	1.0500	19.60M	2.25%
Jan 05, 2018	0.9949	1.1800	1.3000	0.9070	18.53M	-9.91%
Jan 06, 2018	1.0358	1.0099	1.0830	0.9229	9.12M	4.11%
Jan 07, 2018	1.0060	1.0200	1.0534	0.9800	4.81M	-2.88%
Jan 08, 2018	0.9000	1.0130	1.0130	0.7520	10.97M	-10.54%
Jan 09, 2018	0.8100	0.9000	0.9290	0.7729	7.89M	-10.00%
Jan 10, 2018	0.7765	0.8100	0.8290	0.6559	12.11M	-4.14%
Jan 11, 2018	0.6780	0.7829	0.7829	0.5230	17.30M	-12.69%
Jan 12, 2018	0.8900	0.6790	0.9221	0.6310	13.17M	31.27%
Jan 13, 2018	0.8790	0.8983	0.9578	0.8329	9.33M	-1.24%
Jan 14, 2018	0.8000	0.8790	0.8790	0.7290	9.25M	-8.99%
Jan 15, 2018	0.7710	0.8000	0.8666	0.7673	6.27M	-3.62%

As the data is ready now, let's start implementing the model with python,

PART 1: LOAD THE REQUIRED PACKAGES AND DATASET

Step 1:

Import the required packages,

```
# Importing the required packages
import numpy as np
import pandas as pd
import tensorflow as tf
import keras
```

Step 2:

Load the downloaded Cardano crypto prices dataset into python, and print the top rows to have a glance at the records,

```
df=pd.read_csv('/content/drive/MyDrive/Cardano Historical Data - Investing.com India.csv')
df.head() # View the first 5 records of the dataset
```

	Date	Price	Open	High	Low	Vol.	Change %
0	Dec 31, 2017	0.7100	0.5900	0.740	0.5900	17.22M	20.34%
1	Jan 01, 2018	0.7022	0.7100	0.715	0.6320	17.95M	-1.10%
2	Jan 02, 2018	0.7620	0.7022	0.800	0.6750	10.97M	8.52%
3	Jan 03, 2018	1.0800	0.9782	1.090	0.8882	20.23M	41.73%
4	Jan 04, 2018	1.1043	1.1000	1.350	1.0500	19.60M	2.25%

Note: The Price Feature/Column resembles the close price of the cryptocurrency.

Step 3:

Check the summary of the data, ensure there are no null values,

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1351 entries, 0 to 1350
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Date      1351 non-null   object
 1   Price     1351 non-null   float64
 2   Open      1351 non-null   float64
 3   High      1351 non-null   float64
 4   Low       1351 non-null   float64
 5   Vol.      1351 non-null   object
 6   Change %  1351 non-null   object
dtypes: float64(4), object(3)
memory usage: 74.0+ KB
```

As per the summary in the above image, there are 1351 records in the dataset and there are no null values, thus we can proceed further.

Step 4:

Reset the index for 'Price feature, so that it will handle the case of any multi-index (in the current data which you are using does not have any multi-index), including this step here only for your future references. Just in case, you find any input data with multi-index this step will help, for more information on this you can refer to the below URL,

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.reset_index.html

```
df1=df.reset_index()['Price']
df1
```

```
0       0.7100
1       0.7022
2       0.7620
3       1.0800
4       1.1043
         ...
1346    2.5047
1347    2.4699
1348    2.5161
1349    2.3825
1350    2.6902
Name: Price, Length: 1351, dtype: float64
```

PART 2: DATA PRE-PROCESSING

Step 5:

Scale the values, as the LSTM model will work best when the input values are scaled, you will use MinMaxScaler and transform the values,

```
from sklearn.preprocessing import MinMaxScaler
scaler=MinMaxScaler(feature_range=(0,1))
df1=scaler.fit_transform(np.array(df1).reshape(-1,1))
```

Validate if the values are scaled or not,

```
print(df1)
```

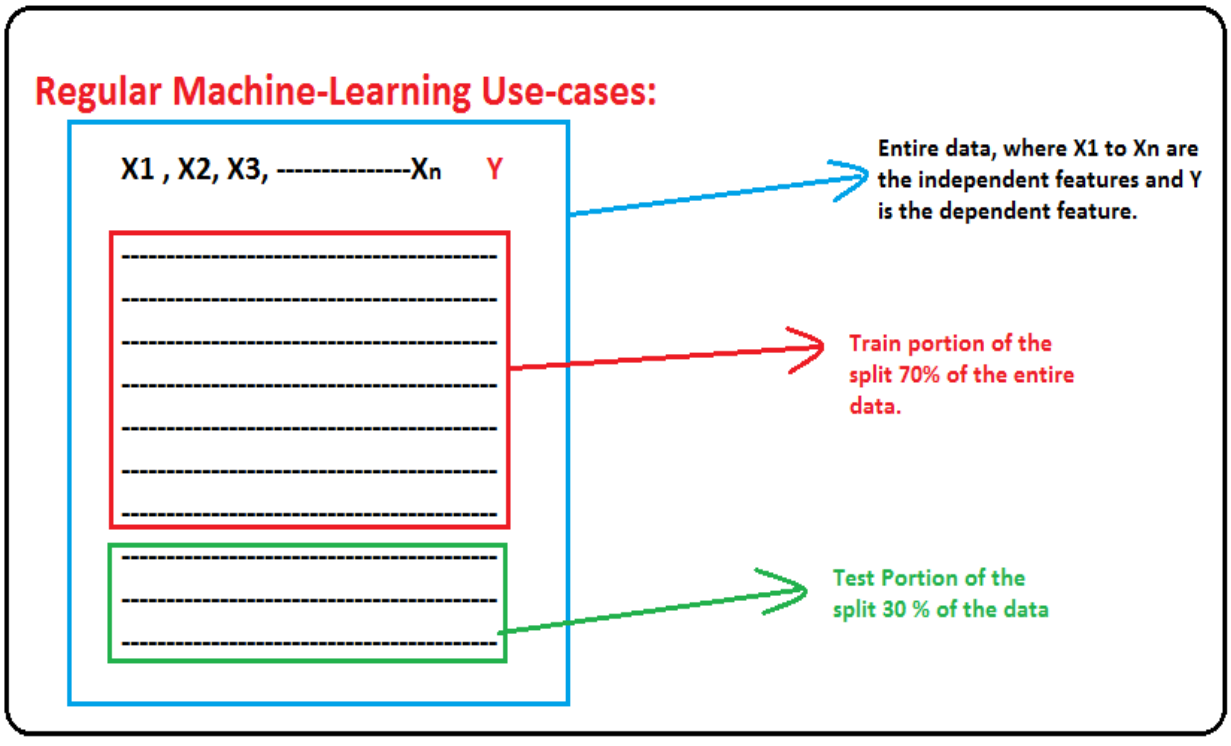
```
[[0.23344663]
 [0.23079538]
 [0.25112169]
 ...
 [0.84734874]
 [0.80193746]
 [0.90652617]]
```

Step 6:

Split the data into train and test.

Before keying in the piece of code to split the data, you need to understand the difference between how the train-test data is split in the regular machine learning use-cases and Time-series forecasting use-case.

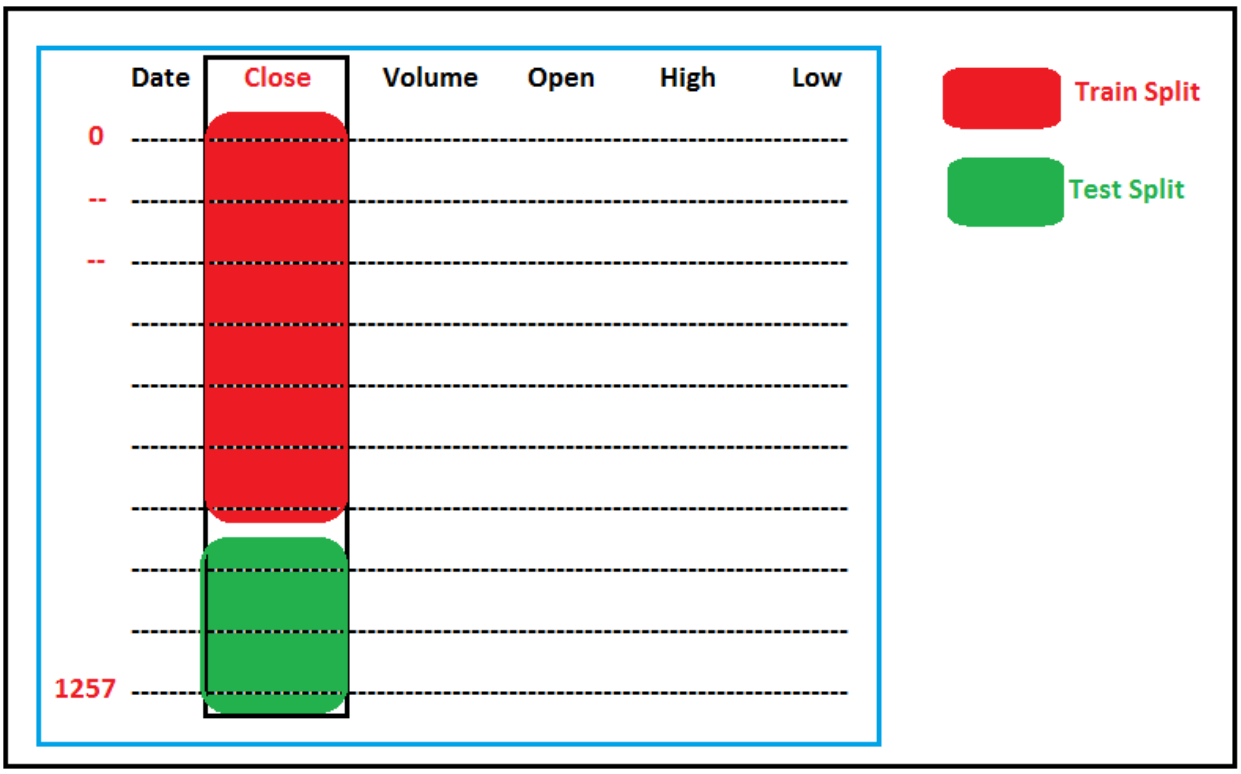
In regular machine-learning use-cases, assume a **Train: Test** split of **70: 30**, upon splitting the original dataset will be divided into 2 chunks as shown below,



However, note that the split, in reality, happens randomly, not necessarily the first 70 % of the records are Training data and the last 30 % records are the test data, the above image is only an illustration to make you understand that the train-test split will divide the entire data into 2 chunks.

How is the train test splitting done in time-series forecasting use-case?

In the use-cases of time-series forecasting or cases where the input data is sequential information, the entire data is not split, rather you would pick the feature which you are intending to forecast and then split that particular feature into train and test, in this particular use case you will be picking and splitting the 'Price' feature from the original dataset as shown below,



Use the below code to split the data into train and test,

```
# splitting dataset into train and test split
training_size=int(len(df1)*0.65) # Defining the training data size
test_size=len(df1)-training_size # Test data =  Total length of the data - length of training data
train_data,test_data=df1[0:training_size,:],df1[training_size:len(df1),:1]
```

Validate the split by having a glance at the train and test data sizes created in the above code,

```
training_size,test_size
```

```
(878, 473)
```

You can interpret it as the 0 to 878 indexed data is the Training portion and the remaining values are the Test portion of the data.

Step 7:

See if the data is in the form of a matrix or an array, you need the data in the form of a matrix to proceed further.

```
train_data
```

```
array([[2.33446635e-01],
       [2.30795377e-01],
       [2.51121686e-01],
       [3.59211421e-01],
       [3.67471108e-01],
       [3.30285520e-01],
       [3.44187627e-01],
       [3.34058464e-01],
       [2.98028552e-01],
       [2.67437118e-01],
       [2.56050306e-01],
       [2.22569680e-01],
       [2.94629504e-01],
       [2.90890551e-01],
       [2.64038069e-01],
       [2.54180829e-01],
       [1.96464990e-01],
```

Step 8:

As the training data is in the form of an array you need to convert it in the form of a matrix,

```
import numpy
# convert an array of values into a dataset matrix
def create_dataset(dataset, time_step=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-time_step-1):
    a = dataset[i:(i+time_step), 0]   ###i=0, 0,1,2--- 150 as the time step is 150
    dataX.append(a)
    dataY.append(dataset[i + time_step, 0])
  return numpy.array(dataX), numpy.array(dataY)
```

Step 9:

Reshape the data into the form of x=t, t+1, t+2, t+3, and y=t+4 based on time steps of 150 using the below code,

```
# reshape the data
time_step = 150
X_train, y_train = create_dataset(train_data, time_step)
X_test, ytest = create_dataset(test_data, time_step)
```

Step 10:

Check if the data is reshaped based on time-steps of 150 or not,

```
print(X_train)
```

```
[[0.23344663 0.23079538 0.25112169 ... 0.05774983 0.05098572 0.06074099]
 [0.23079538 0.25112169 0.35921142 ... 0.05098572 0.06074099 0.06308634]
 [0.25112169 0.35921142 0.36747111 ... 0.06074099 0.06308634 0.06825289]
 ...
 [0.00346703 0.00373895 0.00329708 ... 0.01145479 0.00982325 0.01104691]
 [0.00373895 0.00329708 0.003569   ... 0.00982325 0.01104691 0.01077498]
 [0.00329708 0.003569   0.00370496 ... 0.01104691 0.01077498 0.00958532]]
```

```
print(X_train.shape), print(y_train.shape)
```

```
(727, 150)
(727,)
(None, None)
```

You can notice that X_train has got 150 features and y_train has got only 1 feature and the total number of records is 727.

Similarly, check for the test data,

```
print(X_test.shape), print(ytest.shape)
```

```
(322, 150)
(322,)
(None, None)
```

You can notice that X_test has got 150 features and y_test has got only 1 feature and the total number of records is 322.

PART 3: MODEL CREATION AND MODEL EVALUATION

Step 11:

Convert the data into 3 dimensional as LSTM will accept a 3 Dimensional data as input, so let's reshape the data into 3 dimensions by adding a value of 1 in the third dimension as shown below,

```
# reshape input to be [samples, time steps, features] which is required for LSTM
X_train =X_train.reshape(X_train.shape[0],X_train.shape[1] , 1)
X_test = X_test.reshape(X_test.shape[0],X_test.shape[1] , 1)
```

Step 12:

Load the required packages to build the LSTM model,

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
```

Step 13:

Create the LSTM network by stacking multiple layers of LSTM, use 'return_sequences = True' to stack the LSTM layers.

```
model=Sequential() # Initializing the sequential model
model.add(LSTM(200,return_sequences=True,input_shape=(150,1))) # 150 = time_step and 1 = 3rd dimension
model.add(LSTM(200,return_sequences=True)) # adding the second LSTM layer
model.add(LSTM(100)) # Adding the third LSTM Layer
model.add(Dense(1)) # adding the final/Output layer
model.compile(loss='mean_squared_error',optimizer='adam')
```

Step 14:

Check the model summary, i.e. the overview of layers created in the above step,

```
model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_18 (LSTM)               (None, 150, 200)          161600
_________________________________________________________________
lstm_19 (LSTM)               (None, 150, 200)          320800
_________________________________________________________________
lstm_20 (LSTM)               (None, 100)               120400
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 101
=================================================================
Total params: 602,901
Trainable params: 602,901
Non-trainable params: 0
_________________________________________________________________
```

Step 15:

Fit the model,

```
model.fit(X_train,y_train,validation_data=(X_test,ytest),epochs=150,batch_size=64,verbose=1)
```

Step 16:

Predict the model,

```
# predict the model
train_predict=model.predict(X_train)
test_predict=model.predict(X_test)
```

Step 17:

Transform the values to their original form,

As you have scaled the values before building and fitting the model using MinMaxScaler, you now need to transform the values back into their original form as it is not feasible to measure the performance metrics i.e. RMSE with the scaled-down values.

You can do this by applying the inverse transformation

```
#Transformback to original form
train_predict=scaler.inverse_transform(train_predict)
test_predict=scaler.inverse_transform(test_predict)
```

Step 18:

Calculate RMSE metrics for train data,

```
# RMSE performance metrics for training data
import math
from sklearn.metrics import mean_squared_error
math.sqrt(mean_squared_error(y_train,train_predict))
```

```
0.05622934367869417
```

Step 19:

Calculate RMSE metrics for test data,

```
# Test Data RMSE
math.sqrt(mean_squared_error(ytest,test_predict))
```

```
0.8368174399943379
```

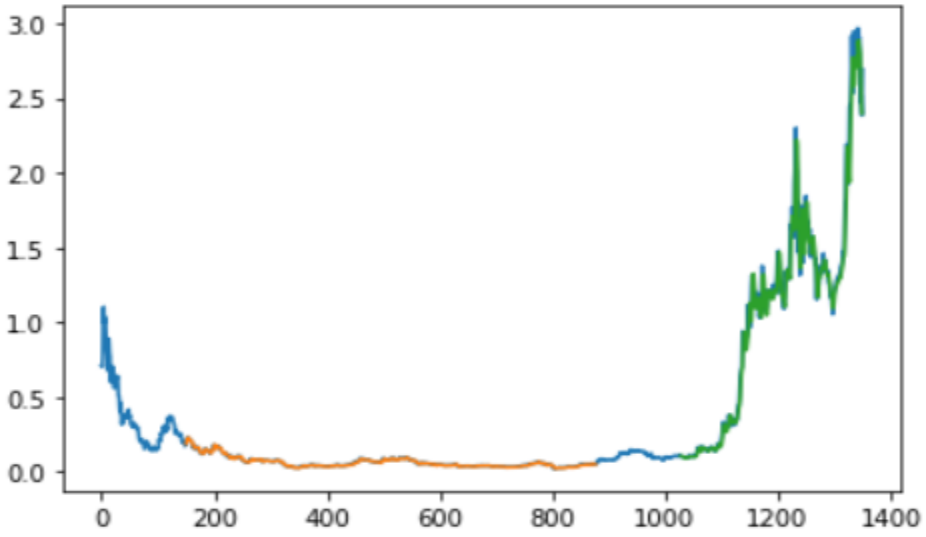
Note that the RMSE values under the price forecasting use-cases will differ based on the value of the price,

Step 20:

Plot the graph which shows the line chart of original data, train predict and test predict

```
# Plot the original, Train and Test data
import matplotlib.pyplot as plt

# shift train predictions for plotting
look_back=150 # resembles the time-step of 150
trainPredictPlot = numpy.empty_like(df1)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(train_predict)+look_back, :] = train_predict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(df1)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(train_predict)+(look_back*2)+1:len(df1)-1, :] = test_predict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(df1))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Output:



You can interpret the above chart as, the Blue Line as the original data points, i.e., the original values from the 'Price' feature, Orange line as the Training-predicted data, and the green line as the Test-predicted data.

PART 4: FORECASTING THE NEXT 30 DAYS PRICE, BASED ON THE DOWNLOADED HISTORICAL DATA AND TIME STEP OF 150 DAYS.

Step 21:

From the test dataset, the last day is Sep-11-2021, thus the forecasting needs to be done for the next 30 days starting Sep-12-2021 based on the time step of 150 days you need to predict the value of Close price of Sep-12-2021 and so on for the next 30 days' time frame.

From Step 6, you already know that the Test data length is 473, Thus to predict the data for Sep-12-2021, using the time step of 150 you need to use 323 as shown below and label it as x_input.

```
# Creating x_input variable for the test data
x_input=test_data[323:].reshape(1,-1) # 473 -150 = 323
x_input.shape
```

```
(1, 150)
```

Step 22:

Convert all the values of above created 'x_input' into a list before proceeding further,

```
# Convert the values of X_input into a list and label it as temp_input
temp_input=list(x_input)
temp_input=temp_input[0].tolist()
```

Step 23:

Validate if the values are converted to list or not,

```
temp_input
```

```
[0.49473147518694766,
 0.47168592794017683,
 0.45808973487423527,
 0.4261386811692726,
 0.398606390210741,
 0.42290958531611156,
 0.40152957171991843,
 0.38174711080897356,
 0.3853161114887832,
 0.36672331747110815,
 0.362576478585996,
 0.4117267165193746,
 0.43667573079537736,
 0.44653297076818493,
 0.43636981645139367,
 ...
 0.9180829367777024,
 0.9235214140040789,
 0.8536709721278042,
 0.9934398368456833,
 0.9597212780421481,
 0.961760707002039 5,
 0.9218558803535011,
 0.9334126444595513,
 0.9676410605030592,
 0.9985723997280761,
 0.9999999999999999,
 0.9541128484024474,
 0.981645139360979,
 0.9548266485384093,
 0.8434738273283482,
 0.831645139360979,
 0.8473487423521414,
 0.8019374575118966,
 0.9065261726716519]
```

You can notice that the values are enclosed within [ ], resembling a list of values.

Step 24:

Demonstrate the prediction for the next 30 days,

```
# demonstrate prediction for next 30 days
from numpy import array

lst_output=[]
n_steps=150
i=0
while(i<30):

    if(len(temp_input)>150):
        #print(temp_input)
        x_input=np.array(temp_input[1:])
        print("{} day input {}".format(i,x_input))
        x_input=x_input.reshape(1,-1)
        x_input = x_input.reshape((1, n_steps, 1))
        #print(x_input)
        yhat = model.predict(x_input, verbose=0)
        print("{} day output {}".format(i,yhat))
        temp_input.extend(yhat[0].tolist())
        temp_input=temp_input[1:]
        #print(temp_input)
        lst_output.extend(yhat.tolist())
        i=i+1
```

```
    else:
        x_input = x_input.reshape((1, n_steps,1))
        yhat = model.predict(x_input, verbose=0)
        print(yhat[0])
        temp_input.extend(yhat[0].tolist())
        print(len(temp_input))
        lst_output.extend(yhat.tolist())
        i=i+1


print(lst_output)
```

In the output, you can see that the values of each day are predicted by using the previous 150 days Price data as input.

Step 25:

Check the forecasted Price for the next 30 days,

```
# Lets check the predicted Close prices for next 30 days
scaler.inverse_transform(lst_output)
```

Output:

```
array([[2.47172088],
       [2.5172081 ],
       [2.52258524],
       [2.51366431],
       [2.4973698 ],
       [2.47518313],
       [2.44831775],
       [2.41840187],
       [2.38717871],
       [2.35620472],
       [2.32668024],
       [2.29937507],
       [2.27463414],
       [2.25243502],
       [2.23248011],
       [2.21430226],
       [2.197367  ],
       [2.18115701],
       [2.16523689],
       [2.14928537],
       [2.13311063],
       [2.11664094],
       [2.09990173],
       [2.08298733],
       [2.06603085],
       [2.04917361],
       [2.03254487],
       [2.0162451 ],
       [2.00033866],
       [1.98485429]])
```

Step 26:

Create variables to plot a graph of the forecasted values,

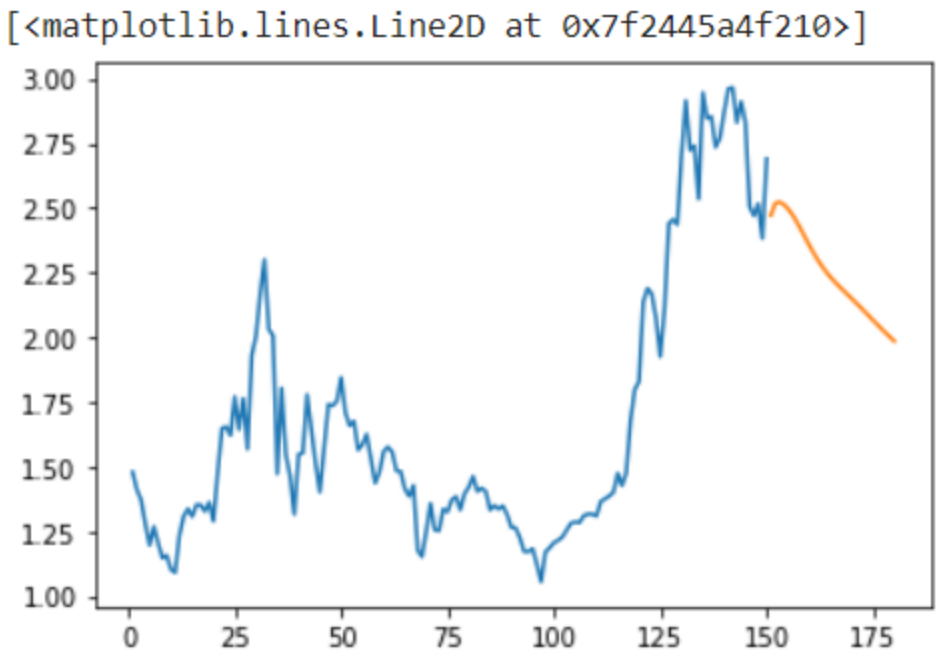
```
day_new=np.arange(1,151) # 150 Days time step
day_pred=np.arange(151,181) # to make a forecast of 30Days i.e 151 to 180 days
```

Step 27:

Plot the forecast,

```
plt.plot(day_new,scaler.inverse_transform(df1[1201:])) # 1201 = 1351 - 150 (time-step)
plt.plot(day_pred,scaler.inverse_transform(lst_output))
```

Output:



Interpretation: The Orange portion of the line is the forecast made using the LSTM model for the next 30 days.

SUMMARY:

You have seen the practical implementation of how to forecast the Cryptocurrency prices with univariate data and Multi-variate data, you can follow the same sequence of steps to forecast the prices of any other Crypto-Currencies of your choice, in this chapter you have seen the forecasting for the Price feature of the Cryptocurrency (Cardano), however, you can implement the model on the other features as well, such as High, Low, Open.

Before concluding, it is important to understand that the forecasting shown in this chapter is purely based on the available Historical data and the theory of RNN-LSTM models, however, in reality, the Crypto-currency prices may be influenced by other multiple market and economic factors.

PROGRAMMING ASSIGNMENT:

Using the same Cryptocurrency data i.e. Cardano, do the next 30 days forecasting for the 'Low' feature from the original dataset downloaded at the beginning of the practical implementation.

Hint: You can replicate all the steps shown above, the only change will be instead of using the 'Price' feature you have to use the 'Low' feature from the data.

CHAPTER 6: GRU (Gated Recurrent Units).

INTRODUCTION:

In the previous chapter you have studied RNN –LSTM and how the architecture of LSTM functions to deliver the results by forgetting the irrelevant context and remembering the relevant context, GRU has a similar architecture to that of LSTM. GRU was built after the LSTM, however, it has gained a lot of popularity.

To reiterate, from the previous chapters you might already be aware of how a simple RNN network works, the problem of long-term dependencies, and how the network suffers from a short-term memory problem, in case you are not aware of these listed theories you can refer to chapter 3 of this book.

"GRU"

Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks, introduced in 2014 by Kyunghyun Cho et al. The GRU is like a long short-term memory (LSTM) with a forget gate, but has fewer parameters than LSTM, as it lacks an output gate. GRU's performance on certain tasks of polyphonic music modeling, speech signal modeling and natural language processing was found to be similar to that of LSTM. GRUs have been shown to exhibit better performance on certain smaller and less frequent datasets.

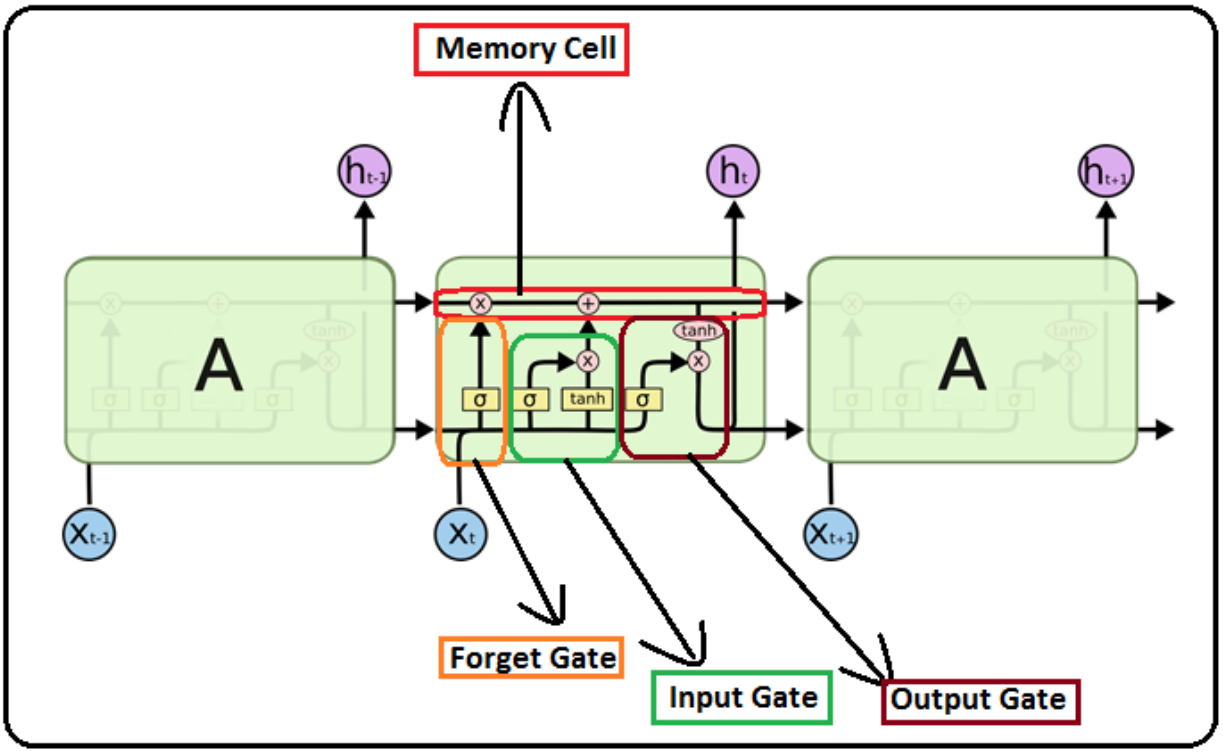
For a better understanding of the concept, let us quote the same example which you have already studied in the previous chapters i.e. auto sentence completion by predicting the final word from the following sentence, "Washington is the capital of the **United States**… the current president is **Joe Biden.**" In such use-cases you need the algorithm to accurately guess the final word (Joe Biden) after '**is'** in the sentence, for which the model needs to remember the word 'United States' from the sentence, the traditional RNN model suffers from a short-term memory problem, suppose say, it only remembers the last three words ("current president is") it will become difficult for the model to predict and auto-complete the final word as it will be confused, the model will be unaware of which country's president to fill in the space.

Thus, to avoid this problem you will need a better model which can remember the words in the sentence which has a relevant context. In the above sentence "United States…" is the word

that the model needs to remember to predict the final word I.e. Joe Biden. To remember the words with relevant context, the model needs to cascade it through the long-term memory, LSTM is an expert in handling such tasks in comparison to the traditional RNN, for you to understand the concept with ease, it is important that you complete and be aware of the concepts studied in the previous chapters of this book, especially the concepts covered in chapter 3.

LSTM will handle long and short-term memory by using the Cell state and hidden state in its network, you can refer to GRU as a modified version of LSTM, i.e. the lighter version of LSTM.

Let's recollect the overview and the architecture of LSTM, it mainly comprises the Forget Gate, Input Gate, and the Output Gate.



Forget Gate - The key functionality of the forget gate in the LSTM model is that it discovers the information which needs to be discarded by using the sigmoid function. The sigmoid function is the one that returns values between 0 & 1 irrespective of the input values.

Input Gate - The functionality of the Input gate is that it supports adding information with regards to the changed context.

Output Gate - The final layer of the LSTM, the input, and the memory of the block is used to determine the output. Sigmoid function decides which values to let through 0, 1 and Tanh function contributes the weightage to the values which are passed to decide their level of importance ranging from-1 to 1 and multiplied with the output returned by the sigmoid function.

LSTM network is nothing but Long-term memory in short-term memory where **Cell state** in the network is referred to the Long-term memory and it carries the information across all the timestamps and **Hidden state** is referred to the short-term memory, **GRU** network combines both Long-term and short-term memory into its hidden-state. As GRU is a lighter version of LSTM it will only have the hidden state. GRU network does not possess the cell state as in LSTM.

Adding to the above LSTM will have three gates, i.e. the Forget Gate, Input Gate, and the Output Gate, GRU network will only have two gates namely, **Update Gate** and **Reset Gate**.

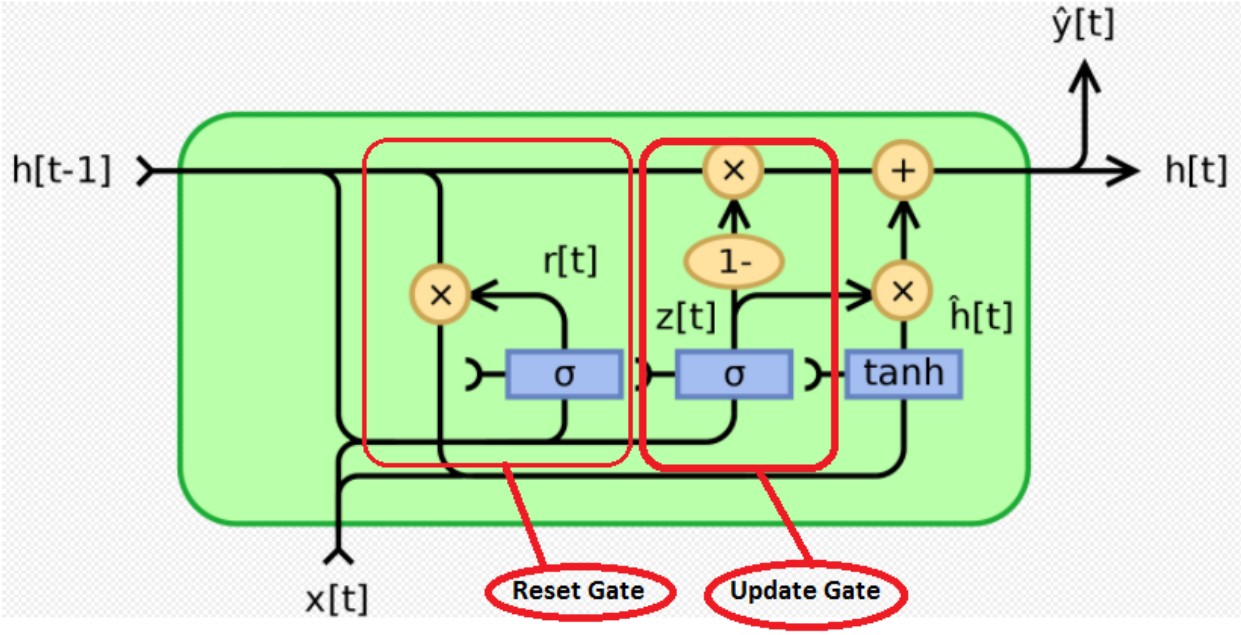


Image Source:
https://en.wikipedia.org/wiki/File:Gated_Recurrent_Unit,_base_type.svg#/media/File:Gated_Recurrent_Unit,_base_type.svg

Update Gate – It is responsible for the Short-term memory of the network, the functionality of the update gate is similar to the Forget Gate and Input Gate of LSTM, and it determines what

information needs to be discarded and what new information needs to be added when the context of information is changed.

Reset Gate – The functionality of the Reset Gate is that it helps the model to decide how much past information to forget.

Though the functionality of these two gates look similar to the forget gate and the input gate of the LSTM network they are not the same, there is a slight difference to understand here.

Let us reiterate another sentence from Chapter 3**, "John and Michael are good at Soccer, Michael is also good at pop-singing, he is a State level performer, and the winner of the last year Pop singing competition held at the university is _____."**

Before auto-filling the blank, You need to understand that the above sentence is mainly divided into 2 statements, The first part of the statement comprises the information about Soccer, and the second part of the statement is mainly confined to Michael and pop-singing, when you implement this sentence into a model to auto-fill the blank, the model shall be able to fill the blank with Michael.

Basically, in the above sentence, we are first talking about John and Michael who are good at soccer, the model shall first remember this context, and later when the context is changed, we are talking only about Michael who is good at pop-singing, the model shall forget the previous context about John and soccer and only remember the context about Michael and pop-singing only then the model will be able to predict the final word accurately, i.e. Michael. This is how GRU is expected to combine Long term and short-term memory to remember and forget the information with relevance to the context of information with the help of the Update gate and Reset gate.

To drill down further, when the context is changed to Pop-singing from Soccer, the model needs to forget about Soccer and this task is accomplished by the Reset gate of the GRU network, Reset gate takes the hidden state $\mathbf{h_{(t-1)}}$ and the current word/context i.e. **Xt** and calculates the value of the reset gate i.e. $\mathbf{r_{(t)}}$

$$\mathbf{r}_{(t)} = \delta \left(\mathbf{W}_r \mathbf{x}_{(t)} + \mathbf{U}_r \mathbf{h}_{(t-1)} \right)$$

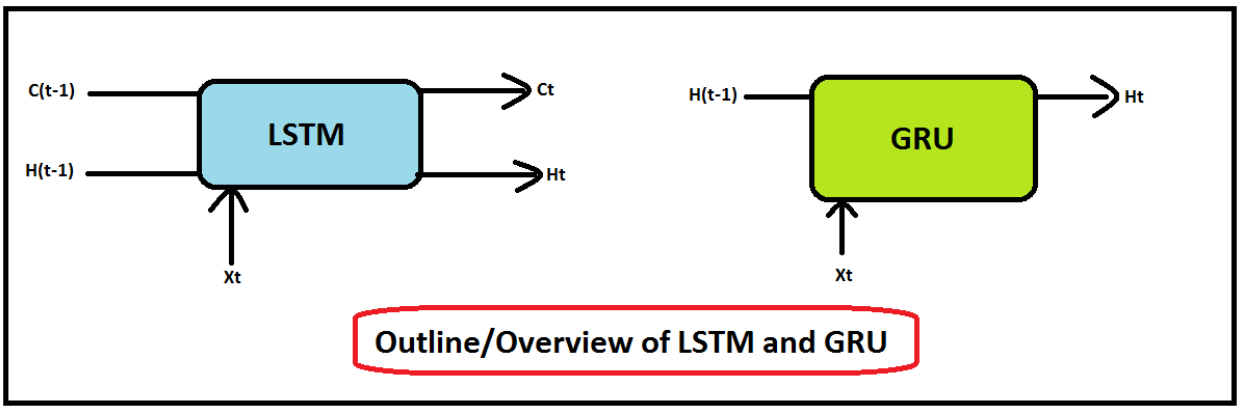
Now, when it comes to pop-singing the model needs to remember only about pop-singing, while the reset gate has already helped the model to forget about Soccer, the update gate helps

the model to remember the memory of pop-singing, the value of the update gate is denoted by $\mathbf{Z_{(t)}}$

$$z_{(t)} = \delta \left(W_z x_{(t)} + U_z h_{(t-1)} \right)$$

In the above two equations, **W = Weight matrix of input associated with hidden state** and **U = weight matrix of input** and the values of $r_{(t)}$ and $z_{(t)}$ ranges between 0 & 1 because of the sigmoid function.

OUTLINE/OVERVIEW OF GRU



In the above image of outline about LSTM and GRU networks, you can notice that there is no cell state in the GRU network, whereas, in the LSTM network, the network had a cell state where it functions as long-term memory and carries information throughout all the timestamps and the hidden state of the LSTM network functions as the Short-term memory.

To simplify, in GRU the reset gate functions as the short-term memory and the update gate functions as the Long term memory. Now that you have studied about Reset gate and the update gate, let us study further how GRU works and how the hidden state (Ht) is calculated.

To calculate the Hidden state in GRU, the network follows two steps:

- Computation of Candidate Hidden state.
- Computation of Current Hidden state.

Candidate hidden state

The input and the hidden state from the previous timestamp i.e. t-1 are multiplied by the reset gate output i.e. **r(t)** and then this output is passed through the Tanh function.

$$\hat{H}_t = \tanh(x_t * u_g + (r_t \,.\, h_{t-1}) * w_g)$$

The value of the reset gate plays a vital role in this calculation, if the value of r(t) is 1, then the entire information from the previous hidden state will be considered and Vice –versa if the value of r(t) is 0.

Current hidden state

After calculating the candidate's hidden state, its value is used to calculate the current hidden state using the Update gate, The update gate of the GRU controls both the past information i.e. H(t-1), and the new information which comes from the candidate hidden state.

$$H_t = U_t \,.\, H_{t-1} + (1-U_t) \,.\, \hat{H}_t$$

The value of **Ut** controls the value of Ht in the above equation. It ranges between 0 and 1.

Differences between LSTM and GRU Networks

- LSTM has 3 Gates – Forget Gate, Input Gate, and the Output Gate, GRU comprises of only 2 Gates i.e. Reset Gate and the Update Gate.
- LSTM network possesses 2 States namely, the Cell State - Long Term Memory and the Hidden State – Short Term Memory. Whereas, GRU network only deals with Hidden State and there is no Cell State.
- LSTM uses its internal memory, whereas GRU does not possess any internal memory.

Advantage of GRU over LSTM NETWORK

Since GRU has got fewer gates and there is no separate Cell state and hidden state it makes the computation lighter and thus contributes to delivering faster results when compared to the LSTM network.

PRACTICAL IMPLEMENTATION OF GRU USING PYTHON-KERAS:

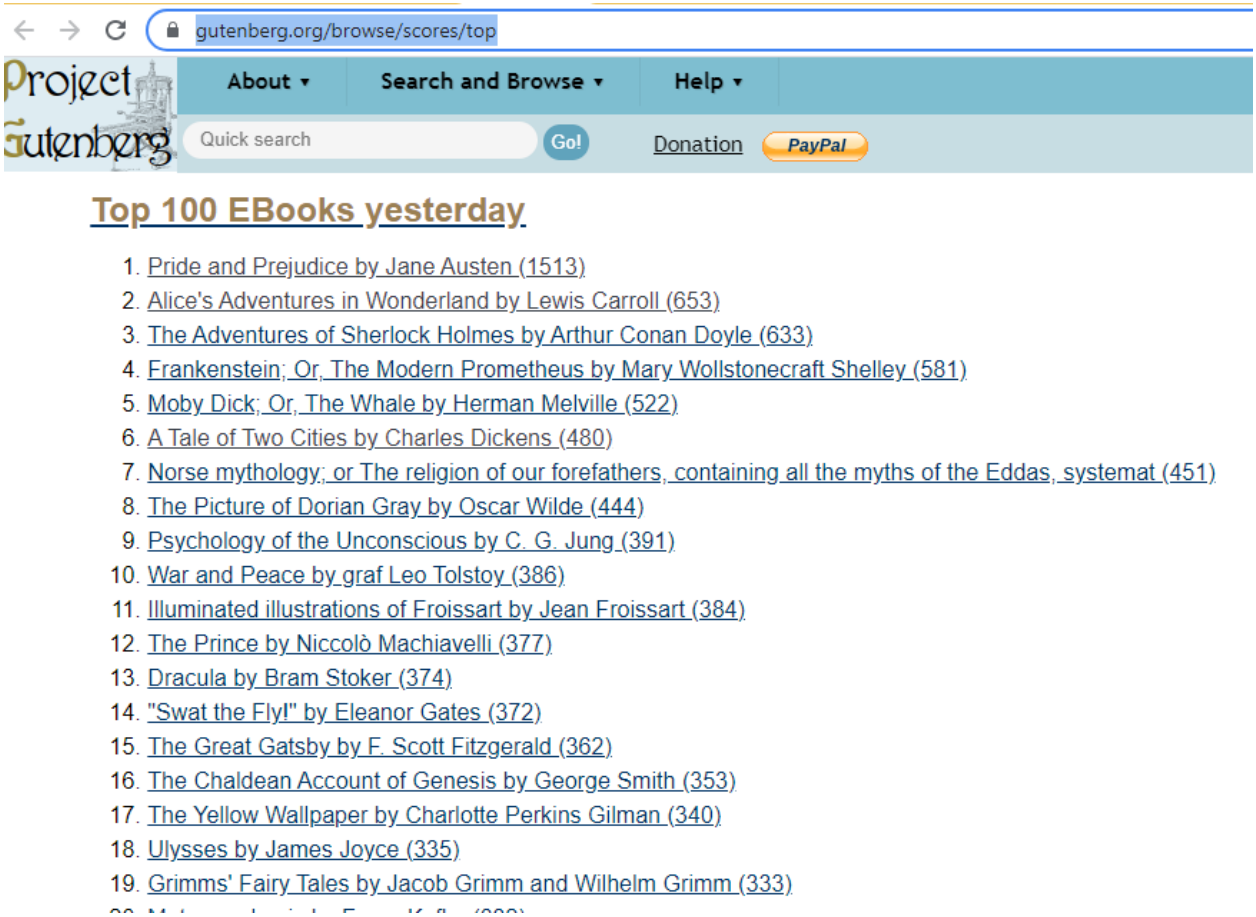
Task: You will use sequential text data and convert the text data into vectors and feed these vectors into the RNN – GRU network and build a neural network model which will use the preceding 3 words to predict the next word.

Downloading the Dataset:

You can download the dataset from the below URL.

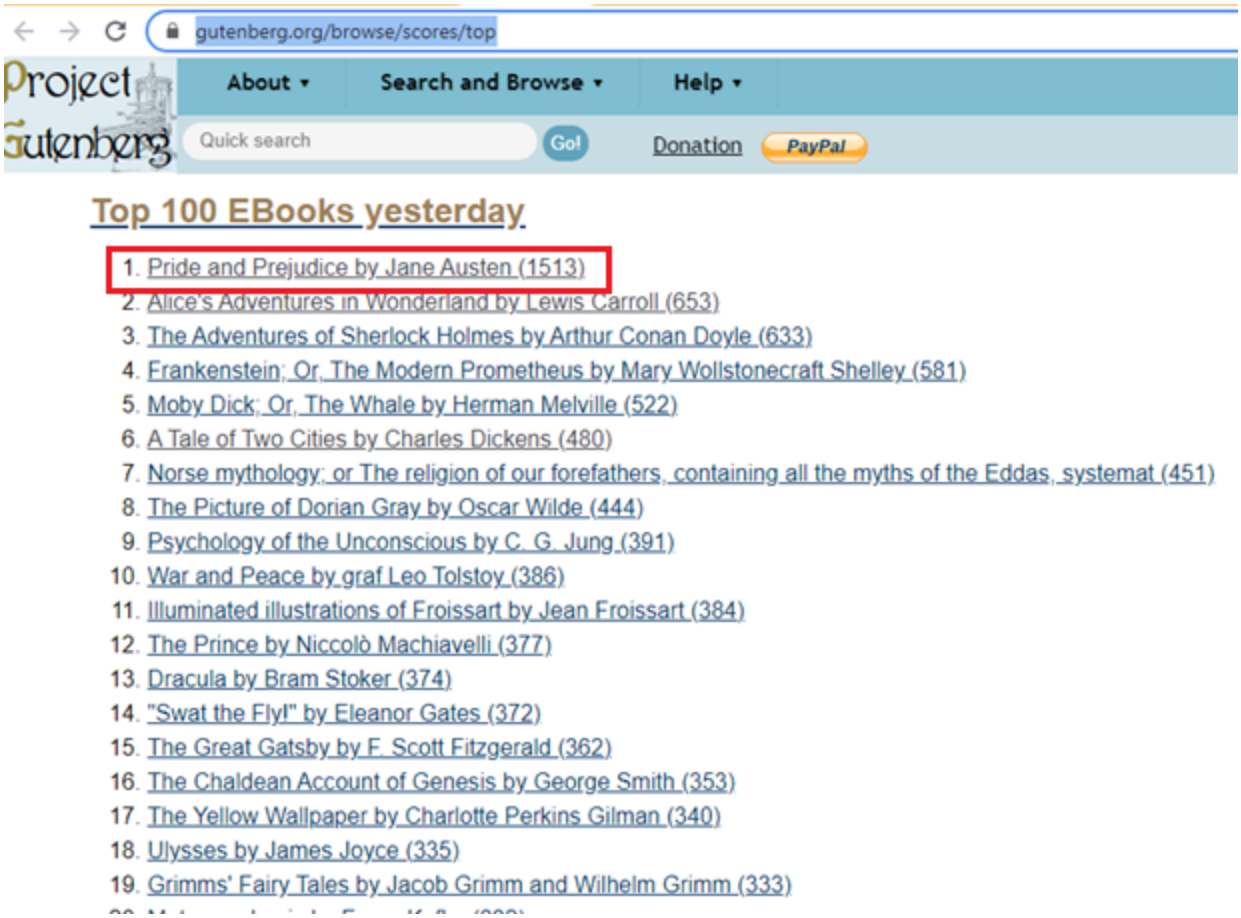
https://www.gutenberg.org/browse/scores/top

Once you click the URL and open the webpage and scroll down a little, you will find the below screen,

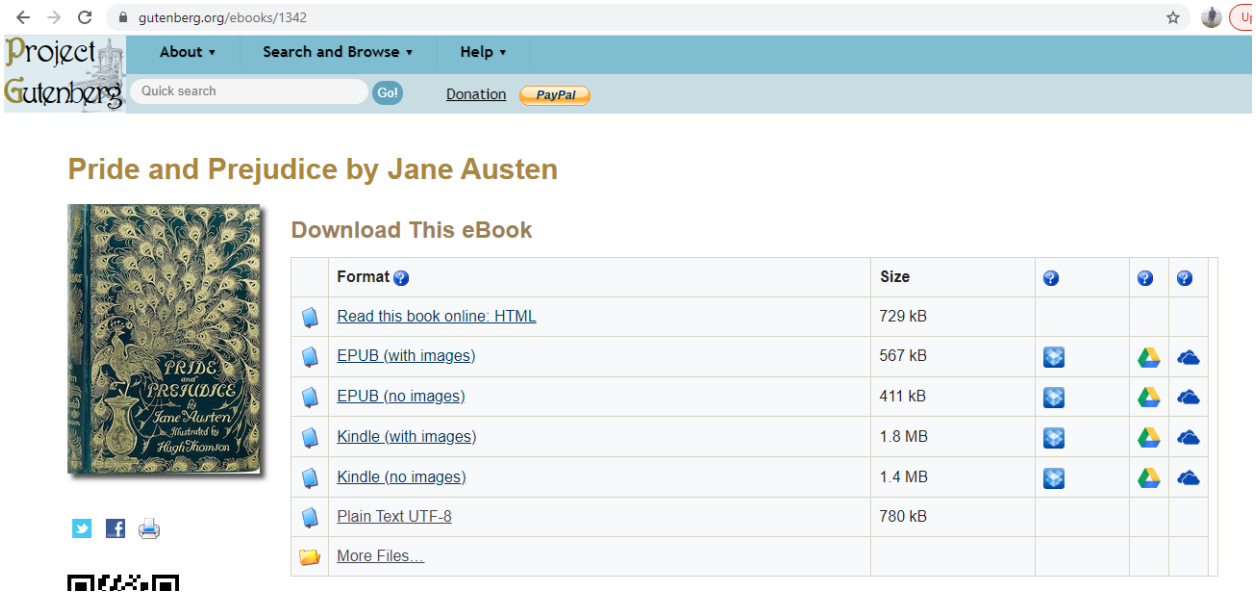


There are multiple text documents available and you can choose any of them to practice, we are using **“Pride and Prejudice by Jane Austen”** for now.

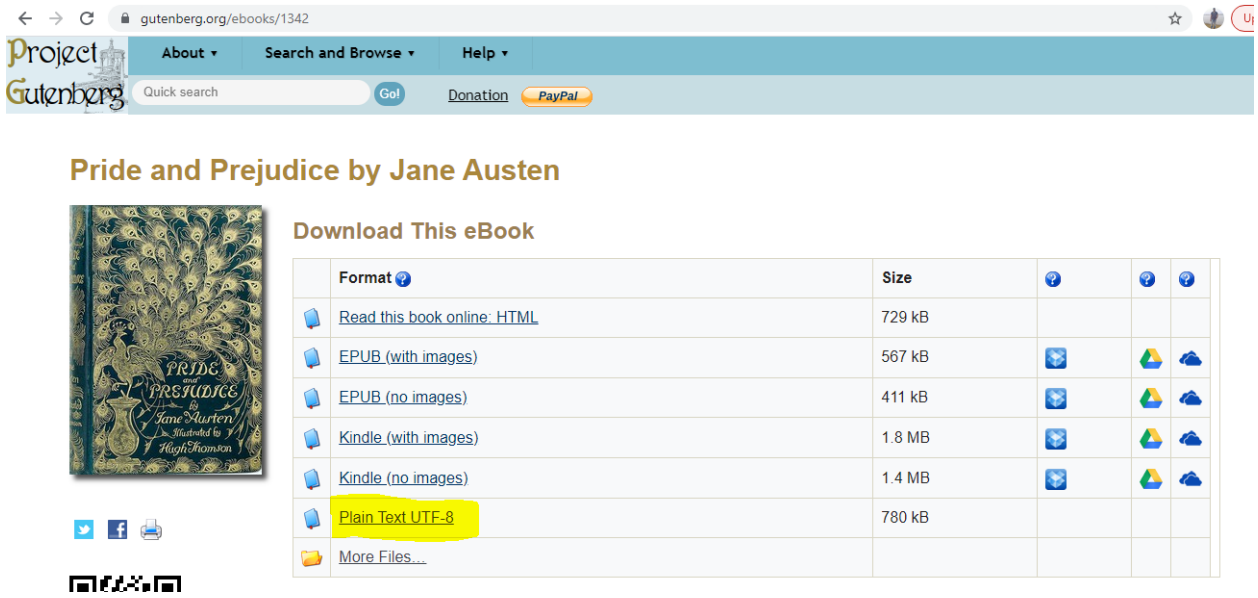
Click on **“Pride and Prejudice by Jane Austen”**



Upon clicking on **"Pride and Prejudice by Jane Austen",** it will lead you to the below page,



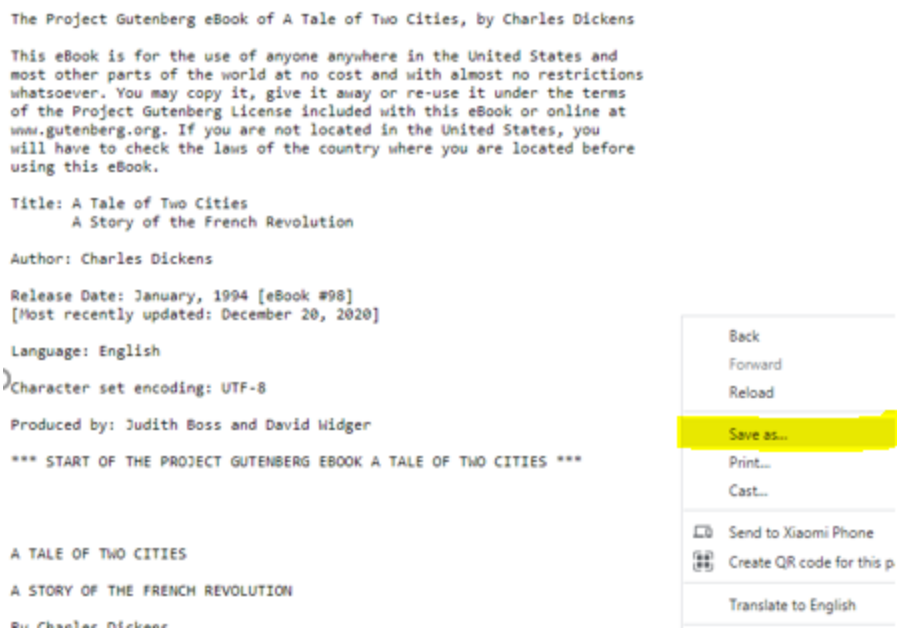
Now, click on "**Plain Text UTF -8",**



After clicking on "Plain Text UTF – 8", you will find the text document opened as below,

```
The Project Gutenberg eBook of Pride and Prejudice, by Jane Austen

This eBook is for the use of anyone anywhere in the United States and
most other parts of the world at no cost and with almost no restrictions
whatsoever. You may copy it, give it away or re-use it under the terms
of the Project Gutenberg License included with this eBook or online at
www.gutenberg.org. If you are not located in the United States, you
will have to check the laws of the country where you are located before
using this eBook.

Title: Pride and Prejudice

Author: Jane Austen

Release Date: June, 1998 [eBook #1342]
[Most recently updated: February 10, 2021]

Language: English

Character set encoding: UTF-8

Produced by: Anonymous Volunteers and David Widger

*** START OF THE PROJECT GUTENBERG EBOOK PRIDE AND PREJUDICE ***

THERE IS AN ILLUSTRATED EDITION OF THIS TITLE WHICH MAY VIEWED AT EBOOK
[# 42671 ]

cover



      Pride and Prejudice

      By Jane Austen
```

You can Right Click and save the text document into your local drive, you will use this file for practice.



The sequential text data is ready now, let us start implementing step by step using Google Colab. (You may also use your local machine if you are working on a GPU)

Part1: Load the required Packages and dataset.

Step 1:

Load the packages,

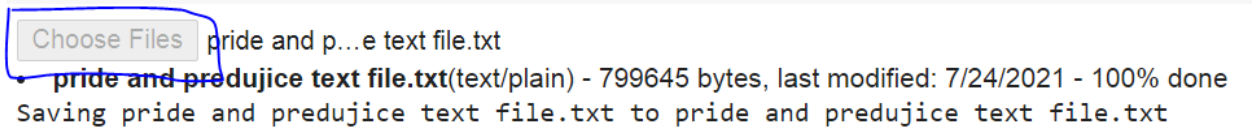
```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.layers import Embedding, LSTM, Dense, GRU
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam
import pickle
import numpy as np
import os
```

Step 2:

Load the Sequential text data which was downloaded earlier into your local drive. i.e. "Pride and Prejudice by Jane Austen"

```
# Load the text file into colab workspace
from google.colab import files
uploaded = files.upload()
```

Upon executing the above code you will find the below button i.e. "Choose Files" appearing on the screen, you can click on it to upload the Text file/Data from your local drive.



Part 2: Data Preprocessing

You have already studied that RNN will take inputs in the form of vectors, thus we need to convert the text data into Vectors, to do that we first need to eliminate unwanted stuff from the textual dataset.

You will do the pre-processing step-by-step,

- **Open the text file in reading mode,**
- **Store the file in the form of a list,**
- **Convert the list into a string,**
- **Replace unnecessary stuff with space,**
- **Finally, remove the unwanted spaces.**

Step 3:

Open File in reading mode,

```
# Open the file in read mode
file = open("pride and predujice text file.txt", "r", encoding = "utf8")
```

Step 4:

Store file in the form of a list,

```
# store file in list
lines = []
for i in file:
    lines.append(i)
```

Step 5:

Covert list to String,

```
# Convert list to string
data = ""
for i in lines:
  data = ' '. join(lines)
```

Step 6:

Replace unnecessary stuff with spaces,

```
#replace unnecessary stuff with space
data = data.replace('\n', '').replace('\r', '').replace('\ufeff', '').replace('“','').replace('”','')
```

Step 7:

Remove unnecessary spaces,

```
#remove unnecessary spaces
data = data.split()
data = ' '.join(data)
```

Step 8:

Let us check the initial 500 fields from the pre-processed data,

```
# Validate the preprocessed data by checking the initial 500 fields
data[:500]
```

You can see that the unwanted stuff is eliminated,

```
'The Project Gutenberg eBook of Pride and Prejudice, by Jane Austen This eBook is for the use of anyon
e anywhere in the United States and most other parts of the world at no cost and with almost no restri
ctions whatsoever. You may copy it, give it away or re-use it under the terms of the Project Gutenberg
License included with this eBook or online at www.gutenberg.org. If you are not located in the United
States, you will have to check the laws of the country where you are located before using th'
```

Step 9:

Check the length of the data, to know the number of words in the dataset.

```
# Check the length of the data
len(data)
```

```
698483
```

Part 3: Tokenization of data – Converting text to sequences

Step 10:

You already studied that you need vectors to feed into the network, the input data is in the form of text, you will need to convert them into vectors now before building the GRU model.

With Keras "Tokenizer" you can easily convert the sequential text data into vectors, you will do it step-by-step.

- **Initiate a tokenizer object,**
- **Save the tokenizer object into a pickle file,**
- **Finally, convert text data into vectors.**

```
# Initiate a tokenizer object
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data]) # Fit the preprocessed data into the above initiated Tokenizer object

# save the tokenizer into a pickle file
pickle.dump(tokenizer, open('token.pkl', 'wb'))

# Convert the textual data into vectors i.e. neumerical values
sequence_data = tokenizer.texts_to_sequences([data])[0]
```

While saving the tokenizer object, **'wb'** means write binary used to handle the file, **token.pkl** is giving the file name.

'texts_to_sequences' converts text data into vectors, i.e. numerical sequences.

Step 11:

Now check if the Text is converted to vectors or not by calling the first 15 fields,

```
sequence_data[:15]
```

This line of code will return the below output,

```
[1, 176, 157, 916, 3, 321, 4, 1174, 30, 72, 2535, 41, 916, 23, 21]
```

Each number represents a word.

Step 12:

Let us check the length of the vectors now,

```
# Check the length of the sequential data
len(sequence_data)
```

```
125317
```

The length of the sequential data is lesser than the actual data because each unique value will get a numerical representation, thus all the repeated values will get the label of the same numerical value, also white spaces and punctuations are omitted.

Step 13:

Check how many unique words are present in the data,

```
# let us check how many unique words are present in the data
vocab_size = len(tokenizer.word_index) + 1  # use +1 because 0 will be used for padding
print(vocab_size)
```

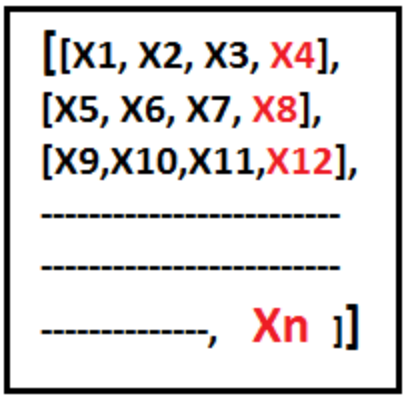
This piece of code will return the total number of unique words in the dataset,

```
7036
```

Step 14:

The goal is to predict the next word when the preceding three words are given as the input.

To achieve the goal you will need the vectors in the below form of a matrix with 4 columns in each row where the first 3 values are the input and the fourth value is the output in each row.



You can do it with the help of the below chunk of code,

```
# Initiate a blank list
sequences = []

# Iterate the sequence data into the above blank list
for i in range(3, len(sequence_data)):
    words = sequence_data[i-3:i+1] # Will use previous 3 words to predict the next word
    sequences.append(words)
   # Print the length of sequences
print("The Length of sequences are: ", len(sequences))
sequences = np.array(sequences) # Convert the sequences into array
sequences[:10] # print the initial 10 sequences
```

It will return the below output,

```
The Length of sequences are:  140773
array([[   1,  209,  192, 1027],
       [ 209,  192, 1027,    3],
       [ 192, 1027,    3,    5],
       [1027,    3,    5, 1907],
       [   3,    5, 1907,    3],
       [   5, 1907,    3,   77],
       [1907,    3,   77, 1908],
       [   3,   77, 1908,   32],
       [  77, 1908,   32,  189],
       [1908,   32,  189, 3242]])
```

In the above matrix of sequences, the first 3 words are the inputs and the fourth word is the output in each line.

Step 15:

Separate the dependent and independent data from the above step, as the first 3 words are the independent features and the 4th word is the dependent feature.

```
# Split the dependent and independent features
# initiate empty lists for x and y features
X = []
y = []
# Append the first 3 words into the x feature and the last word into the y feature
for i in sequences:
    X.append(i[0:3])
    y.append(i[3])
# Convert the above appended lists into arrays
X = np.array(X)
y = np.array(y)
```

Step 16:

Convert the vectors into a binary class matrix and check the first 5 fields,

```
# Convert the vectors into binary class matrix
y = to_categorical(y, num_classes=vocab_size)
# print the first 5 fields
y[:5]
```

It will return the first 5 fields of the categorical variable.

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

Part 4: Build the model

Step 17:

Initialize the neural network model and create the layers of the neural network.

```
# Initiate a Sequential Deep Network
model = Sequential()
# Create an embedding layer which takes 3 arguments as input
model.add(Embedding(vocab_size, 10, input_length=3))
# Vocab_size = input dim which is the size of vocabulary in the text data,
# 10 = Output dim which is the size of the vector space in which words will be embedded
# add a GRU layer
model.add(GRU(500, return_sequences=True))
# 500 is the dimensionality given to the output space,
# Return_sequences=True is to create next GRU layer
# add the second GRU layer
model.add(GRU(500))
# add a Dense Layer
model.add(Dense(1000, activation="relu"))
# add a second Dense Layer
model.add(Dense(vocab_size, activation="softmax"))
```

Step 18:

Check the layers created in the above step, you can have the overview by executing the below line of code,

```
# Check the layers created above by printing the summary of the model
model.summary()
```

Output:

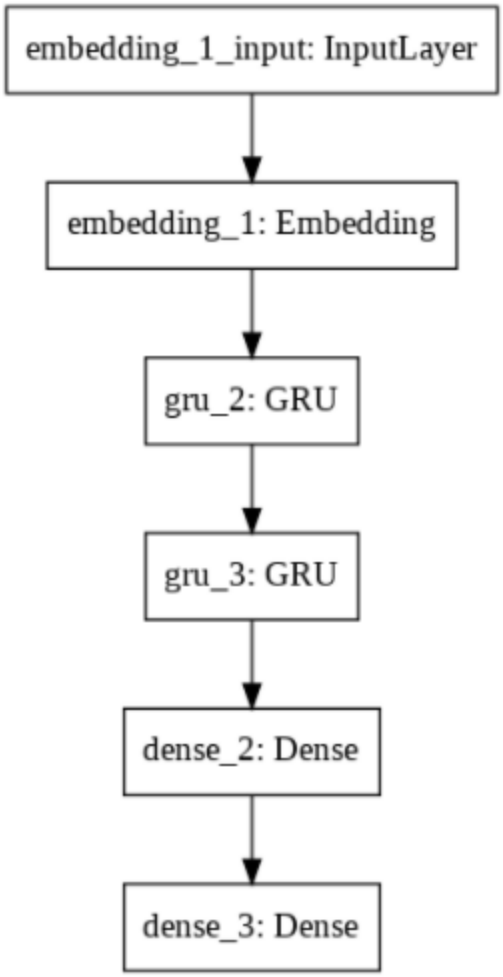
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 3, 10)             70360
_________________________________________________________________
gru_2 (GRU)                  (None, 3, 500)            768000
_________________________________________________________________
gru_3 (GRU)                  (None, 500)               1503000
_________________________________________________________________
dense_2 (Dense)              (None, 1000)              501000
_________________________________________________________________
dense_3 (Dense)              (None, 7036)              7043036
=================================================================
Total params: 9,885,396
Trainable params: 9,885,396
Non-trainable params: 0
_________________________________________________________________
```

Step 19:

You can visualize the outline of the layers in the model using the below line of code,

```
# Vizualise the model
from tensorflow import keras
from keras.utils.vis_utils import plot_model

keras.utils.plot_model(model, to_file='plot.png', show_layer_names=True)
```

Output:



Step 20:

Compile and fit the model,

```
# Compile and fit the model i.e train the model
from tensorflow.keras.callbacks import ModelCheckpoint

checkpoint = ModelCheckpoint("next_words.h5", monitor='loss', verbose=1, save_best_only=True)
model.compile(loss="categorical_crossentropy", optimizer=Adam(learning_rate=0.001))
model.fit(X, y, epochs=70, batch_size=64, callbacks=[checkpoint])
```

Step 21:

Define a simple function to predict the next word,

```
from tensorflow.keras.models import load_model
import numpy as np
import pickle
```

```
# Load the model and tokenizer
model = load_model('next_words.h5')
tokenizer = pickle.load(open('token.pkl', 'rb'))

def Predict_Next_Words(model, tokenizer, text):

  sequence = tokenizer.texts_to_sequences([text])
  sequence = np.array(sequence)
  preds = np.argmax(model.predict(sequence))
  predicted_word = ""

  for key, value in tokenizer.word_index.items():
      if value == preds:
          predicted_word = key
          break

  print(predicted_word)
  return predicted_word
```

Step 22:

Create a loop that creates a blank box, where you can input 3 words and the created model will predict the next word,

```
while(True):
  text = input("Enter your line: ")

  if text == "0":
      print("Execution completed.....")
      break

  else:
      try:
          text = text.split(" ")
          text = text[-3:]
          print(text)

          Predict_Next_Words(model, tokenizer, text)

      except Exception as e:
        print("Error occurred: ",e)
        continue
```

Upon executing the above loop, Python will return the below output,

You can now key in any three words in a sequence from the original data downloaded and the model will output the 4th word for you.

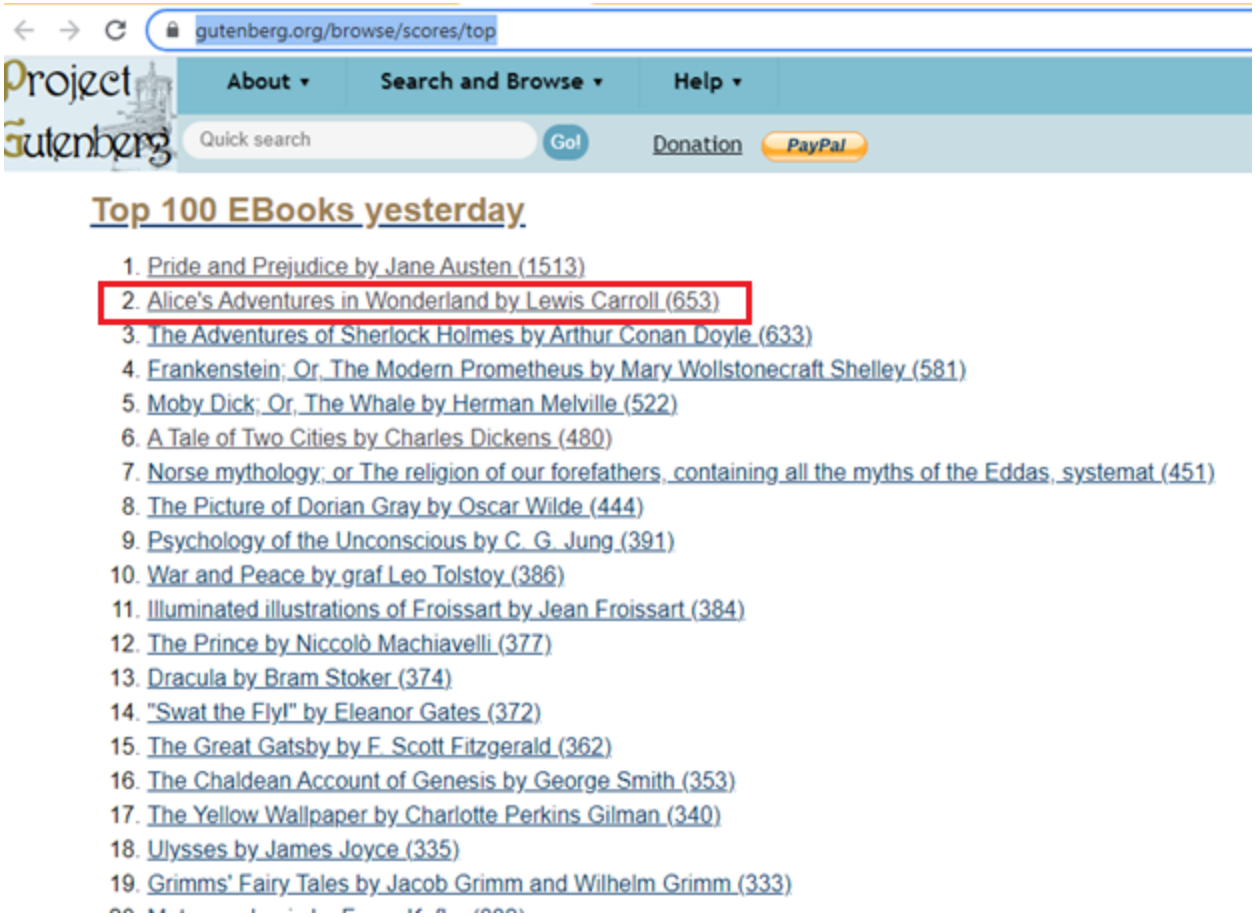
```
Enter your line: Long says that
['Long', 'says', 'that']
netherfield
```

“Long says that” are the 3 sequential words from the text document which are keyed-in and the model returned the 4th word i.e. Netherfield, for your validation below is a snap attached from the text document on which the model is built.

“Why, my dear, you must know, Mrs. Long says that Netherfield is taken by a young man of large fortune from the north of England; that he came down on Monday in a chaise and four to see the place, and was so much delighted with it, that he agreed with Mr. Morris immediately; that he is to take possession before Michaelmas, and some of his servants are to be in the house by the end of next week.”

PROGRAMMING ASSIGNMENT:

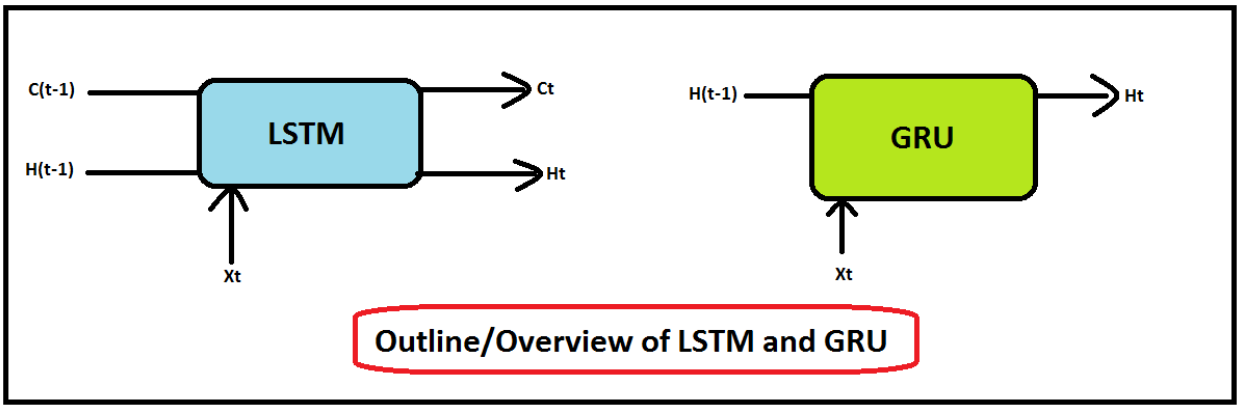
Download “Alice’s Adventures in Wonderland by Lewis Carroll “text file and create a GRU model which will take 3 sequential words as input and predict the next word.



Hint: You can replicate all the steps shown in the above practical implementation, except for steps 2 and 3, where you will need to replace **“Pride and Prejudice by Jane Austen”.** With **“Alice’s Adventures in Wonderland by Lewis Carroll “**

SUMMARY:

GRU - You can understand GRU as a smaller version of the LSTM network, It is comparatively faster than LSTM, as LSTM uses 3 Gates namely, Forget Gate, Input Gate, Output Gate along with 2 states i.e. Cell State (Long Term Memory) and the hidden state (Short Term Memory), the GRU network uses only 2 Gates namely, Reset Gate and the Update Gate and possesses only one state i.e. Hidden State.



ASSESSMENT

Choose the appropriate option:

1) GRU models are simplified/special version of :

A. Artificial Neural Networks.
B. LSTM Networks.
C. None of the above.

2) How many Modules/Gates are present in GRU Architecture?

A. 2
B. 4
C. 6
D. 8

3) GRU is a special kind of:

A. Feed-Forward Network.
B. Recurrent Neural Network.
C. Both of the above.

4) The Reset Gate of the GRU network refers to the:

A. Long Term Memory
B. Short term memory

5) GRU Model does not possess:

A. Cell State
B. Hidden State
C. Both of the above.

Fill in the Spaces:

1) GRU is considered to be similar to LSTM, though not the same, the computation is lighter and faster because it uses _____ Gates in its architecture.
2) The GRU architecture mainly comprises of _____, _____, _____.
3) _____ And _____ are the two steps of the computing the hidden state in a GRU network.
4) _____function is used in the computation of $r_{(t)}$ and $Z_{(t)}$.
5) GRU is a special kind of _____ Neural Networks.

True or False:

1) **To implement a GRU model using python you need to install and import Keras and GRU Packages.**

A. True
B. False

2) **In GRU models, the input data is not in the form of vectors.**

A. True
B. False

3) **Sigmoid function converts the values of r(t) and Z(t) into o & 1.**

A. True
B. False

4) **The Keras-Tokenizer helps in converting Text to Sequences.**

A. True
B. False

5) **The Reset Gate Decides how much past information to forget.**

A. True
B. False

Solutions for Assessment

Choose the appropriate options answers

1) B
2) A
3) B
4) B
5) A

Fill in the spaces with appropriate answers

1) Two.
2) Reset Gate, Update Gate, Hidden State.
3) Computation of Candidate Hidden State and Computation of Current hidden state.
4) Sigmoid.
5) Recurrent.

True or False

1) True.
2) False.
3) True.
4) True.
5) True.