Deep Learning

Part I

Biological and Artificial Neuron,
Perceptron and its learning rule and drawbacks,
Multilayer Perceptron,
Training MLP: Backpropagation ,
Loss function ,
Activation Functions ,
Introduction to Tensorflow and Keras, Vanishing and Exploding Gradient Problem

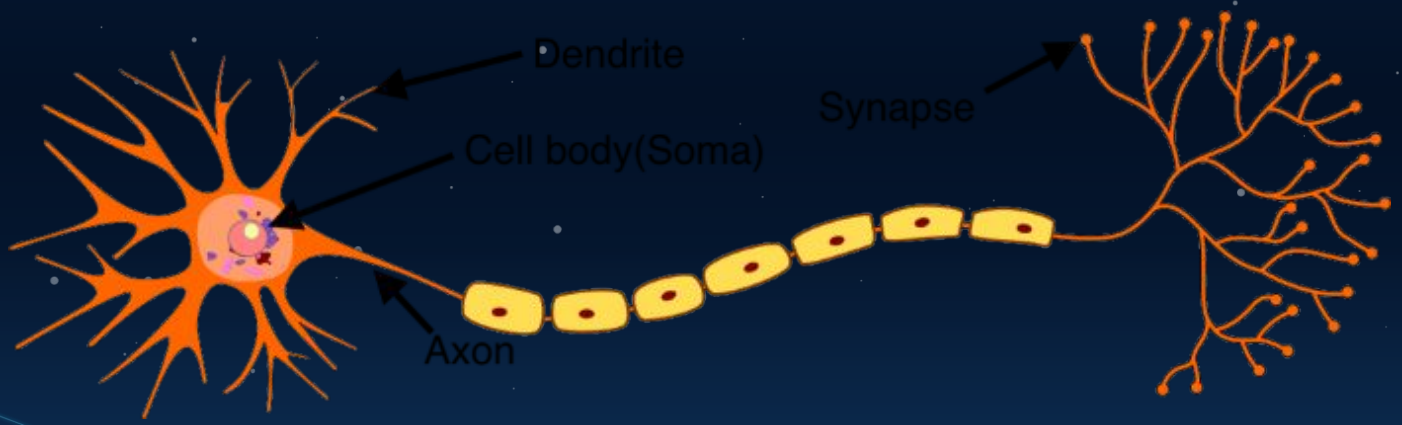
Biological Neurons

Neurons are the basic functional units of the nervous system, and they generate electrical signals called action potentials, which allows them to quickly transmit information over long distances.
Almost all the neurons have three basic functions essential for the normal functioning of all the cells in the body.

These are to:
1. Receive signals (or information) from outside.
2. Process the incoming signals and determine whether or not the information should be passed along.
3. Communicate signals to target cells which might be other neurons or muscles or glands.

Diagram of Biological Neurons

A biological neuron is mainly composed of **3 main parts and an external part called synapse**:-



Basic parts of a neuron

1. Dendrite

Dendrites are responsible for getting incoming signals from outside

2. Soma

Soma is the cell body responsible for the processing of input signals and deciding whether a neuron should fire an output signal

3. Axon

Axon is responsible for getting processed signals from neuron to relevant cells

4. Synapse

Synapse is the connection between an axon and other neuron dendrites

Working of Biological Neurons

The task of receiving the incoming information is done by dendrites, and processing generally takes place in the cell body. Incoming signals can be either *excitatory – which means they tend to make the neuron fire* (generate an electrical impulse) – *or inhibitory – which means that they tend to keep the neuron from firing.*

Most neurons receive many input signals throughout their dendritic trees. A single neuron may have more than one set of dendrites and may receive many thousands of input signals. Whether or not a neuron is excited into firing an impulse depends on the sum of all of the excitatory and inhibitory signals it receives. The processing of this information happens in soma which is neuron cell body. If the neuron does end up firing, the nerve impulse, or action potential, is conducted down the axon.

Towards its end, the axon splits up into many branches and develops bulbous swellings known as axon terminals (or nerve terminals). These axon terminals make connections on target cells.

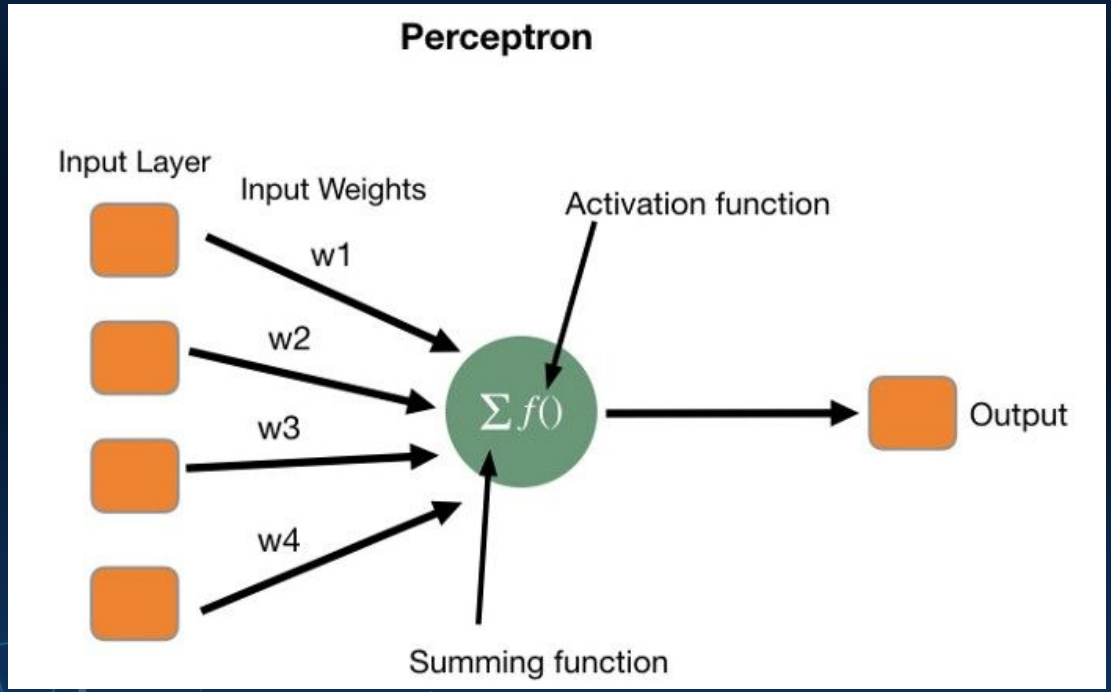
Artificial Neurons

Artificial neuron also known as perceptron is the basic unit of the neural network. In simple terms, it is a mathematical function based on a model of biological neurons. It can also be seen as a simple logic gate with binary outputs. They are sometimes also called perceptrons.

Each artificial neuron has the following main functions:

1. Takes inputs from the input layer
2. Weighs them separately and sums them up
3. Pass this sum through a nonlinear function to produce output.

Artificial Neurons



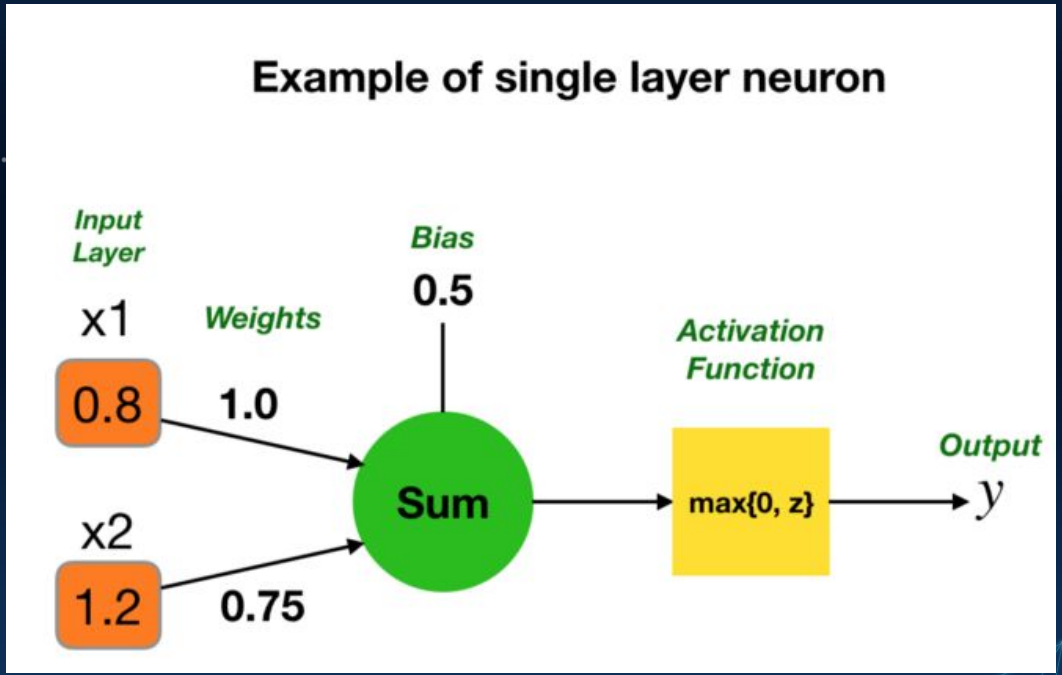
Artificial Neurons

The perceptron(neuron) consists of 4 parts:

1. Input values or One input layer
   We pass input values to a neuron using this layer. It might be something as simple as a collection of array values. It is similar to a dendrite in biological neurons.
2. Weights and Bias
   Weights are a collection of array values which are multiplied to the respective input values. We then take a sum of all these multiplied values which is called a weighted sum. Next, we add a bias value to the weighted sum to get final value for prediction by our neuron.
3. Activation Function
   Activation Function decides whether or not a neuron is fired. It decides which of the two output values should be generated by the neuron.
4. Output Layer
   Output layer gives the final output of a neuron which can then be passed to other neurons in the network or taken as the final output value.

Working of an artificial neuron with an example

Consider a neuron with two inputs (x1,x2) as shown below:



Working of an artificial neuron with an example

1. The values of the two inputs(x1,x2) are 0.8 and 1.2
2. We have a set of weights (1.0,0.75) corresponding to the two inputs
3. Then we have a bias with value 0.5 which needs to be added to the sum

The input to activation function is then calculated using the formula:-

$$C = w1 * x1 + w2 * x2 + b$$
$$= (1 * 0.8) + (0.75 * 1.2) + 0.5$$
$$= 0.8 + 0.9 + 0.5$$
$$= 2.2$$

Working of an artificial neuron with an example

Now the combination(C) can be fed to the activation function. Let us first understand the logic of Rectified linear (ReLU) activation function which we are currently using in our example:

$$activation = \begin{cases} 0 & if \quad combination < 0 \\ combination & if \quad combination \geq 0 \end{cases}$$

ReLU activation function

In our case, the combination value we got was 2.2 which is greater than 0 so the output value of our activation function will be 2.2.
This will be the final output value of our single layer neuron.

Biological Neuron vs. Artificial Neuron

Biological Neuron	Artificial Neuron
Dendrites	Input
Cell Nucleus(Soma)	Node
Axon	Output
Synapse	Interconnections

Characteristics Of Artificial Neuron

The artificial neuron has the following characteristics:

- A neuron is a mathematical function modeled on the working of biological neurons
- It is an elementary unit in an artificial neural network
- One or more inputs are separately weighted
- Inputs are summed and passed through a nonlinear function to produce output
- Every neuron holds an internal state called activation signal
- Each connection link carries information about the input signal
- Every neuron is connected to another neuron via connection link

Perceptron

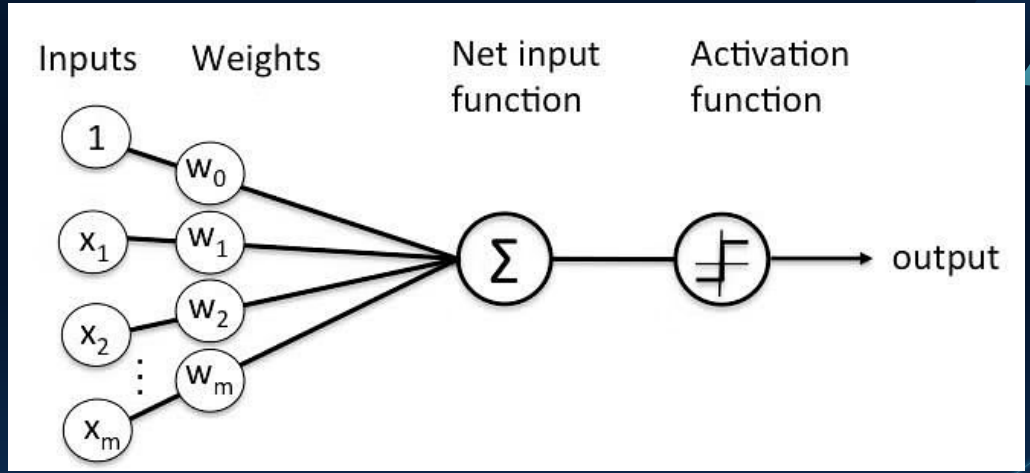
Frank Rosenblatt (1928 – 1971) was an American psychologist notable in the field of Artificial Intelligence.In 1957 he started something really big. He "invented" a Perceptron program, on an IBM 704 computer at Cornell Aeronautical Laboratory.
Scientists had discovered that brain cells (Neurons) receive input from our senses by electrical signals.
The Neurons, then again, use electrical signals to store information, and to make decisions based on previous input.
Frank had the idea that Perceptrons could simulate brain principles, with the ability to learn and make decisions.

A Perceptron is an Artificial Neuron.It is the simplest possible Neural Network. Neural Networks are the building blocks of Machine Learning.

Perceptron

The original Perceptron was designed to take a number of binary inputs, and produce one binary output (0 or 1).

The idea was to use different weights to represent the importance of each input, and that the sum of the values should be greater than a threshold value before making a decision like true or false (0 or 1).



Example of Perceptron

Imagine a perceptron (in your brain).
The perceptron tries to decide if you should go to a concert.
Is the artist good? Is the weather good?
What weights should these facts have?

Criteria	Input	Weight
Artists is Good	**x1** = 0 or 1	**w1** = 0.7
Weather is Good	**x2** = 0 or 1	**w2** = 0.6
Friend will Come	**x3** = 0 or 1	**w3** = 0.5
Food is Served	**x4** = 0 or 1	**w4** = 0.3
Alcohol is Served	**x5** = 0 or 1	**w5** = 0.4

Algorithm of Perceptron

Frank Rosenblatt suggested this algorithm:

1. Set a threshold value
2. Multiply all inputs with its weights
3. Sum all the results
4. Activate the output

1. Set a threshold value:

- Threshold = 1.5

2. Multiply all inputs with its weights:

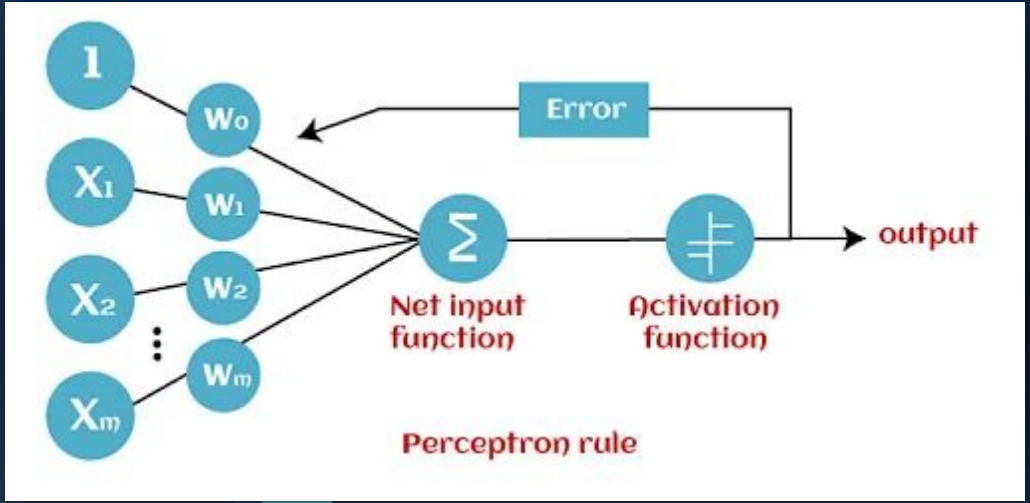
- x1 * w1 = 1 * 0.7 = 0.7
- x2 * w2 = 0 * 0.6 = 0
- x3 * w3 = 1 * 0.5 = 0.5
- x4 * w4 = 0 * 0.3 = 0
- x5 * w5 = 1 * 0.4 = 0.4

Algorithm of Perceptron

3. Sum all the results:
   - 0.7 + 0 + 0.5 + 0 + 0.4 = 1.6 (The Weighted Sum)
4. Activate the Output:
   - Return true if the sum > 1.5 ("Yes I will go to the Concert")

Note
If the weather weight is 0.6 for you, it might different for someone el
higher weight means that the weather is more important to them.
If the threshold value is 1.5 for you, it might be different for someone else
A lower threshold means they are more wanting to go to the concert.

Mathematical approach



Well Perceptron is considered a single-layer neural link with four main parameters. The perceptron model begins with multiplying all input values and their weights, then adds these values to create the weighted sum. Further, this weighted sum is applied to the activation function ‘f’ to obtain the desired output. This activation function is also known as the step function and is represented by ‘f.’

Mathematical approach

This step function or Activation function is vital in ensuring that output is mapped between (0,1) or (-1,1). Take note that the weight of input indicates a node's strength. Similarly, an input value gives the ability the shift the activation function curve up or down.

Step 1: Multiply all input values with corresponding weight values and then add to calculate the weighted sum. The following is the mathematical expression of it:

$$\sum wi*xi = x1*w1 + x2*w2 + x3*w3 + \ldots\ldots x4*w4$$

Add a term called bias 'b' to this weighted sum to improve the model's performance.

Step 2: An activation function is applied with the above-mentioned weighted sum giving us an output either in binary form or a continuous value as follows:

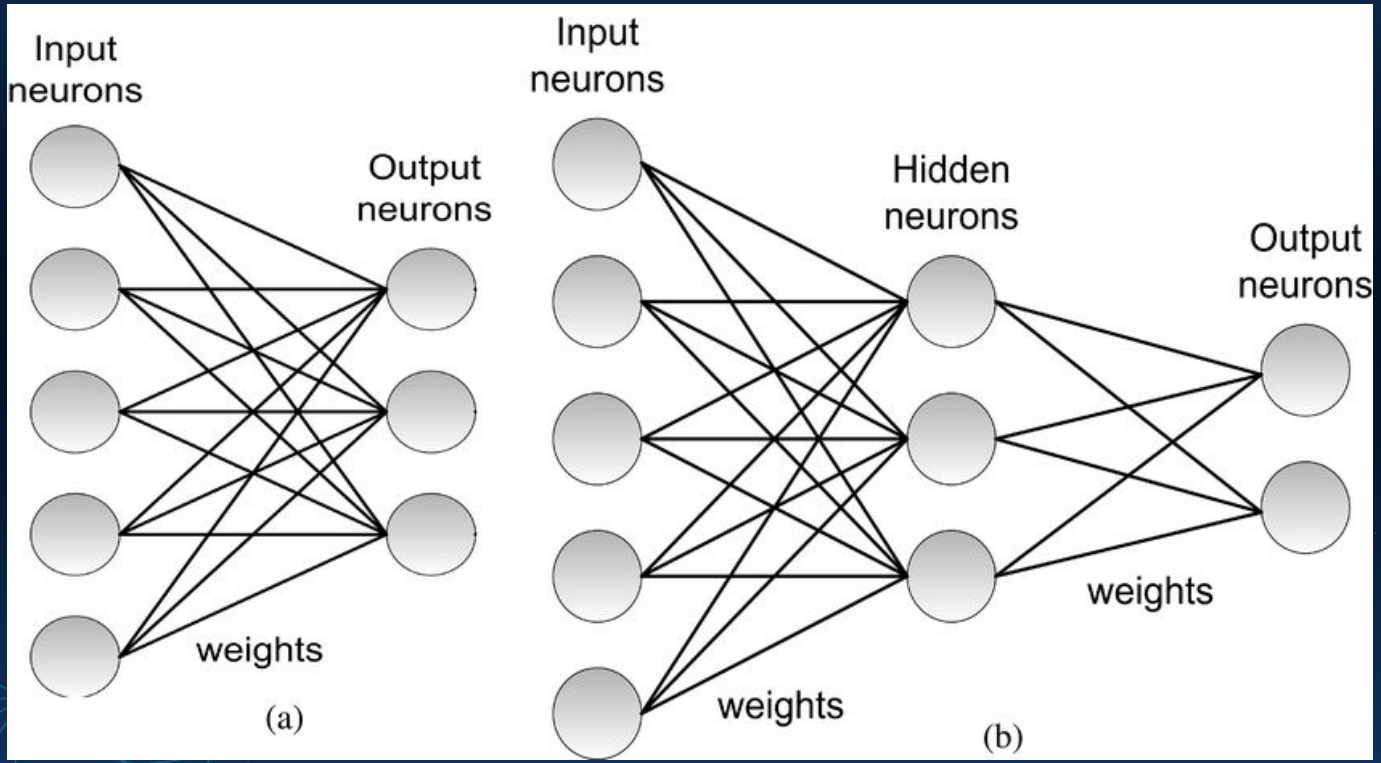
$$Y = f(\sum wi*xi + b).$$

Types of Perceptron models

1. Single Layer Perceptron model: One of the easiest ANN(Artificial Neural Networks) types consists of a feed-forward network and includes a threshold transfer inside the model. The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes. A Single-layer perceptron can learn only linearly separable patterns.

2. Multi-Layered Perceptron model: It is mainly similar to a single-layer perceptron model but has more hidden layers.

a) Single layer b)Multi Layer



Types of Perceptron models

Forward Stage: From the input layer in the on stage, activation functions begin and terminate on the output layer.

Backward Stage: In the backward stage, weight and bias values are modified per the model's requirement. The backstage removed the error between the actual output and demands originating backward on the output layer.

A multilayer perceptron model has a greater processing power and can process linear and non-linear patterns. Further, it also implements logic gates such as AND, OR, XOR, XNOR, and NOR.

Advantages of Multi Layer

- A multi-layered perceptron model can solve complex non-linear problems.
- It works well with both small and large input data.
- Helps us to obtain quick predictions after the training.
- Helps us obtain the same accuracy ratio with big and small data.

Disadvantages of Multi Layer

- In multi-layered perceptron model, computations are time-consuming and complex.
- It is tough to predict how much the dependent variable affects each independent variable.
- The model functioning depends on the quality of training.

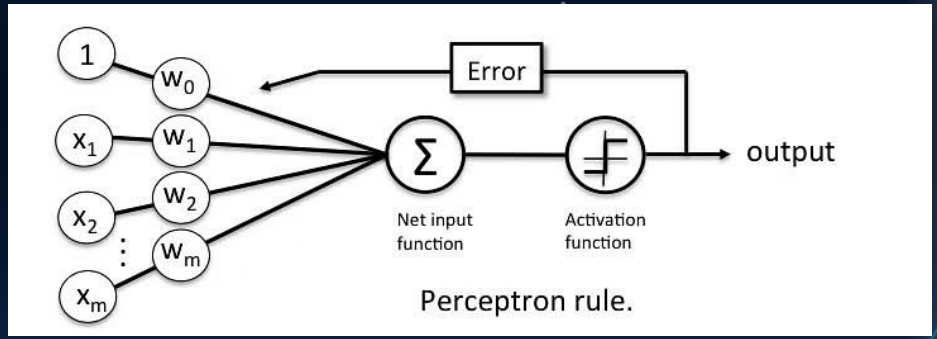
Characteristics of the Perceptron Model

1. It is a machine learning algorithm that uses supervised learning of binary classifiers.
2. In Perceptron, the weight coefficient is automatically learned.
3. Initially, weights are multiplied with input features, and then the decision is made whether the neuron is fired or not.
4. The activation function applies a step rule to check whether the function is more significant than zero.
5. The linear decision boundary is drawn, enabling the distinction between the two linearly separable classes +1 and -1.
6. If the added sum of all input values is more than the threshold value, it must have an output signal; otherwise, no output will be shown.

Perceptron Learning Rule

Perceptron Learning Rule states that the algorithm would automatically learn the optimal weight coefficients. The input features are then multiplied with these weights to determine if a neuron fires or not.

The Perceptron receives multiple input signals, and if the sum of the input signals exceeds a certain threshold, it either outputs a signal or does not return an output. In the context of supervised learning and classification, this can then be used to predict the class of a sample.



Perceptron Terminology

Perceptron Function

Perceptron is a function that maps its input “x,” which is multiplied with the learned weight coefficient; an output value ”f(x)”is generated.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

In the equation given above:

- “w” = vector of real-valued weights
- “b” = bias (an element that adjusts the boundary away from origin without any dependence on the input value)
- “x” = vector of input x values

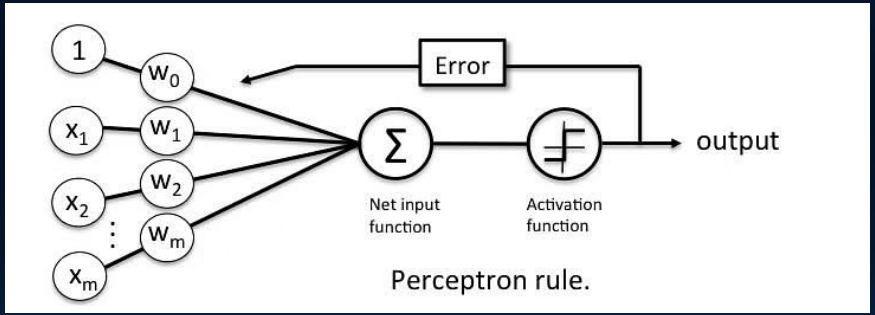
$$\sum_{i=1}^{m} w_i x_i$$

- “m” = number of inputs to the Perceptron

The output can be represented as “1” or “0.” It can also be represented as “1” or “-1” depending on which activation function is used.

Perceptron Input

A Perceptron accepts inputs, moderates them with certain weight values, then applies the transformation function to output the final result. The image below shows a Perceptron with a Boolean output.

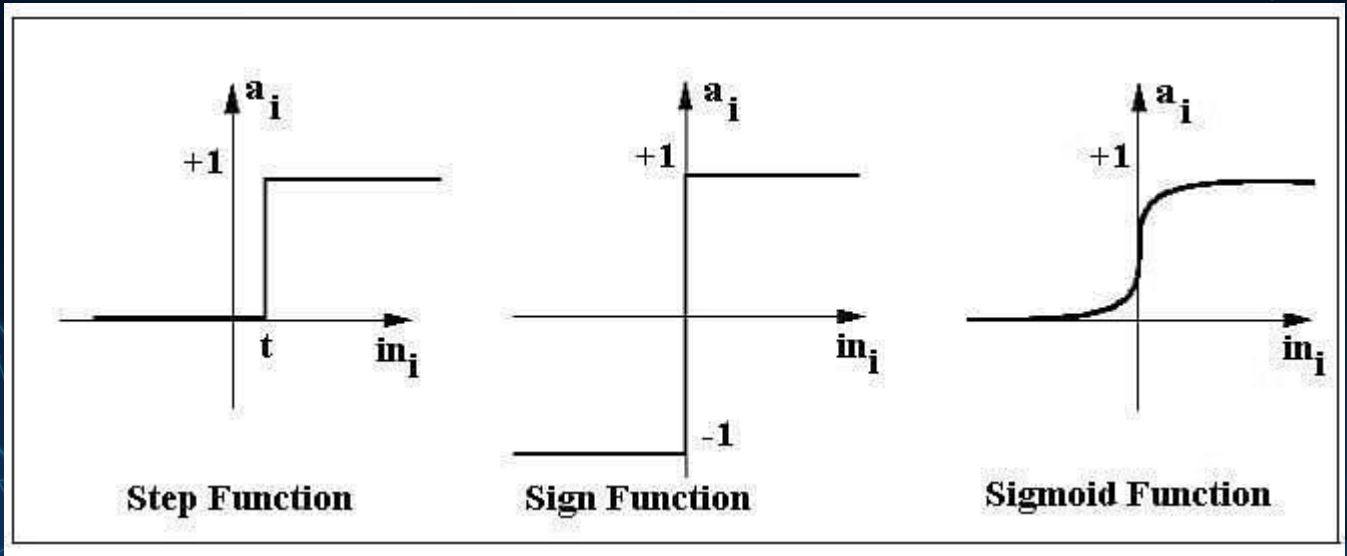


A Boolean output is based on inputs such as salaried, married, age, past credit profile, etc. It has only two values: Yes and No or True and False. The summation function "∑" multiplies all inputs of "x" by weights "w" and then adds them up as follows:

$$w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

Activation Functions of Perceptron

The activation function applies a step rule (convert the numerical output into +1 or -1) to check if the output of the weighting function is greater than zero or not.



Activation Functions of Perceptron

For example:

If ∑ wixi> 0 => then final output “o” = 1 (issue bank loan)

Else, final output “o” = -1 (deny bank loan)

Step function gets triggered above a certain value of the neuron output; else it outputs zero. Sign Function outputs +1 or -1 depending on whether neuron output is greater than zero or not. Sigmoid is the S-curve and outputs a value between 0 and 1.

Output of Perceptron

Perceptron with a Boolean output:

Inputs: x1...xn
Output: o(x1....xn)

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

Weights: wi=> contribution of input xi to the Perceptron output;
w0=> bias or threshold

Output of Perceptron

If ∑w.x > 0, output is +1, else -1. The neuron gets triggered only when weighted input reaches a certain threshold value.

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

An output of +1 specifies that the neuron is triggered. An output of -1 specifies that the neuron did not get triggered.

“sgn” stands for sign function with output +1 or -1

Error in Perceptron

In the Perceptron Learning Rule, the predicted output is compared with the known output. If it does not match, the error is propagated backward to allow weight adjustment to happen.

Perceptron: Decision Function

A decision function φ(z) of Perceptron is defined to take a linear combination of x and w vectors.

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Perceptron: Decision Function

The value z in the decision function is given by:

$$z = w_1 x_1 + \ldots + w_m x_m$$

The decision function is +1 if z is greater than a threshold θ, and it is -1 otherwise.

$$\phi(z) = \begin{cases} 1 & if\ z \geq \theta \\ -1 & otherwise \end{cases}$$

This is the Perceptron algorithm.

Bias Unit

For simplicity, the threshold θ can be brought to the left and represented as w0x0, where w0= -θ and x0= 1.

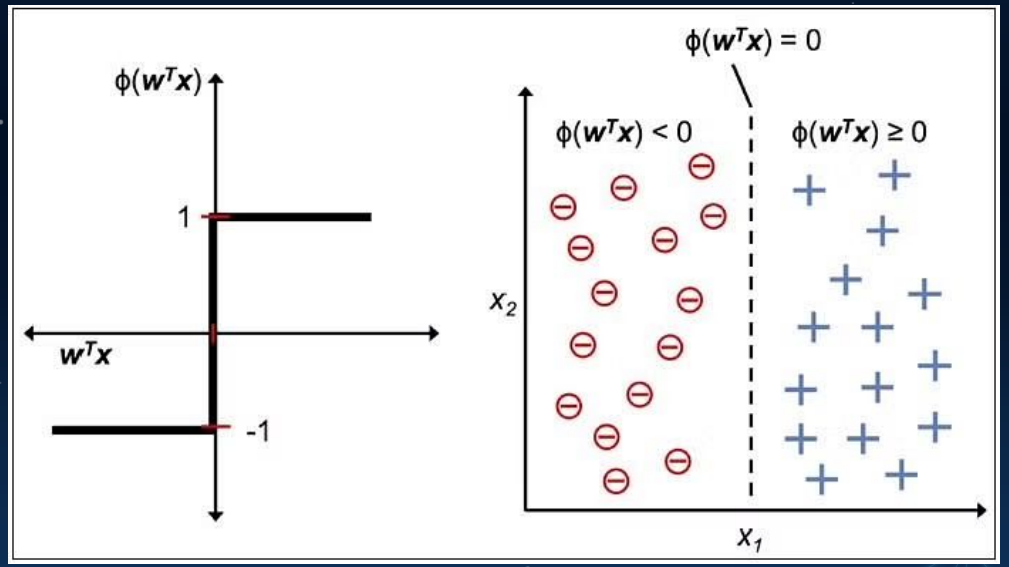
$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \boldsymbol{w}^T \boldsymbol{x}$$

The value w0 is called the bias unit.

The decision function then becomes:

$$\phi(z) = \begin{cases} 1 & if\ z \geq 0 \\ -1 & otherwise \end{cases}$$

Output:
The figure shows how the decision function squashes wTx to either +1 or -1 and how it can be used to discriminate between two linearly separable classes.



Summary

- Perceptron is an algorithm for Supervised Learning of single layer binary linear classifiers.
- Optimal weight coefficients are automatically learned.
- Weights are multiplied with the input features and decision is made if the neuron is fired or not.
- Activation function applies a step rule to check if the output of the weighting function is greater than zero.
- Linear decision boundary is drawn enabling the distinction between the two linearly separable classes +1 and -1.
- If the sum of the input signals exceeds a certain threshold, it outputs a signal; otherwise, there is no output.

Limitation of Perceptron Model

The following are the limitation of a Perceptron model:

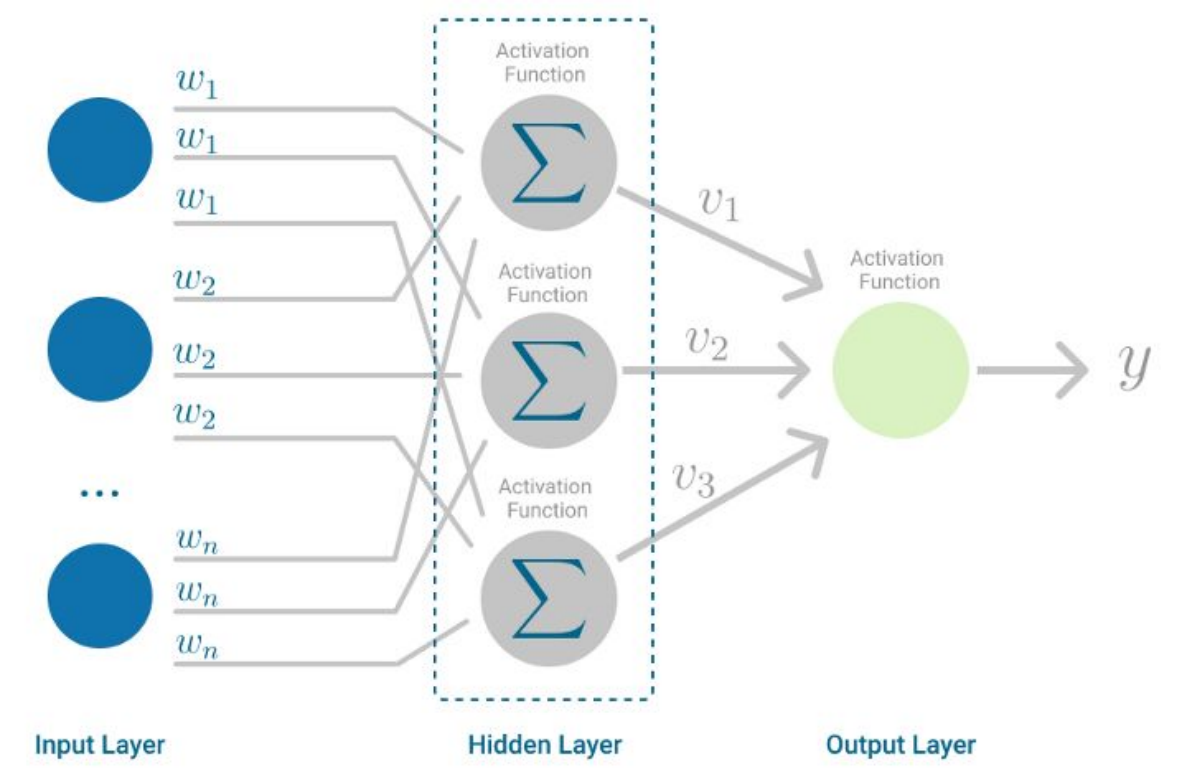
1. The output of a perceptron can only be a binary number (0 or 1) due to the hard-edge transfer function.
2. It can only be used to classify the linearly separable sets of input vectors. If the input vectors are non-linear, it is not easy to classify them correctly.
3. Although it was said the Perceptron could represent any circuit and logic, the biggest criticism was that it couldn't represent the XOR gate, exclusive OR, where the gate only returns 1 if the inputs are different.

Multi Layer Perceptron

The Multilayer Perceptron was developed to tackle the above explained limitation. It is a neural network where the mapping between inputs and output is **non-linear.**

A Multilayer Perceptron has input and output layers, and one or more hidden layers with many neurons stacked together. And while in the Perceptron the neuron must have an activation function that imposes a threshold, like ReLU or sigmoid, neurons in a Multilayer Perceptron can use any arbitrary activation function

Diagram


Activation Function
Activation Function
Activation Function
Activation Function
w_1
w_1
w_1
w_2
w_2
w_2
w_n
w_n
w_n
v_1
v_2
v_3
y
Input Layer
Hidden Layer
Output Layer

Multi Layer Perceptron

Multilayer Perceptron falls under the category of feedforward algorithms, because inputs are combined with the initial weights in a weighted sum and subjected to the activation function, just like in the Perceptron. But the difference is that each linear combination is propagated to the next layer.

Each layer is feeding the next one with the result of their computation, their internal representation of the data. This goes all the way through the hidden layers to the output layer.

Multi Layer Perceptron

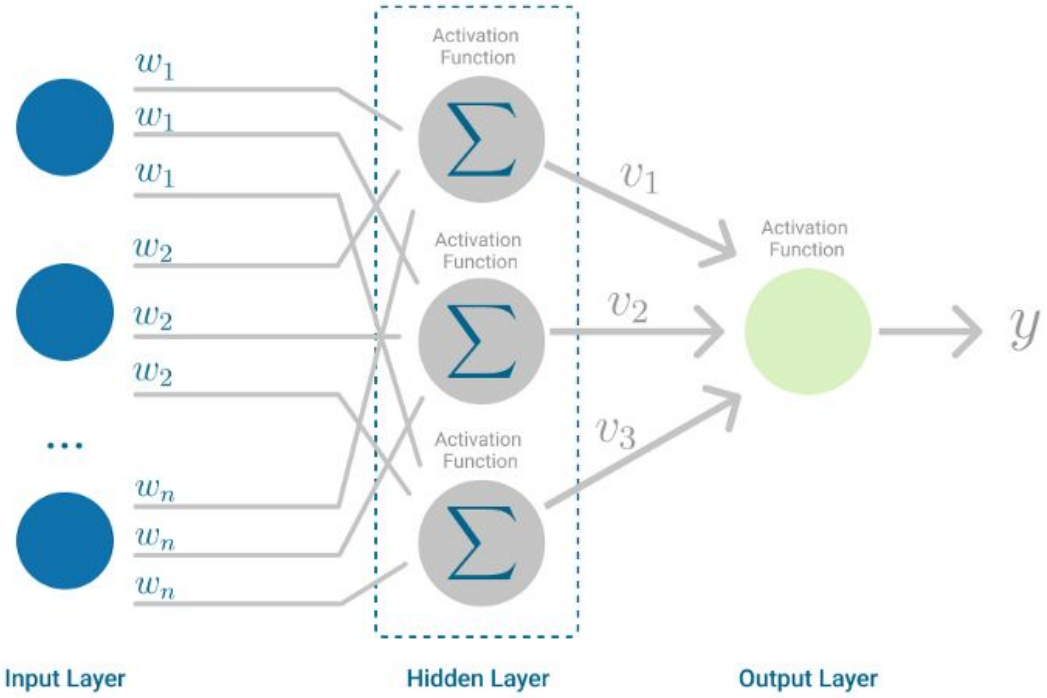
But it has more to it.

If the algorithm only computed the weighted sums in each neuron, propagated results to the output layer, and stopped there, it wouldn't be able to learn the weights that minimize the cost function. If the algorithm only computed one iteration, there would be no actual learning.
This is where Backpropagation comes into play.

Multi Layer Perceptron

Backpropagation

Backpropagation is the learning mechanism that allows the Multilayer Perceptron to iteratively adjust the weights in the network, with the goal of minimizing the cost function.
There is one hard requirement for backpropagation to work properly. The function that combines inputs and weights in a neuron, for instance the weighted sum, and the threshold function, for instance ReLU, must be differentiable. These functions must have a bounded derivative, because Gradient Descent is typically the optimization function used in MultiLayer Perceptron.

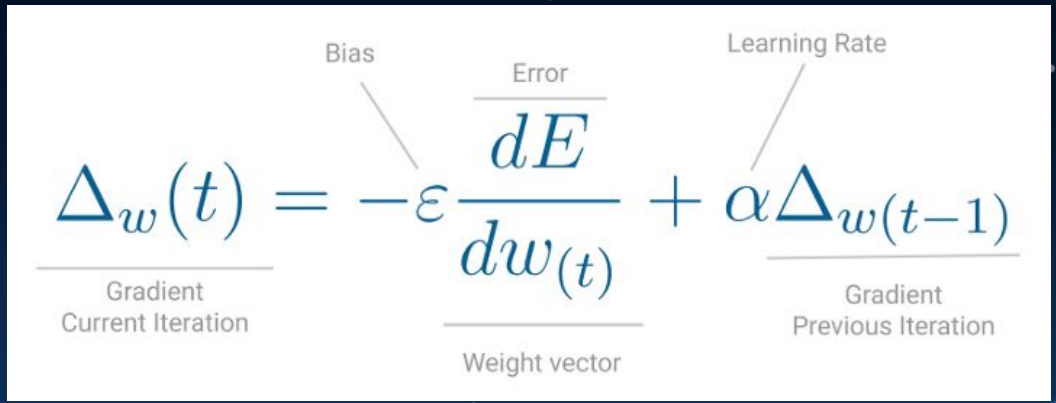


1. Feedforward | Mean Squared Error (MSE) computed

2. Backpropagation | Gradient is computed

Multi Layer Perceptron

In each iteration, after the weighted sums are forwarded through all layers, the gradient of the Mean Squared Error is computed across all input and output pairs. Then, to propagate it back, the weights of the first hidden layer are updated with the value of the gradient. That's how the weights are propagated back to the starting point of the neural network!



One iteration of Gradient Descent

Multi Layer Perceptron

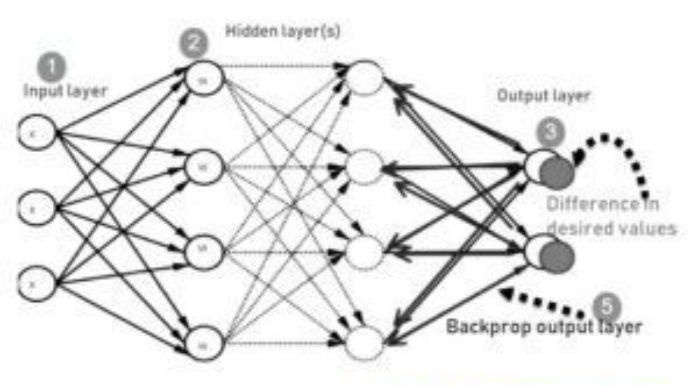
This process keeps going until gradient for each input-output pair has converged, meaning the newly computed gradient hasn’t changed more than a specified convergence threshold, compared to the previous iteration.

How backpropagation Works - Simple Algorithm

The backpropagation works on 4 layers. They are the input layer, hidden layer, hidden layer 2, and final output layer. Hence, it has 3 main layers. They are

- Input layer
- Hidden layer and
- Output layer

Each layer works independently in its way to get the desired output and the scenarios correspond to our conditions.



How backpropagation Works - Simple Algorithm

- The input layer receives the inputs X through the preconnected path
- Input is customized by using actual weights ‘W’, where the weights are selected randomly.
- Output is calculated for every neuron from the input layer, at the hidden layer and the output data has arrived at the output layer
- Evaluate the errors obtained from the outputs.
- To decrease the error, adjust the weights by going back to the hidden layer from the output layer.
- Repeat the process until the desired output is obtained.

The difference between the actual output and the desired output is used to calculate errors obtained in the result.

Error = actual output – desired output.

Why We Need Backpropagation?

The backpropagation technology helps to adjust the weights of the network connections to minimize the difference between the actual output and the desired output of the net, which is calculated as a loss function.

- Helps to simplify the network structure by removing the weighted links, so that the trained network will have the minimum effect
- This method is especially applicable in deep neural networks, which work on error-prone projects like speech and image recognition.
- It functions with multiple inputs using chain rules and power rules.
- It is used to calculate the gradient of the loss function with respect to all the weights in the network.
- Minimizes the loss function by updating the weights with the gradient optimization method.
- Modifies the weights of the connected nodes during the process of training to produce ‘learning’.
- This method is iterative, recursive, and more efficient.

Advantage of Backpropagation

- It is very fast, simple, and easy to analyze and program
- Apart from no of inputs, it doesn’t contain any parameters for tuning
- This method is flexible and there is no need to acquire more knowledge about the network.
- This is a standardized method and works very efficiently.
- It doesn’t require any special features of the function.

Disadvantage of Backpropagation

- The function or performance of the backpropagation network on a certain issue depends on the data input.
- These types of networks are very sensitive to noisy data.
- The matrix-based approach is used instead of a mini-batch.

Loss Function

Loss functions are used to determine the error (aka “the loss”) between the output of our algorithms and the given target value. In layman’s terms, the loss function expresses how far off the mark our computed output is.

At its core, a loss function is a measure of how good your prediction model does in terms of being able to predict the expected outcome(or value). We convert the learning problem into an optimization problem, define a loss function and then optimize the algorithm to minimize the loss function.

Loss Function

Example -
Assume you are given a task to fill a bag with 10 Kg of sand. You fill it up till the measuring machine gives you a perfect reading of 10 Kg or you take out the sand if the reading exceeds 10kg.
Just like that weighing machine, if your predictions are off, your loss function will output a higher number. If they're pretty good, it'll output a lower number. As you experiment with your algorithm to try and improve your model, your loss function will tell you if you're getting(or reaching) anywhere.

"The function we want to minimize or maximize is called the objective function or criterion. When we are minimizing it, **we may also call it the cost function, loss function, or error function**".

WHAT ARE LOSS FUNCTIONS IN MACHINE LEARNING?

The loss function is a method of evaluating how well your machine learning algorithm models your featured data set. In other words, loss functions are a measurement of how good your model is in terms of predicting the expected outcome

The cost function and loss function refer to the same context . We calculate the cost function as the average of all loss function values whereas we calculate the loss function for each sample output compared to its actual value.

The loss function is directly related to the predictions of the model you've built. If your loss function value is low, your model will provide good results. The loss function (or rather, the cost function) you use to evaluate the model performance needs to be minimized to improve its performance.

Types of Loss Function

Broadly speaking, loss functions can be grouped into two major categories concerning the types of problems we come across in the real world: **classification and regression**.

In classification problems, our task is to predict the respective probabilities of all classes the problem is dealing with.

On the other hand, when it comes to regression, our task is to predict the continuous value concerning a given set of independent features to the learning algorithm.

Loss function

ASSUMPTIONS

- n/m — number of training samples
- i — ith training sample in a data set
- y(i) — Actual value for the ith training sample
- y_hat(i) — Predicted value for the ith training sample

Cost Function

TYPES OF CLASSIFICATION LOSSES

1. Binary Cross-Entropy Loss / Log Loss
2. Hinge Loss

1. Binary Cross-Entropy Loss / Log Loss

1. BINARY CROSS-ENTROPY LOSS / LOG LOSS

This is the most common loss function used in classification problems. The cross-entropy loss decreases as the predicted probability converges to the actual label. It measures the performance of a classification model whose predicted output is a probability value between 0 and 1.
When the number of classes is 2, it's binary classification.

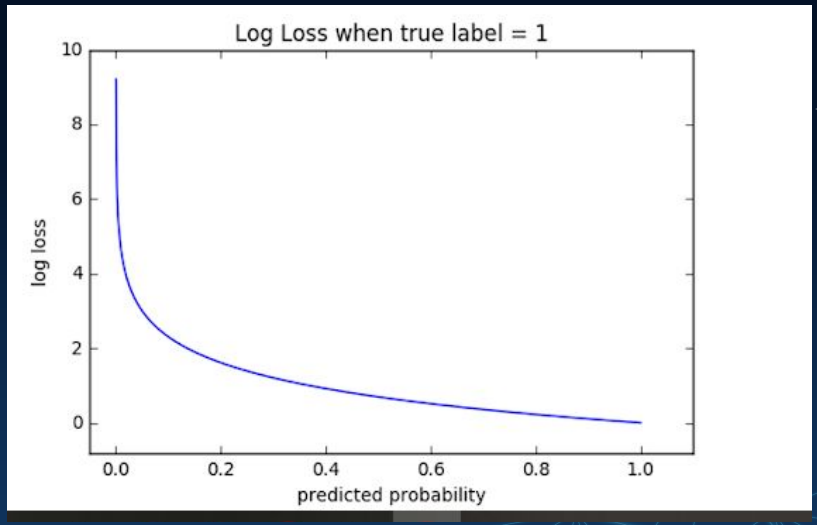
$$L = -\frac{1}{m}\sum_{i=1}^{m}(y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i))$$

1. Binary Cross-Entropy Loss / Log Loss

When the number of classes is more than 2, it's multi-class classification.

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i)$$

We derive the cross-entropy loss formula from the regular likelihood function, but with logarithms added in.

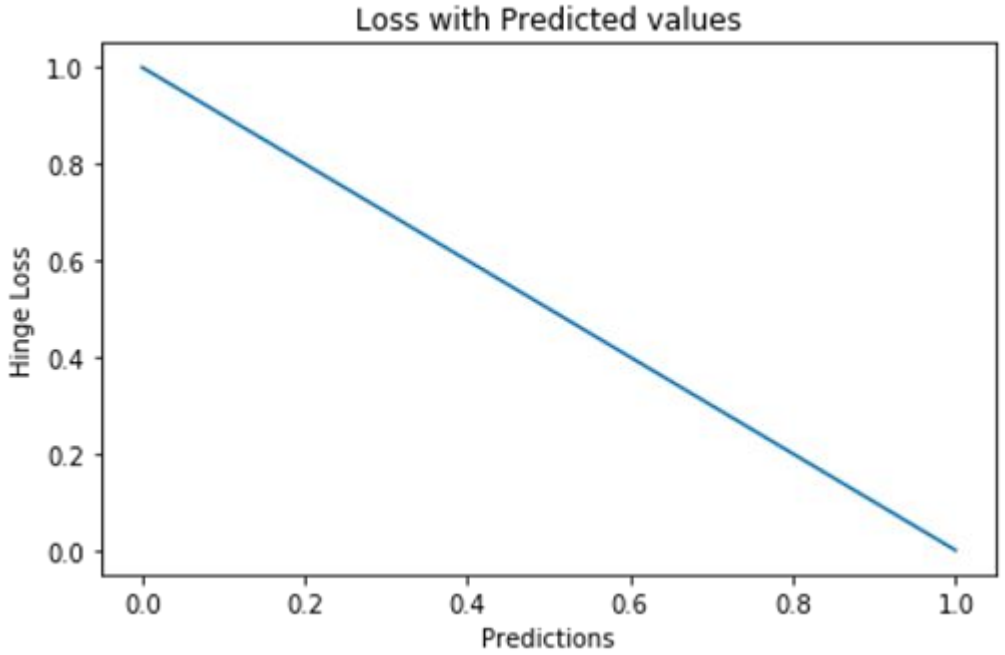


2. Hinge function

Also known as Multi class SVM Loss

In simple terms, the score of correct category should be greater than sum of scores of all incorrect categories by some safety margin (usually one). And hence hinge loss is used for maximum-margin classification, most notably for support vector machines. Although not differentiable, it's a convex function which makes it easy to work with usual convex optimizers used in machine learning domain.

$$L = \max(0, 1 - y * f(x))$$



Hinge loss penalizes the wrong predictions and the right predictions that are not confident. It's primarily used with SVM classifiers with class labels as -1 and 1. Make sure you change your malignant class labels from 0 to -1.

Cost Function

TYPES OF REGRESSION LOSSES

1. Mean Square Error / Quadratic Loss / L2 Loss
2. Mean Absolute Error / L1 Loss
3. Huber Loss / Smooth Mean Absolute Error
4. Log-Cosh Loss
5. Quantile Loss

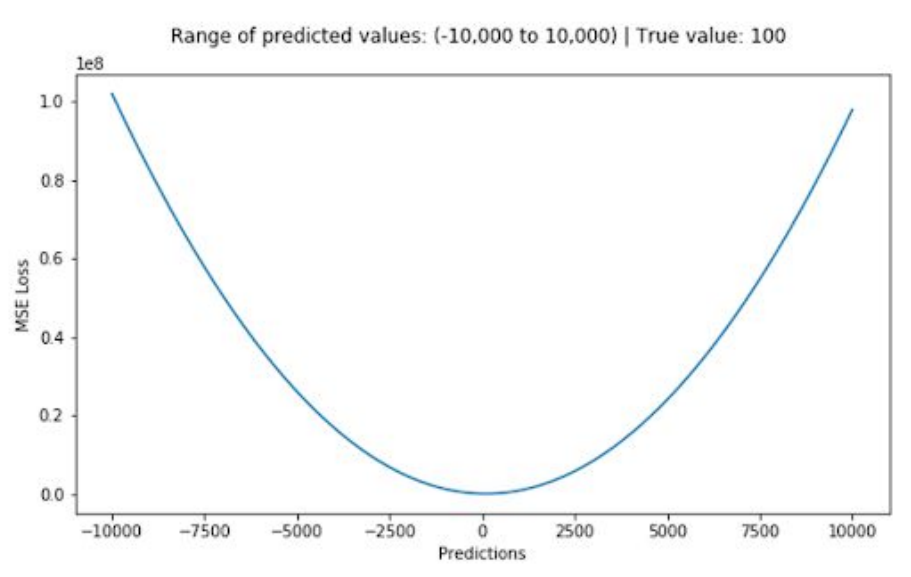
Cost function

Mean Square Error / Quadratic Loss / L2 Loss

We define MSE loss function as the average of squared differences between the actual and the predicted value. It's the most commonly used regression loss function.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

Cost function



The corresponding cost function is the mean of these squared errors (MSE). The MSE loss function penalizes the model for making large errors by squaring them and this property makes the MSE cost function less robust to outliers. Therefore, you shouldn't use it if the data is prone to many outliers.

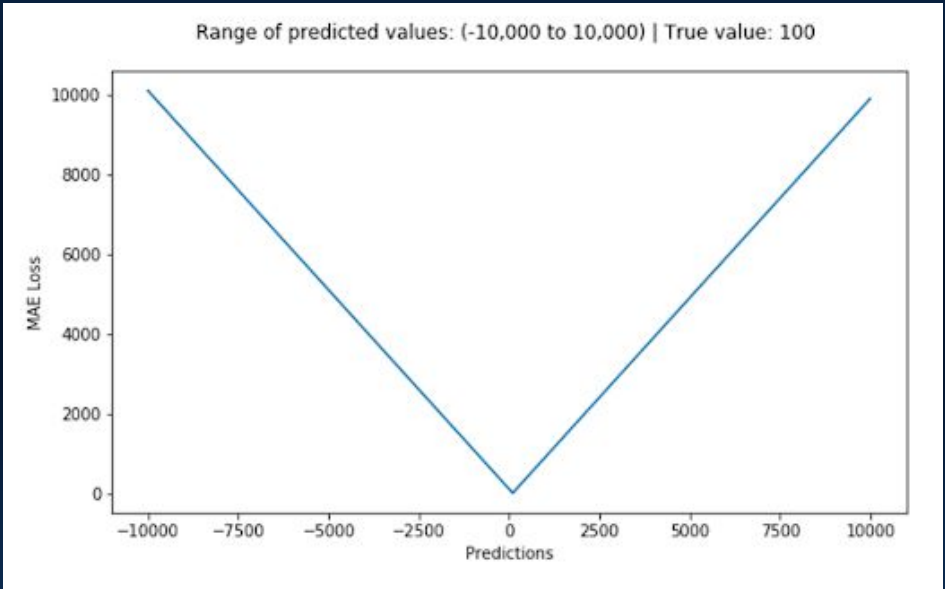
Cost function

MEAN ABSOLUTE ERROR / L1 LOSS

We define MAE loss function as the average of absolute differences between the actual and the predicted value. It's the second most commonly used regression loss function. It measures the average magnitude of errors in a set of predictions, without considering their directions.

$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

Cost function



The corresponding cost function is the mean of these absolute errors (MAE). The MAE loss function is more robust to outliers compared to the MSE loss function. Therefore, you should use it if the data is prone to many outliers.

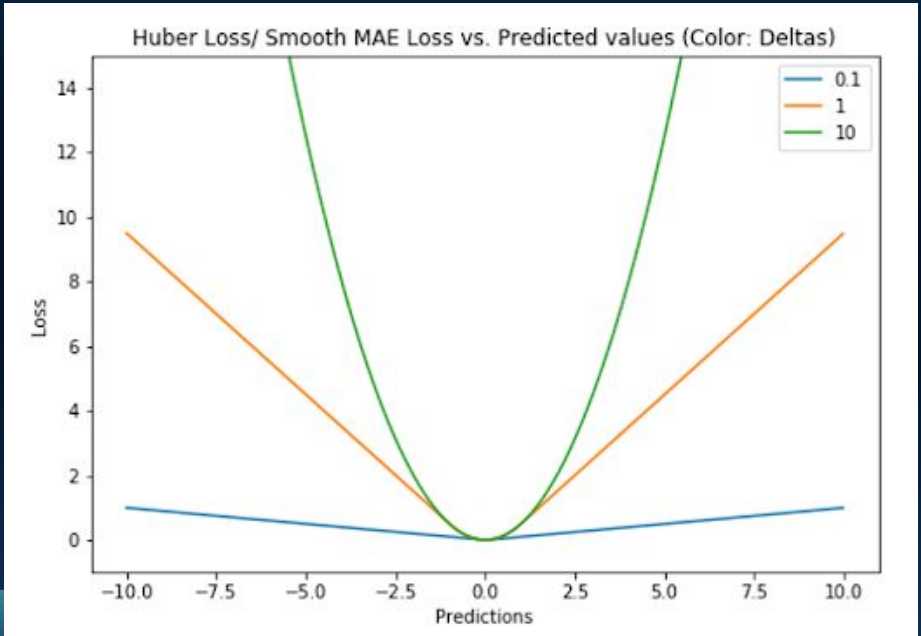
Cost function

HUBER LOSS / SMOOTH MEAN ABSOLUTE ERROR

The Huber loss function is defined as the combination of MSE and MAE loss functions because it approaches MSE when $\delta \sim 0$ and MAE when $\delta \sim \infty$ (large numbers). It is mean absolute error, which becomes quadratic when the error is small. To make the error quadratic depends on how small that error could be, which is controlled by a hyperparameter, δ (delta) that you can tune.

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \leq \delta, \\ \delta |y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

Cost function



The choice of the delta value is critical because it determines what you're willing to consider an outlier. Hence, the Huber loss function could be less sensitive to outliers than the MSE loss function, depending on the hyperparameter value. Therefore, you can use the Huber loss function if the data is prone to outliers. In addition, we might need to train hyperparameter delta, which is an iterative process.

Cost function

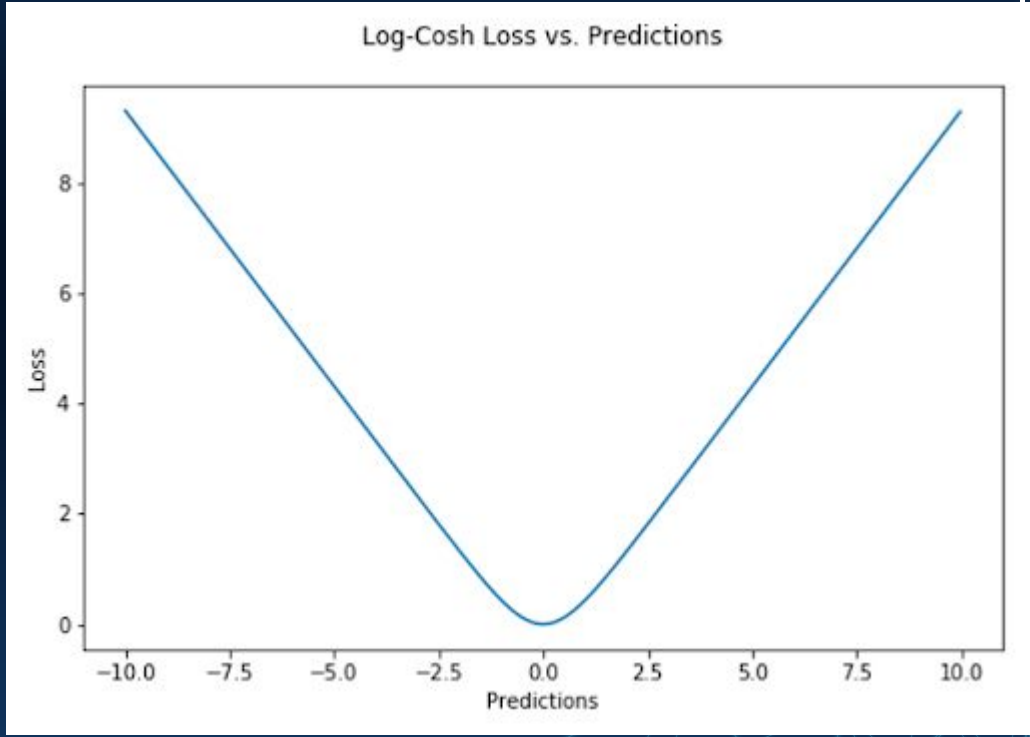
LOG-COSH LOSS

The log-cosh loss function is defined as the logarithm of the hyperbolic cosine of the prediction error. It's another function used in regression tasks that's much smoother than MSE loss. It has all the advantages of Huber loss because it's twice differentiable everywhere, unlike Huber loss, because some learning algorithms like XGBoost use Newton's method to find the optimum, and hence the second derivative (Hessian).

$$L(y, y^p) = \sum_{i=1}^{n} \log(\cosh(y_i^p - y_i))$$

Cost function

“Log(cosh(x)) is approximately equal to (x ** 2) / 2 for small x and to abs(x) - log(2) for large x. This means that ‘logcosh’ works mostly like the mean squared error, but will not be so strongly affected by the occasional wildly incorrect prediction.”

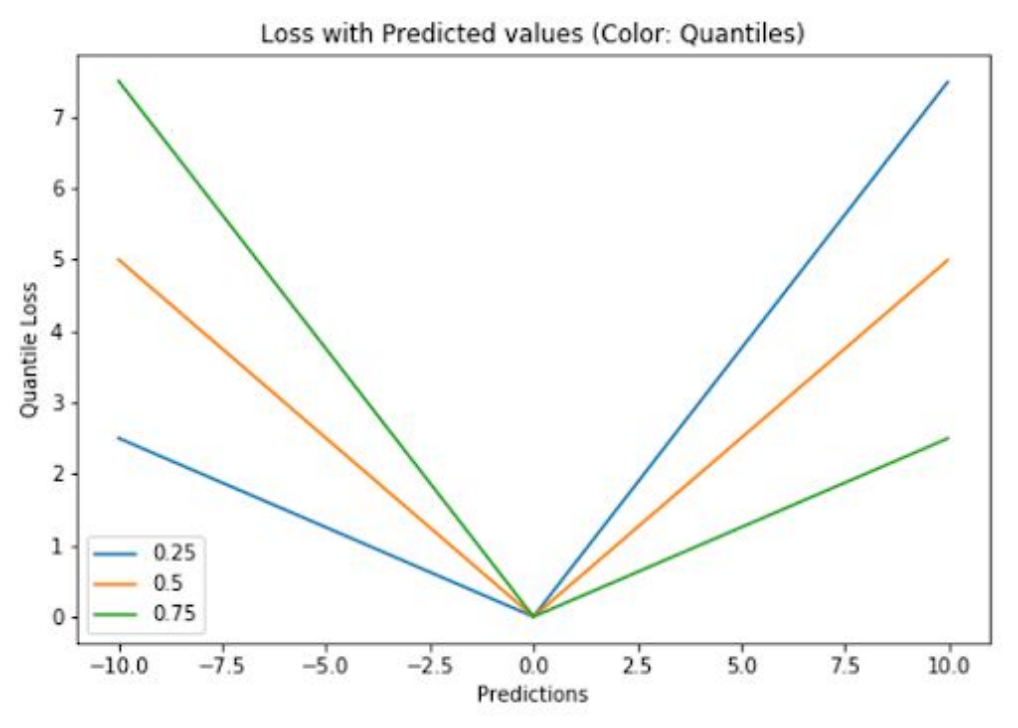


Cost function

QUANTILE LOSS

A quantile is a value below which a fraction of samples in a group falls. Machine learning models work by minimizing (or maximizing) an objective function. As the name suggests, we apply the quantile regression loss function to predict quantiles. For a set of predictions, the loss will be its average.

$$L_\gamma(y, y^p) = \sum_{i=y_i<y_i^p} (\gamma - 1).\,|y_i - y_i^p| + \sum_{i=y_i \geq y_i^p} (\gamma).\,|y_i - y_i^p|$$



Quantile loss function turns out to be useful when we're interested in predicting an interval instead of only point predictions.

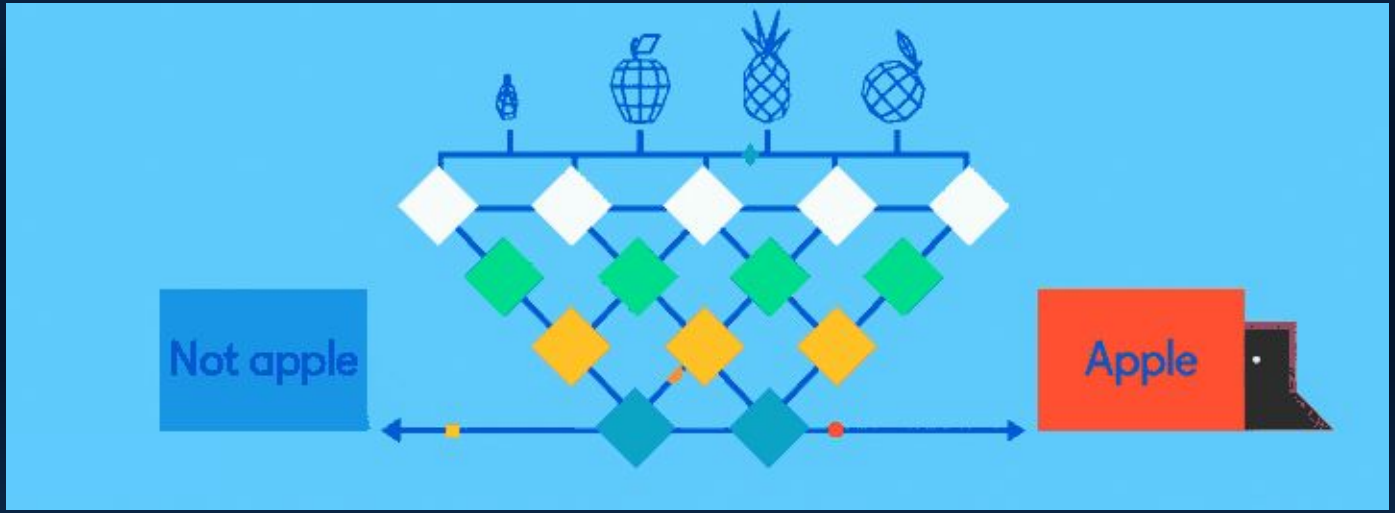
Activation Function

The Internet provides access to plethora of information today. Whatever we need is just a Google (search) away. However, when we have so much information, the challenge is to segregate between relevant and irrelevant information.

When our brain is fed with a lot of information simultaneously, it tries hard to understand and classify the information into “useful” and “not-so-useful” information. We need a similar mechanism for classifying incoming information as “useful” or “less-useful” in case of Neural Networks.

This is important in the way a network learns because not all the information is equally useful. Some of it is just noise. This is where activation functions come into picture. The activation functions help the network use the important information and suppress the irrelevant data points.

Activation Function



Activation Function

Properties that Activation function should hold?

Derivative or Differential: Change in y-axis w.r.t. change in x-axis.It is also known as slope.(Back propagation)

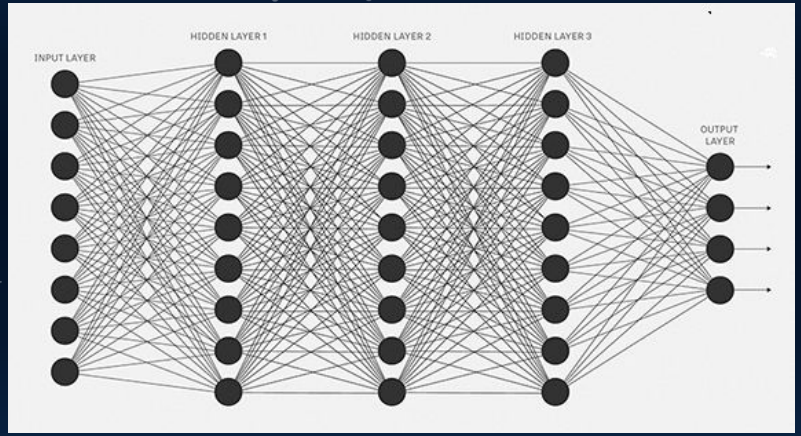
Monotonic function: A function which is either entirely non-increasing or non-decreasing.

Activation Function

An activation function is a very important feature of an artificial neural network , they basically decide whether the neuron should be activated or not.

In artificial neural networks, the activation function defines the output of that node given an input or set of inputs.

An Artificial Neural Network tries to mimic a similar behavior. The network you see below is a neural network made of interconnected neurons. Each neuron is characterized by its weight, bias and activation function.



Activation Function

The input is fed to the input layer, the neurons perform a linear transformation on this input using the weights and biases.

x = (weight * input) + bias

Post that, an activation function is applied on the above result.

$$Y = \text{Activation}(\Sigma(weight * input) + bias)$$

Finally, the output from the activation function moves to the next hidden layer and the same process is repeated. This forward movement of information is known as the forward propagation.

What if the output generated is far away from the actual value? Using the output from the forward propagation, error is calculated. Based on this error value, the weights and biases of the neurons are updated. And that process is known as back-propagation.

Can we do without an activation function?

We understand that using an activation function introduces an additional step at each layer during the forward propagation.

Imagine a neural network without the activation functions. In that case, every neuron will only be performing a linear transformation on the inputs using the weights and biases. Although linear transformations make the neural network simpler, but this network would be less powerful and will not be able to learn the complex patterns from the data.

A neural network without an activation function is essentially just a linear regression model.
Thus we use a non linear transformation to the inputs of the neuron and this non-linearity in the network is introduced by an activation function.

Types OF Activation Function

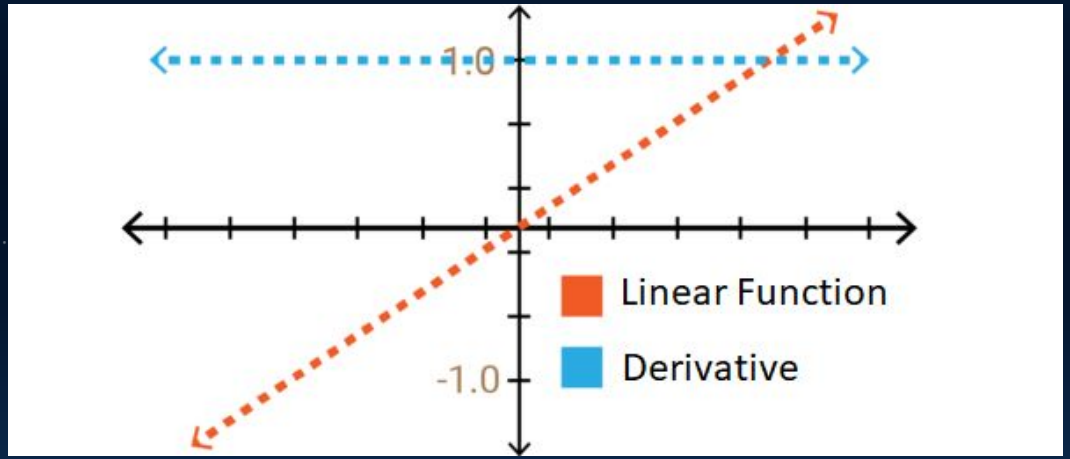
The Activation Functions can be basically divided into 3 types-

1. Linear Activation Function
2. Non-linear Activation Functions
3. Binary Step function

Linear Activation Function

A linear activation function takes the form:
y=mx+c (m is line equation represents W and c is represented as b in neural nets so equation can be modified as y=Wx+b)

It takes the inputs (Xi's), multiplied by the weights(Wi's) for each neuron, and creates an output proportional to the input. In simple term, weighted sum input is proportional to output.



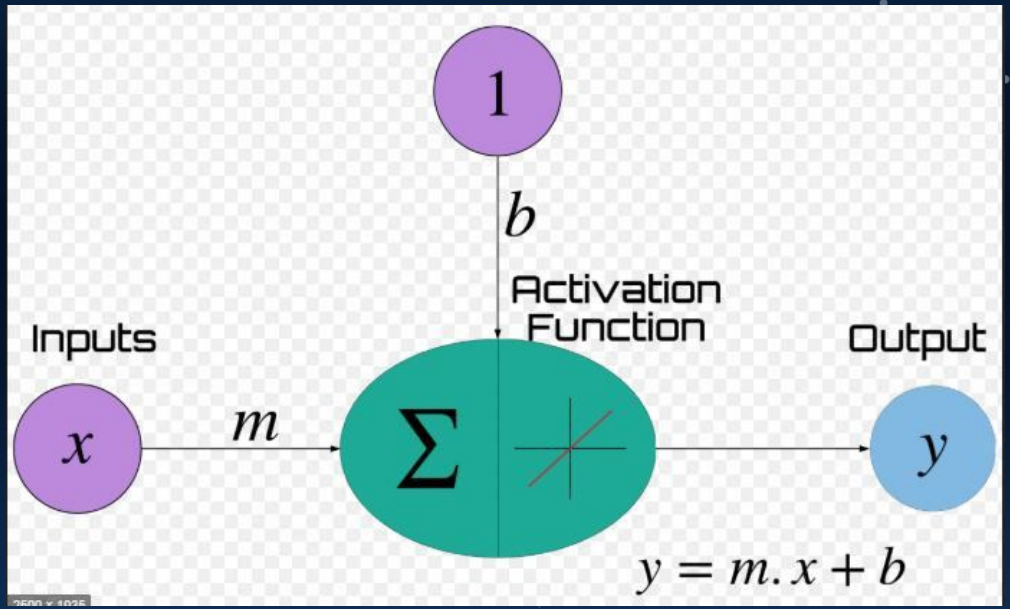
Equation : f(x) = x
Range : (-infinity to infinity)

Linear Activation Function

Problem with Linear function:

1) Differential result is constant.

2) All layers of the neural network collapse into one



Linear Activation Function

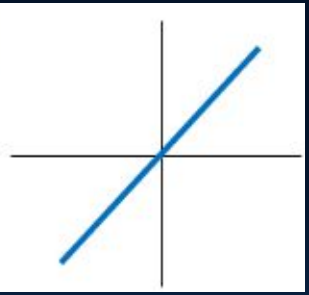
- Differential of linear function is constant and has no relation with the input.Which implies weights and bias will be updated during the backprop but the updating factor (gradient) would be the same.
- linear activation functions, no matter how many layers in the neural network, the last layer will be a linear function of the first layer — Meaning Output of the first layer is same as the output of the nth layer.

A neural networks with a linear activation function is simply a linear regression model.

Linear Activation Function

Pros and Cons : Linear function has limited power and ability to handle complexity.It can be used for simple task like interpretability.

Function	Equation	Range	Derivative
Linear	$f(x) = x$	$-\infty,+\infty$	$f'(x) = 1$



Non Linear Activation Function

Non-Linear function:

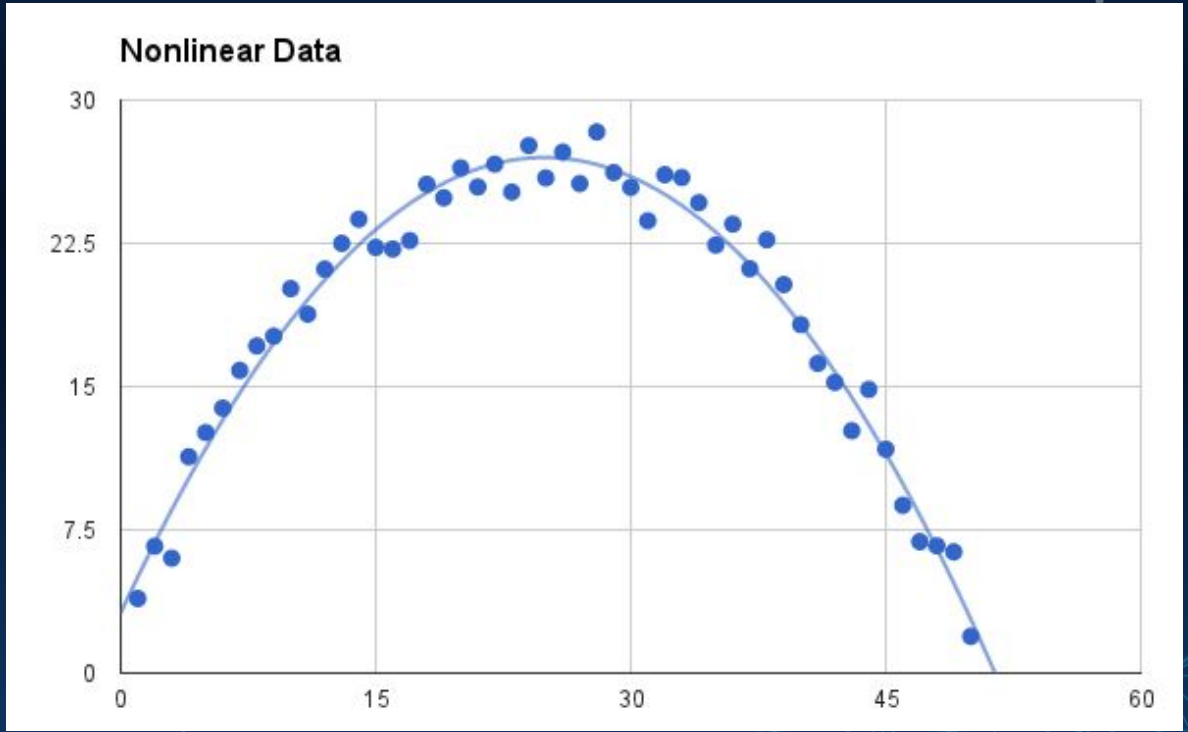
The deep learning rocketing to the sky because of the non-linear functions.Most modern neural network use the non-linear function as their activation function to fire the neuron. Reason being they allow the model to create complex mappings between the network's inputs and outputs, which are essential for learning and modeling complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality.

Advantage of Non-linear function over the Linear function :

- Differential are possible in all the non -linear function.
- Stacking of network is possible , which helps us in creating the deep neural nets.

Non Linear Activation Function

Nonlinearity helps to makes the graph look something like this.



Non Linear Activation Function

It makes it easy for the model to generalize or adapt with variety of data and to differentiate between the output.

The main terminologies needed to understand for nonlinear functions are:

Derivative or Differential: Change in y-axis w.r.t. change in x-axis.It is also known as slope.

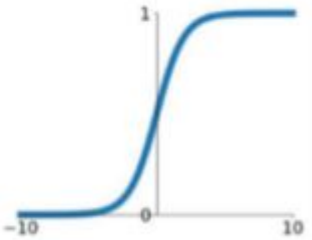
Monotonic function: A function which is either entirely non-increasing or non-decreasing

Non Linear Activation Function

The Nonlinear Activation Functions are mainly divided on the basis of their range or curves

1. Sigmoid or logistic Activation Function:

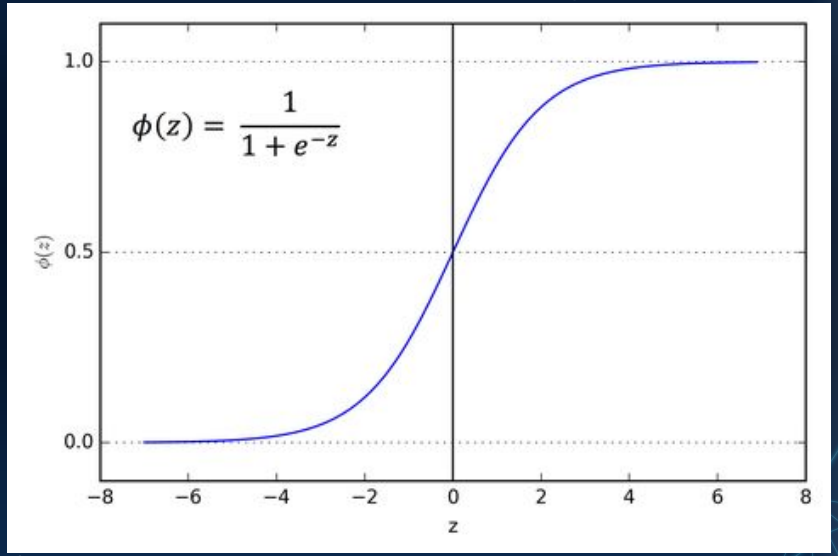
Function	Equation	Range	Derivative
Sigmoid (Logistic)	$f(x) = \frac{1}{1+e^{-x}}$	0,1	$f'(x) = f(x)(1-f(x))$



Non Linear Activation Function

The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.

- The function is differentiable.That means, we can find the slope of the sigmoid curve at any two points.
- The function is monotonic but function's derivative is not.
- The logistic sigmoid function can cause a neural network to get stuck at the training time.
- The softmax function is a more generalized logistic activation function which is used for multiclass classification.



Non Linear Activation Function

Points:

- The output of the sigmoid function always ranges between 0 and 1 .
- Sigmoid is S-shaped , ‘monotonic’ & ‘differential’ function.
- Derivative /Differential of the sigmoid function (f’(x)) will lies between 0 and 0.25.
- Derivative of the sigmoid function is not “monotonic”.

Cons:

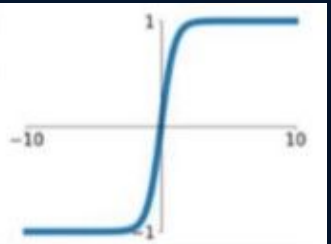
- Derivative of sigmoid function suffers “Vanishing gradient and Exploding gradient problem”.
- Sigmoid function in not “zero-centric”.This makes the gradient updates go too far in different directions. 0 < output < 1, and it makes optimization harder.
- Slow convergence- as its computationally heavy.(Reason use of exponential math function)

“Sigmoid is very popular in classification problems”

Non Linear Activation Function

2. Tanh Activation Function:

Function	Equation	Range	Derivative
Tanh (Hyperbolic tangent)	$f(x) = \frac{2}{1+e^{-2x}} - 1$	-1,1	$f'(x) = 1 - f(x)^2$



Tanh is the modified version of sigmoid function.Hence have similar properties of sigmoid function.

Non Linear Activation Function

The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.

- The function is differentiable.
- The function is monotonic while its derivative is not monotonic.
- The tanh function is mainly used classification between two classes.
- Both tanh and logistic sigmoid activation functions are used in feed-forward nets.

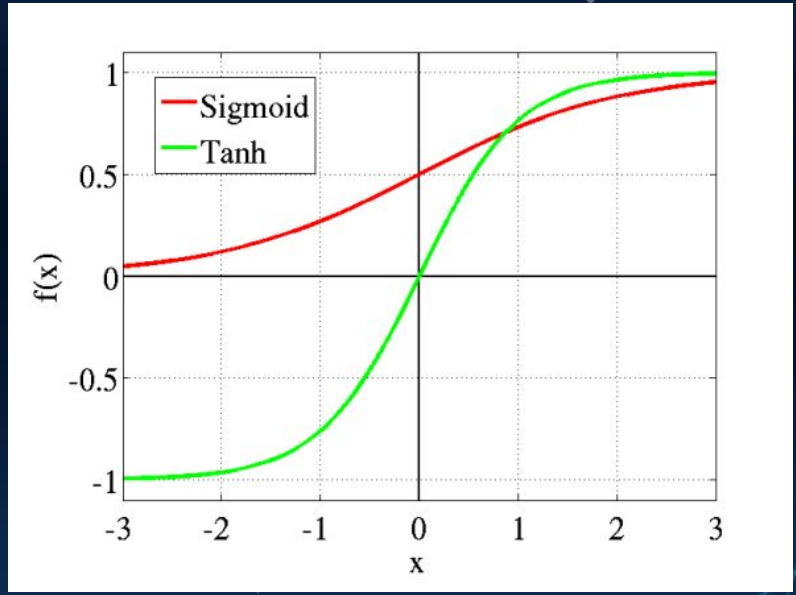


Fig: tanh v/s Logistic Sigmoid

Non Linear Activation Function

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

$$tanh(x) = 2\,sigmoid(2x) - 1$$

Tanh function(left) , Sigmoidal representation of Tanh(right)

“Tanh is preferred over the sigmoid function since it is zero centered and the gradients are not restricted to move in a certain direction”

Non Linear Activation Function

- The function and its derivative both are monotonic
- Output is zero “centric”
- Optimization is easier
- Derivative /Differential of the Tanh function (f’(x)) will lies between 0 and 1.

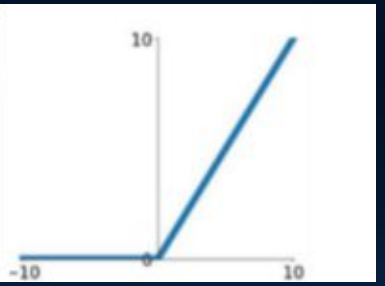
Cons:

- Derivative of Tanh function suffers “Vanishing gradient and Exploding gradient problem”.
- Slow convergence- as its computationally heavy.(Reason use of exponential math function)

Non Linear Activation Function

3. ReLu Activation Function(ReLu — Rectified Linear Units):

Function	Equation	Range	Derivative
ReLu (Rectified Linear Unit)	$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	$0, +\infty$	$f'(x) = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$



The ReLU is the most used activation function in the world right now.Since, it is used in almost all the convolutional neural networks or deep learning.

Non Linear Activation Function

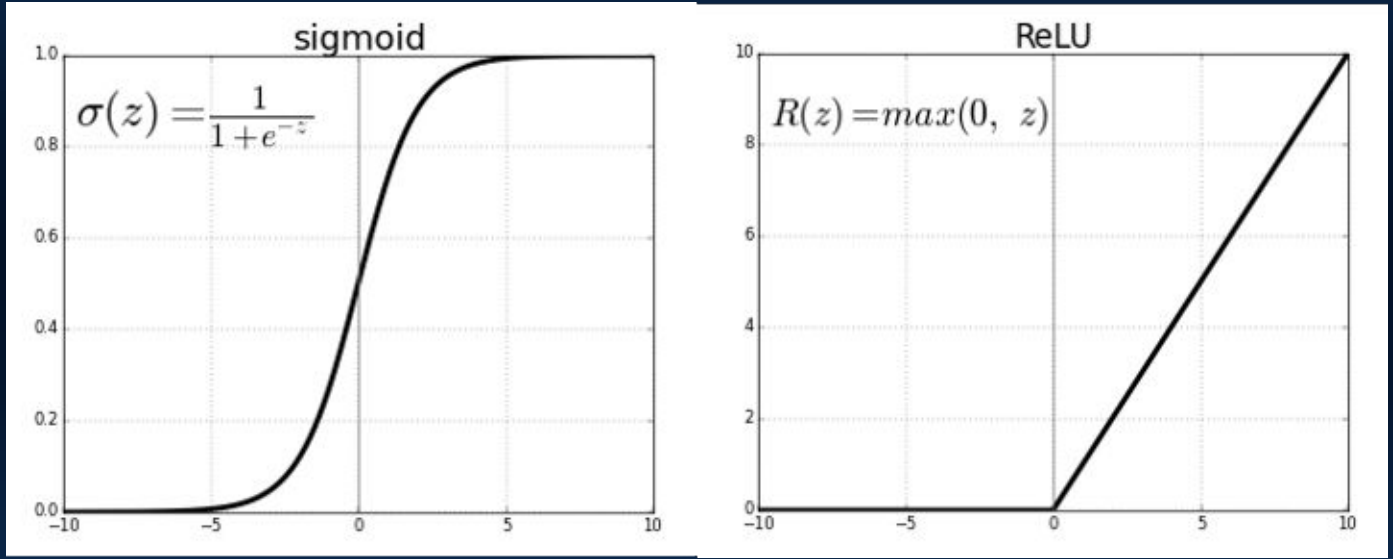


Fig: ReLU v/s Logistic Sigmoid

Non Linear Activation Function

As you can see, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

The function and its derivative both are monotonic.

But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately.

Non Linear Activation Function

Cons:

- ReLu function in not “zero-centric”.This makes the gradient updates go too far in different directions. 0 < output < 1, and it makes optimization harder.
- Dead neuron is the biggest problem.This is due to Non-differentiable at zero.

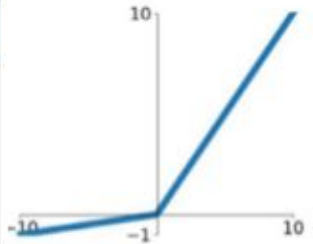
“Problem of Dying neuron/Dead neuron : As the ReLu derivative f’(x) is not 0 for the positive values of the neuron (f’(x)=1 for x ≥ 0), ReLu does not saturate (exploid) and no dead neurons (Vanishing neuron)are reported. Saturation and vanishing gradient only occur for negative values that, given to ReLu, are turned into 0- This is called the problem of dying neuron.”

Non Linear Activation Function

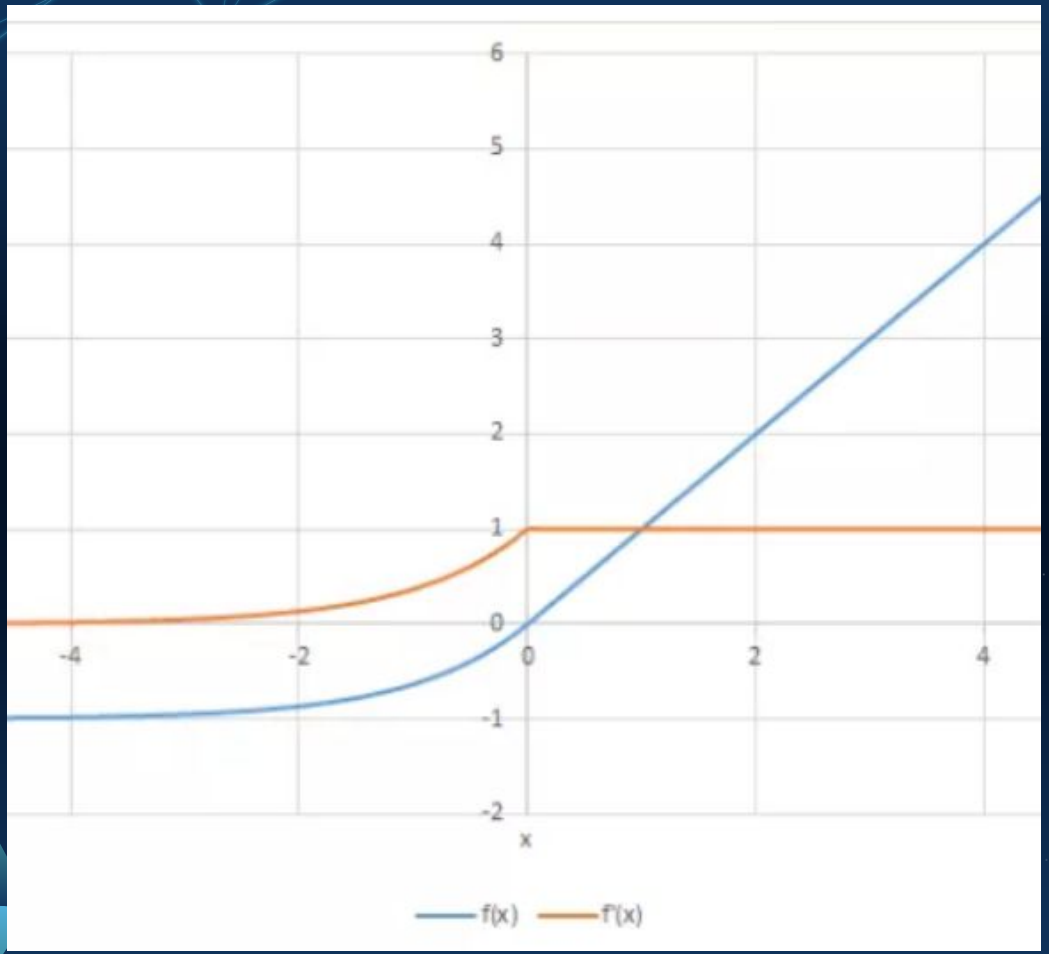
4. Leaky ReLu Activation Function:

Leaky ReLU function is nothing but an improved version of the ReLU function with introduction of “constant slope”

Function	Equation	Range	Derivative
Leaky ReLu	$f(x) = \begin{cases} \alpha x \ for \ x < 0 \\ x \ \ for \ x \geq 0 \end{cases}$	$-\infty, +\infty$	$f'(x) = \begin{cases} \alpha \ for \ x < 0 \\ 1 \ for \ x \geq 0 \end{cases}$



$\alpha = 0.01$



Leaky ReLu activation (blue) , Derivative(organe)

Non Linear Activation Function

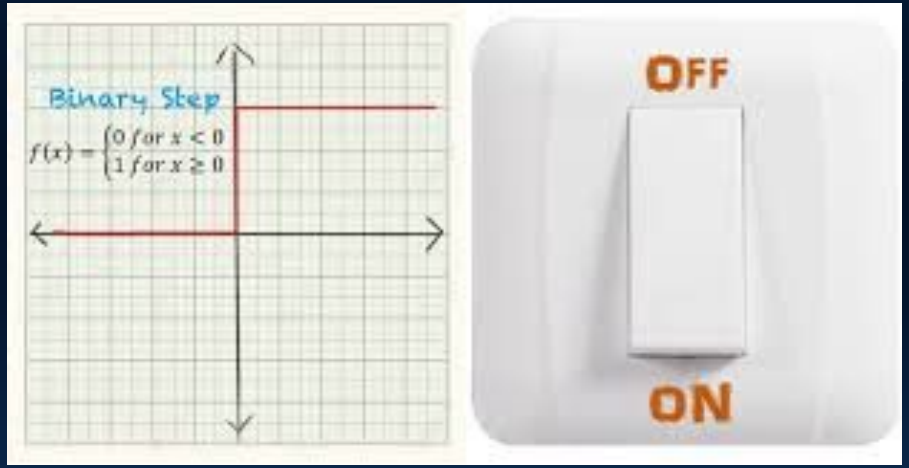
- Leaky ReLU is defined to address problem of dying neuron/dead neuron.
- Problem of dying neuron/dead neuron is addressed by introducing a small slope having the negative values scaled by α enables their corresponding neurons to “stay alive”.
- The function and its derivative both are monotonic
- It allows negative value during back propagation
- It is efficient and easy for computation.
- Derivative of Leaky is 1 when f(x) > 0 and ranges between 0 and 1 when f(x) < 0.

Cons:

- Leaky ReLU does not provide consistent predictions for negative input values.

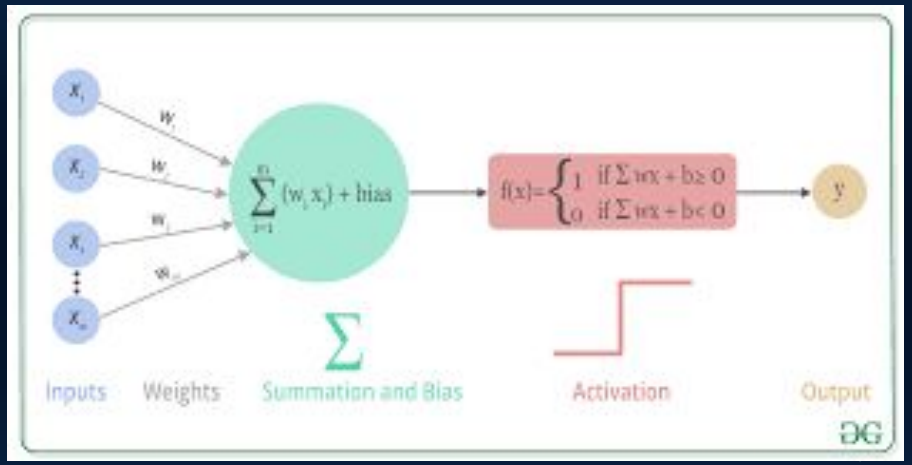
Binary Activation Function

3. Binary Step function

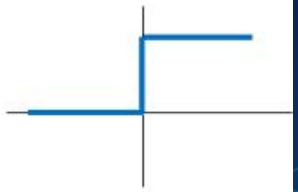


Binary Activation Function

Binary step function are popular known as “ Threshold function”. It is very simple function.



Function	Equation	Range	Derivative
Binary step	$f(x) = \begin{cases} 0 \, for \, x < 0 \\ 1 \, for \, x \geq 0 \end{cases}$	$-\infty, +\infty$	$f'(x) = 0$



Binary Activation Function

Pros and Cons :

- The gradient(differential) of the binary step function is zero,which is the very big problem in back prop for weight updation.
- Another problem with a step function is that it can handle binary class problem alone.(Though with some tweak we can use it for multi-class problem)

Introduction to Tensorflow and Keras

With data taking center stage in most organizational setups, artificial intelligence and machine learning have the potential to run rampant. How do you control them in a way that optimizes the value of data for your business? Deep learning is a necessity. However, organizations also need ways to simplify the management of processes in their deep learning framework.

TensorFlow is one Google framework that works best with all deep learning models. In all actuality, TensorFlow is nothing but a deep neural network that performs based on its surrounding environment. TensorFlow utilizes the concept of positive reinforcement, whereby the machine diverts toward certain (favorable) tasks.

TensorFlow

The framework for TensorFlow takes into account multiple layers of data known as nodes. These nodes come out with the most accurate outcome for every particular action taken by the system.

To simplify the whole process, TensorFlow can take the effort out of machine learning and harness its potential. The framework helps in the creation of high-end applications. Since deep learning models are a type of machine learning, TensorFlow fits perfectly for the task.

Characteristics Of Tensorflow

The characteristics of an optimized deep learning framework include:

- Exceptional performance by the model that meets the expectations of those on the upper-level hierarchy
- Easily comprehensible by your staff
- Processes run parallel to each other, reducing effort and computations
- The ability to compute gradients automatically
- Exceptional portability

TensorFlow

TensorFlow has all these characteristics and is currently being used by some major government entities and private companies. These organizations harness the power in the framework to derive the best results for their own intelligent processes.

NASA, Dropbox, Airbnb, Uber, Airbus, and Snapchat all follow one form of TensorFlow or another for deep learning. TensorFlow's ability lies in the fact that it can be an amazing tool for businesses looking to get the most out of AI and ML.

How Can Tensorflow Help Your Business?

Deep learning is a transformational solution that helps organizations in their data transformation. The neural networks associated with DL can not only solve business problems, but they can also create value for the organization. DL, under the framework of TensorFlow, can be helpful for businesses in many cases. Some general use cases of the framework include:

- Image Recognition
- Video Analysis
- Sound Recognition
- Text-Based Applications
- Object Tagging
- Flaw Detection
- Sentiment Analysis
- Computer Vision
- Anomaly Detection

How Can Tensorflow Help Your Business?

More specific use cases that we can see rolling out in the coming times include:

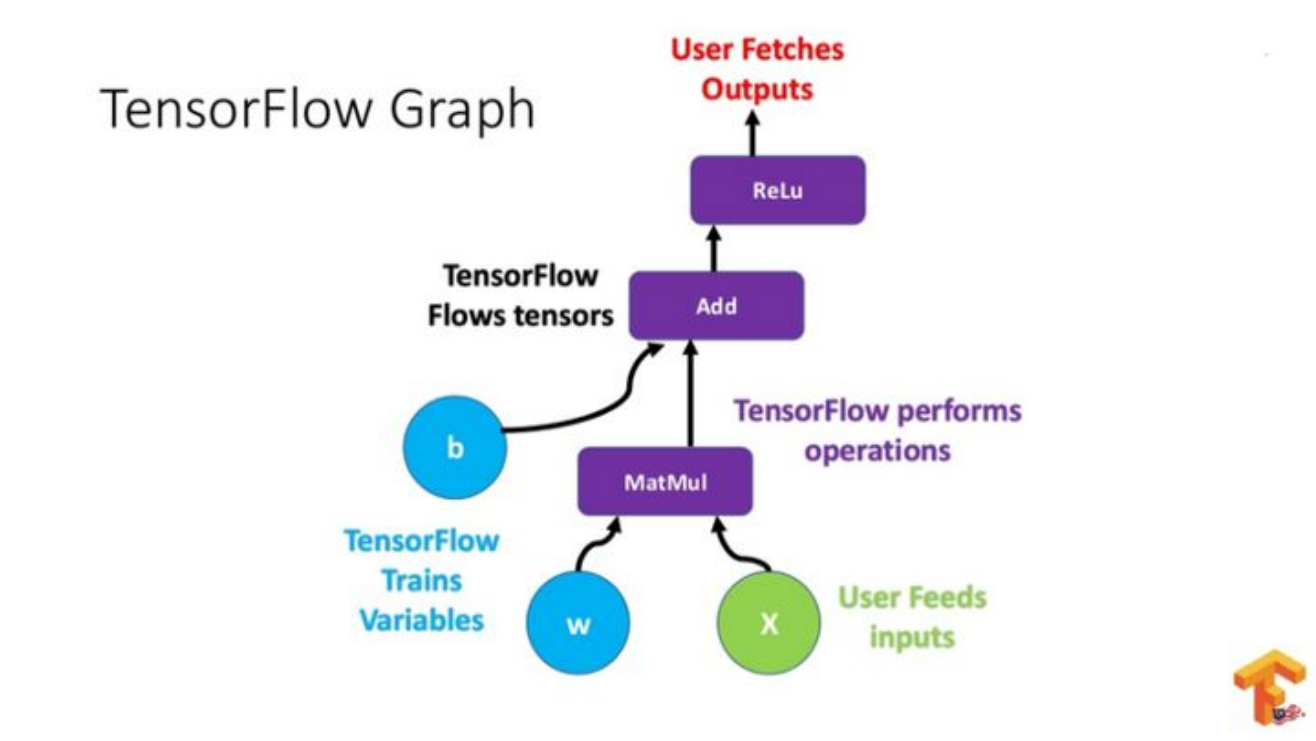
- Self-driving cars
- Sea and air drones
- Smart personal assistants with improved user experience

TensorFlow can prove to help simplify and take the complications out of the algorithms used in these cases.

How Does Tensorflow Work?

- TensorFlow defines computations as Graphs, and these are made with operations (also know as “ops”). So, when we work with TensorFlow, it is the same as defining a series of operations in a Graph.
- To execute these operations as computations, we must launch the Graph into a Session. The session translates and passes the operations represented into the graphs to the device you want to execute them on, be it a GPU or CPU.
- For example, the image below represents a graph in TensorFlow. W, x and b are tensors over the edges of this graph. MatMul is an operation over the tensors W and x, after that Add is called and add the result of the previous operator with b. The resultant tensors of each operation cross the next one until the end where it’s possible to get the wanted result.

How Does Tensorflow Work?



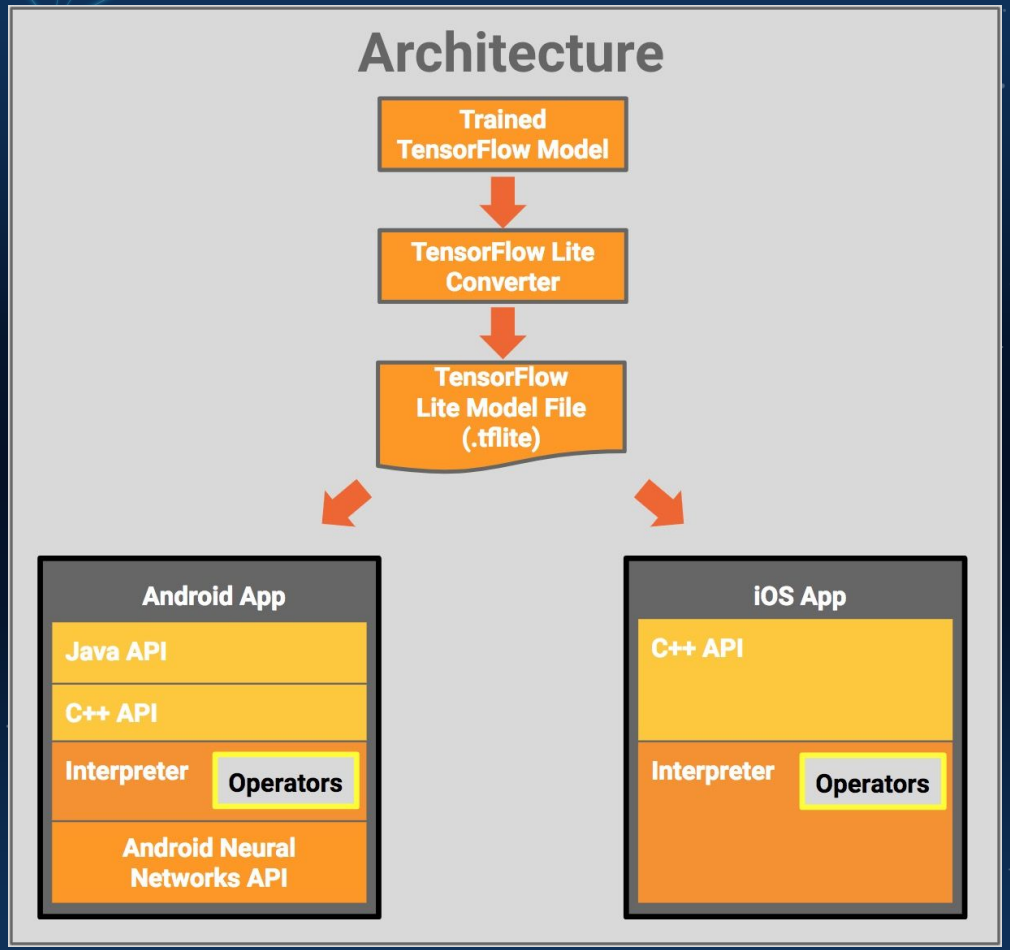
Architecture of TensorFlow

TensorFlow includes transformation libraries written as high-performance C++ binaries, and Python connects the components by routing data between them and providing high-level programming abstractions.

The architecture of TensorFlow is divided into three parts:

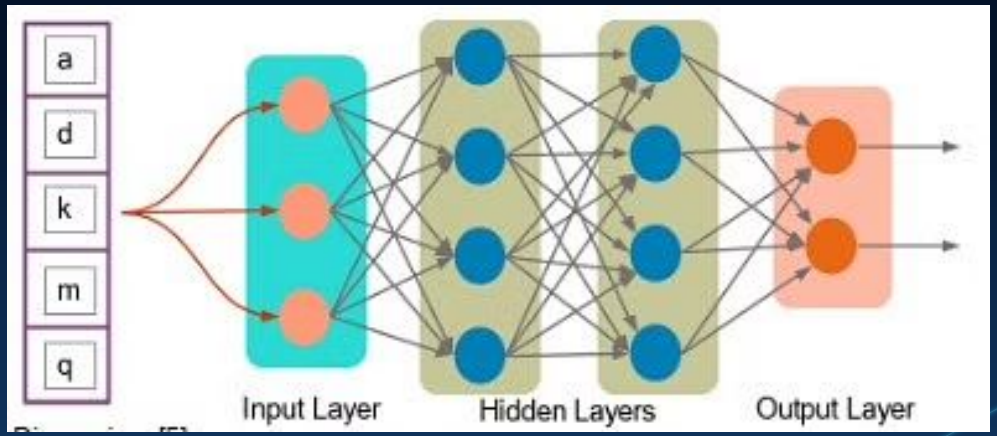
- Preparing the data
- Creating the model
- Prepare the model by training it and estimating it.

Architecture



Mathematical Approaches In Tensorflow

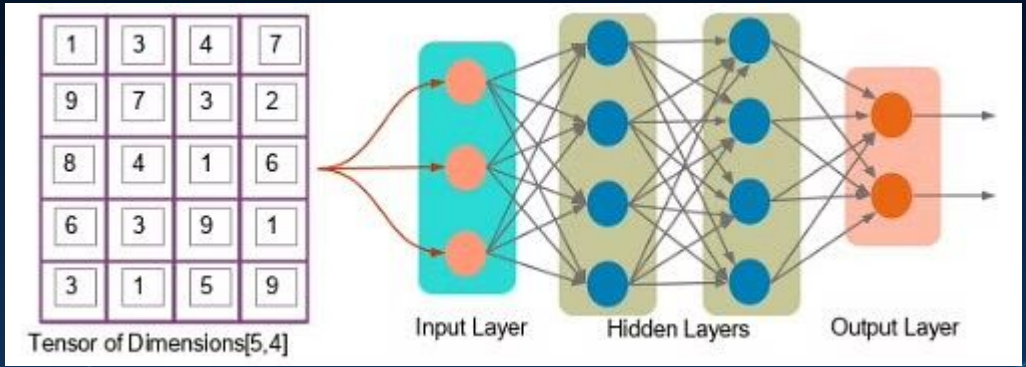
A tensor is a mathematical object represented as arrays of higher dimensions. These arrays of data with different sizes and ranks get fed as input to the neural network. These are the tensors.



Mathematical Approaches In Tensorflow

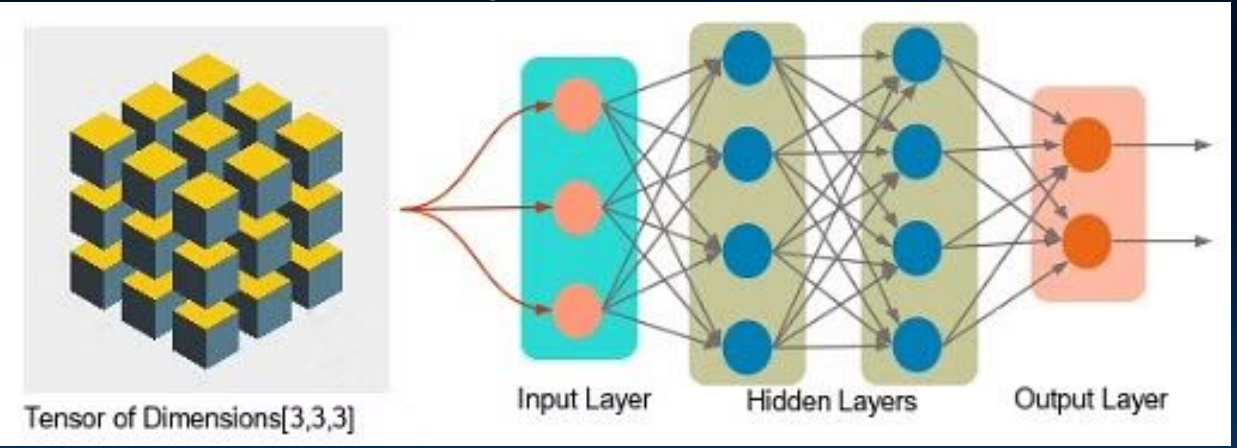
You can have arrays or vectors, which are one-dimensional, or matrices, which are two-dimensional. But tensors can be more than three, four or five-dimensional. Therefore, it helps in keeping the data very tight in one place and then performing all the analysis around that.

Let us look at an example of a tensor of [5,4] dimensions (two-dimensional).



Mathematical Approaches In Tensorflow

Next, you can see a tensor of dimension [3,3,3] (three-dimensional).



Mathematical Approaches In Tensorflow

Tensor Rank

Tensor rank is nothing but the dimension of the tensor. It starts with zero. Zero is a scalar that doesn't have multiple entries in it. It's a single value.

For example, s = 10 is a tensor of rank 0 or a scalar.
V = [10, 11, 12] is a tensor of rank 1 or a vector.
M = [[1, 2, 3],[4, 5, 6]] is a tensor of rank 2 or a matrix.
T = [[[1],[2],[3]],[[4],[5],[6]],[[7],[8],[9]]] is a tensor of rank 3 or a tensor.

Tensor Data Type

In addition to rank and shape, tensors also have a data type. The following is a list of the data type:

Data Type	Python Type	Description
DF_FLOAT	tf.float32	32 bits floating point
DF_DOUBLE	tf.float64	64 bits floating point
DT_INT8	tf.int8	8 bits signed integer
DT_INT16	tf.int16	16 bits signed integer
DT_INT32	tf.int32	32 bits signed integer
DT_INT64	tf.int64	64 bits signed integer
DT_UINT8	tf.unit8	8 bits unsigned integer
DT_STRING	tf.string	Variable length byte arrays
DT_BOOL	tf.bool	Boolean

Keras

Deep learning is a branch of artificial intelligence concerned with solving highly complex problems by emulating the working of the human brain. In deep learning, we use neural networks which use multiple operators placed in nodes to help break down the problem into smaller parts, which are each solved individually. But neural networks can be really hard to implement. This problem is taken care of by Keras, a deep learning framework.

Keras

Definition -

Keras is a high-level, deep learning API developed by Google for implementing neural networks. It is written in Python and is used to make the implementation of neural networks easy. It also supports multiple backend neural network computation.
Keras is relatively easy to learn and work with because it provides a python frontend with a high level of abstraction while having the option of multiple back-ends for computation purposes. This makes Keras slower than other deep learning frameworks, but extremely beginner-friendly.

Keras

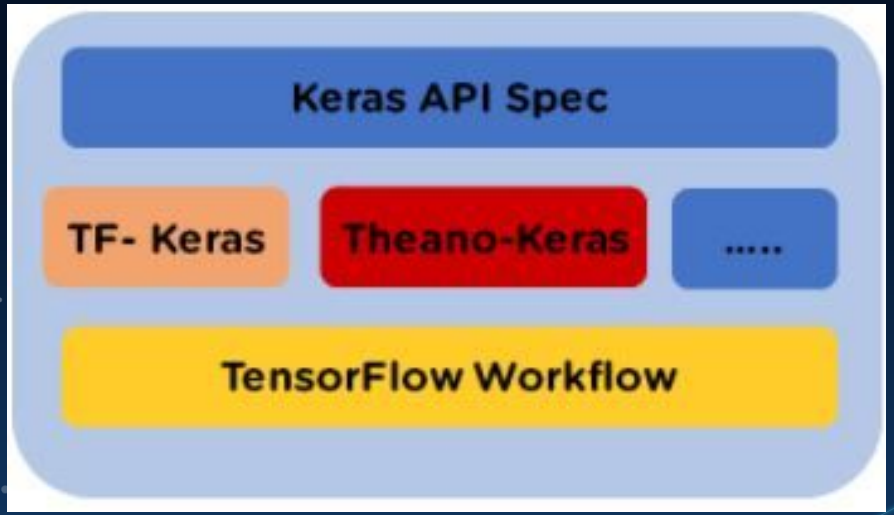
Keras allows you to switch between different back ends. The frameworks supported by Keras are:

- Tensorflow
- Theano
- PlaidML
- MXNet
- CNTK (Microsoft Cognitive Toolkit)

Out of these five frameworks, TensorFlow has adopted Keras as its official high-level API. Keras is embedded in TensorFlow and can be used to perform deep learning fast as it provides inbuilt modules for all neural network computations.

Keras

At the same time, computation involving tensors, computation graphs, sessions, etc can be custom made using the Tensorflow Core API, which gives you total flexibility and control over your application and lets you implement your ideas in a relatively short time.



Why Do We Need Keras?

- Keras is an API that was made to be easy to learn for people. Keras was made to be simple. It offers consistent & simple APIs, reduces the actions required to implement common code, and explains user error clearly.
- Prototyping time in Keras is less. This means that your ideas can be implemented and deployed in a shorter time. Keras also provides a variety of deployment options depending on user needs.
- Languages with a high level of abstraction and inbuilt features are slow and building custom features in then can be hard. But Keras runs on top of TensorFlow and is relatively fast. Keras is also deeply integrated with TensorFlow, so you can create customized workflows with ease.

Why Do We Need Keras?

- The research community for Keras is vast and highly developed. The documentation and help available are far more extensive than other deep learning frameworks.
- Keras is used commercially by many companies like Netflix, Uber, Square, Yelp, etc which have deployed products in the public domain which are built using Keras.

Apart from this, Keras has features such as :

- It runs smoothly on both CPU and GPU.
- It supports almost all neural network models.
- It is modular in nature, which makes it expressive, flexible, and apt for innovative research.

How to Build a Model in Keras?

The below diagram shows the basic steps involved in building a model in Keras:

- Define a network: In this step, you define the different layers in our model and the connections between them. Keras has two main types of models: Sequential and Functional models. You choose which type of model you want and then define the dataflow between them.

How to Build a Model in Keras?

- Compile a network: To compile code means to convert it in a form suitable for the machine to understand. In Keras, the model.compile() method performs this function. To compile the model, we define the loss function which calculates the losses in our model, the optimizer which reduces the loss, and the metrics which is used to find the accuracy of our model.
- Fit the network: Using this, we fit our model to our data after compiling. This is used to train the model on our data.
- Evaluate the network: After fitting our model, we need to evaluate the error in our model.
- Make Predictions: We use model.predict() to make predictions using our model on new data.

Application Of Keras

- Keras is used for creating deep models which can be productized on smartphones.
- Keras is also used for distributed training of deep learning models.
- Keras is used by companies such as Netflix, Yelp, Uber, etc.
- Keras is also extensively used in deep learning competitions to create and deploy working models, which are fast in a short amount of time.

Vanishing and Exploding Gradient Problem

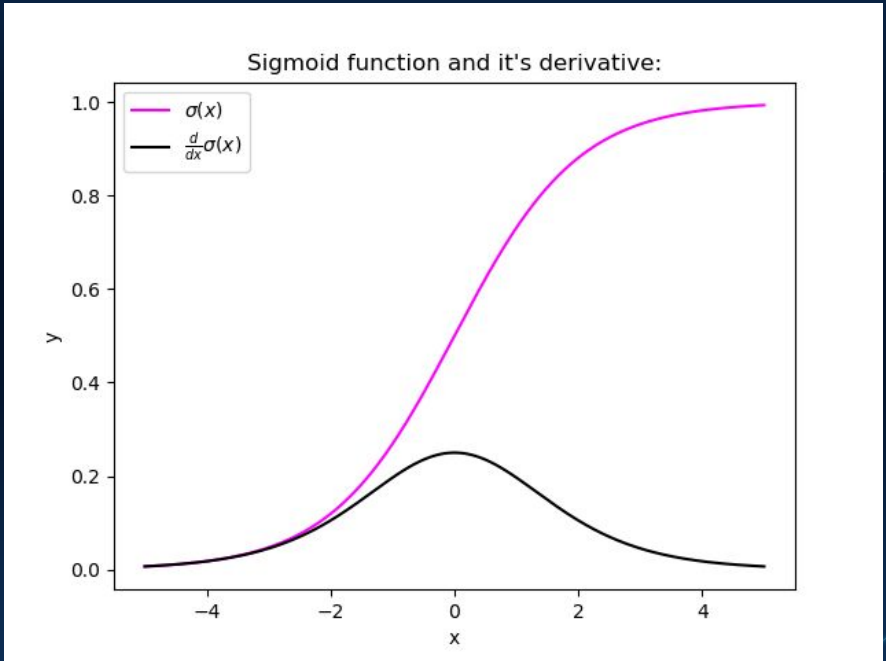
Problems

Vanishing –
As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. This is known as the vanishing gradients problem.

Exploding –
On the contrary, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the exploding gradients problem.

Why do the gradients even vanish/explode?

Certain activation functions, like the logistic function (sigmoid), have a very huge difference between the variance of their inputs and the outputs. In simpler words, they shrink and transform a larger input space into a smaller output space that lies between the range of [0,1].

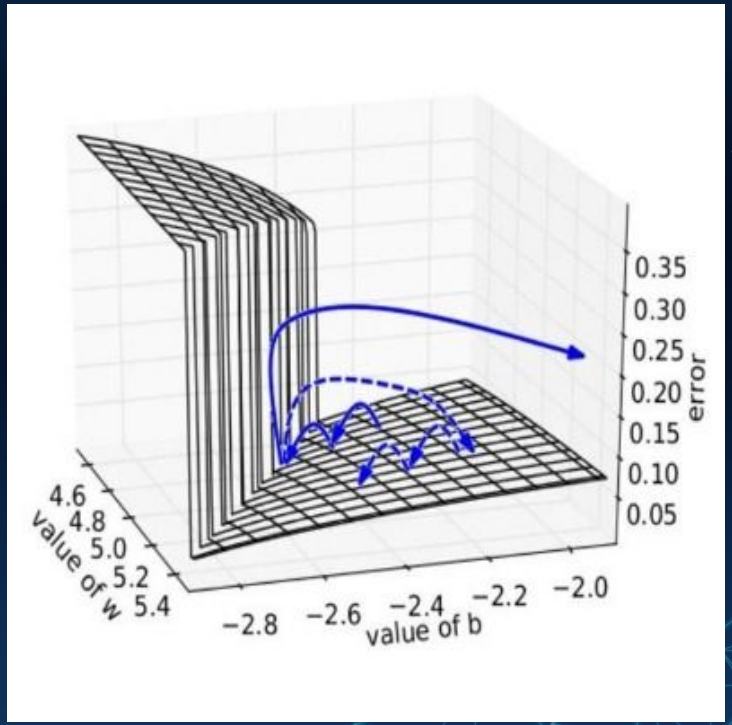


Why do the gradients even vanish/explode?

Observing the above graph of the Sigmoid function, we can see that for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero. Thus, when the backpropagation algorithm chips in, it virtually has no gradients to propagate backward in the network, and whatever little residual gradients exist keeps on diluting as the algorithm progresses down through the top layers. So, this leaves nothing for the lower layers.

Why do the gradients even vanish/explode?

Similarly, in some cases suppose the initial weights assigned to the network generate some large loss. Now the gradients can accumulate during an update and result in very large gradients which eventually results in large updates to the network weights and leads to an unstable network. The parameters can sometimes become so large that they overflow and result in NaN values.



How to know if our model is suffering from the Exploding/Vanishing gradient problem?

Exploding

- There is an exponential growth in the model parameters.
- The model weights may become NaN during training.
- The model experiences avalanche learning.

Vanishing

- The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all).
- The model weights may become 0 during training.
- The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations.

Vanishing and Exploding Gradient Problem

Certainly, neither do we want our signal to explode or saturate nor do we want it to die out. The signal needs to flow properly both in the forward direction when making predictions as well as in the backward direction while calculating gradients.

Solutions

Some techniques that can be used to fix the respective problems.

1. Proper Weight Initialization
2. Using Non-saturating Activation Functions
3. Batch Normalization
4. Gradient Clipping

1. Proper Weight Initialization

In their paper, researchers Xavier Glorot, Antoine Bordes, and Yoshua Bengio proposed a way to remarkably alleviate this problem.
For the proper flow of the signal, the authors argue that:

1. The variance of outputs of each layer should be equal to the variance of its inputs.
2. The gradients should have equal variance before and after flowing through a layer in the reverse direction.

Although it is impossible for both conditions to hold for any layer in the network until and unless the number of inputs to the layer (fan_{in}) is equal to the number of neurons in the layer (fan_{out}), but they proposed a well-proven compromise that works incredibly well in practice: randomly initialize the connection weights for each layer in the network as described in the following equation which is popularly known as *Xavier initialization (after the author's first name) or Glorot initialization (after his last name).*

1. Proper Weight Initialization

$$\text{where } fan_{avg} = (fan_{in} + fan_{out})/2$$

Normal distribution with mean 0 and variance σ2 = 1/ fan_{avg}
Or a uniform distribution between -r and +r , with r = sqrt(3 / fan_{avg})
Following are some more very popular weight initialization strategies for different activation functions, they only differ by the scale of variance and by the usage of either fan_{avg} or fan_{in}

1. Proper Weight Initialization

for uniform distribution, calculate r as: $r = \mathrm{sqrt}(3*\sigma^2)$

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Using the above initialization strategies can significantly speed up the training and increase the odds of gradient descent converging at a lower generalization error.

Wait, but how do we put these strategies into code ?

Keras does it for us.

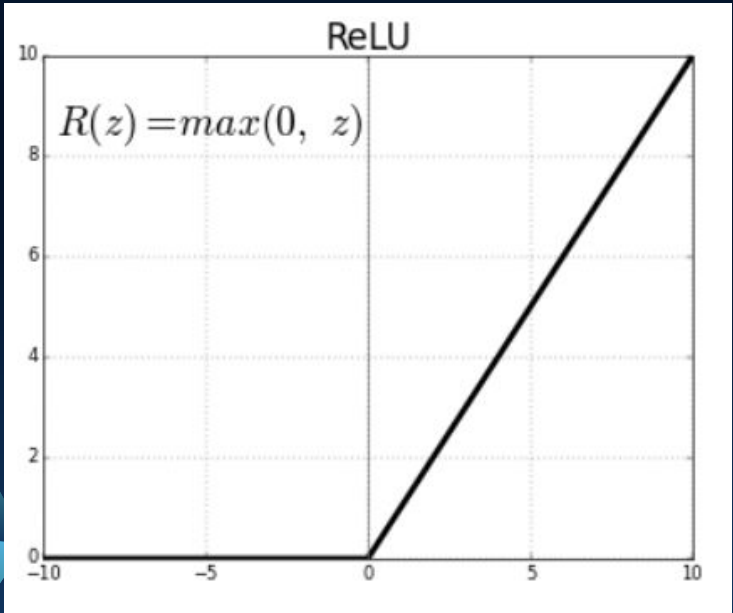
2. Using Non-saturating Activation Functions

In an earlier slides, while studying the nature of sigmoid activation function, we observed that its nature of saturating for larger inputs (negative or positive) came out to be a major reason behind the vanishing of gradients thus making it non-recommendable to use in the hidden layers of the network.

So to tackle the issue regarding the saturation of activation functions like sigmoid and tanh, we must use some other non-saturating functions like ReLu and its alternatives.

2. Using Non-saturating Activation Functions

ReLU (Rectified Linear Unit)



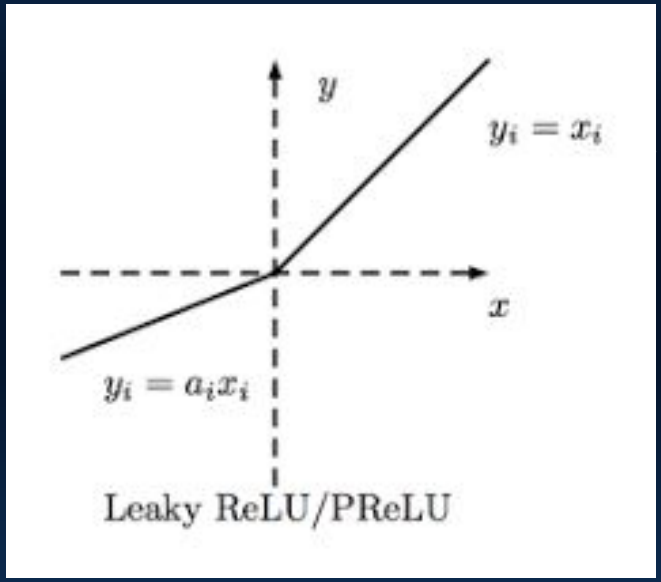
- Relu(z) = max(0,z)
- Outputs 0 for any negative input.
- Range: [0, infinity]

Unfortunately, the ReLu function is also not a perfect pick for the intermediate layers of the network “in some cases”. It suffers from a problem known as dying ReLus wherein some neurons just die out, meaning they keep on throwing 0 as outputs with the advancement in training.

2. Using Non-saturating Activation Functions

Some popular alternative functions of the ReLU that mitigates the problem of vanishing gradients when used as activation for the intermediate layers of the network are LReLU, PReLU, ELU, SELU :

LReLU (Leaky ReLU)

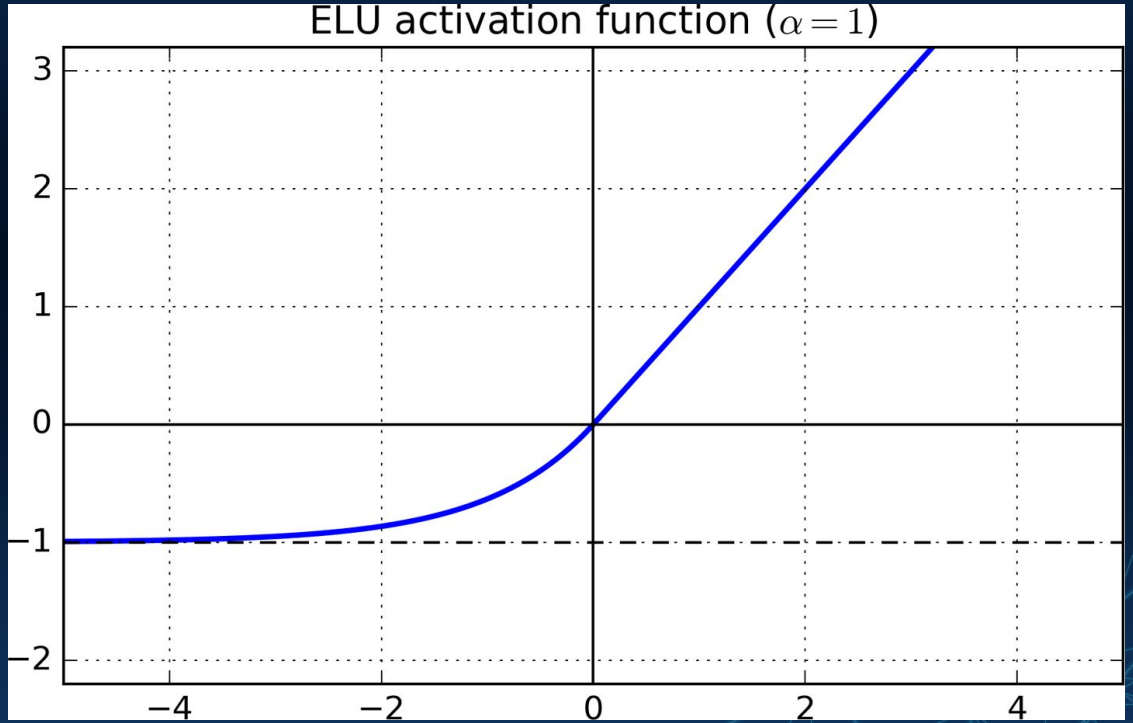


2. Using Non-saturating Activation Functions

- $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$
- The amount of “leak” is controlled by the hyperparameter α, it is the slope of the function for $z < 0$.
- The smaller slope for the leak ensures that the neurons powered by leaky Relu never die; although they might venture into a state of coma for a long training phase they always have a chance to eventually wake up.
- α can also be trained, that is, the model learns the value of α during training. This variant wherein α is now considered a parameter rather than a hyperparameter is called parametric leaky ReLu (PReLU).

2. Using Non-saturating Activation Functions

ELU (Exponential Linear Unit)



2. Using Non-saturating Activation Functions

- For $z < 0$, it takes on negative values which allow the unit to have an average output closer to 0 thus alleviating the vanishing gradient problem
- For $z < 0$, the gradients are non zero. This avoids the dead neurons problem.
- For $\alpha = 1$, the function is smooth everywhere, this speeds up the gradient descent since it does not bounce right and left around $z=0$.
- A scaled version of this function (SELU: Scaled ELU) is also used very often in Deep Learning.

3. Batch Normalization

Using He initialization along with any variant of the ReLU activation function can significantly reduce the chances of vanishing/exploding problems at the beginning. However, it does not guarantee that the problem won't reappear during training.

In 2015, Sergey Ioffe and Christian Szegedy proposed a paper in which they introduced a technique known as Batch Normalization to address the problem of vanishing/exploding gradients.

3. Batch Normalization

The Following key points explain the intuition behind BN and how it works:

- It consists of adding an operation in the model just before or after the activation function of each hidden layer.
- This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting.
- In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs.
- To zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation.
- It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name "Batch Normalization").

4. Gradient Clipping

Another popular technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called Gradient Clipping.

- This optimizer will clip every component of the gradient vector to a value between –1.0 and 1.0.
- Meaning, all the partial derivatives of the loss w.r.t each trainable parameter will be clipped between –1.0 and 1.0

    optimizer = keras.optimizers.SGD(clipvalue = 1.0)

4. Gradient Clipping

- The threshold is a hyperparameter we can tune.
- The orientation of the gradient vector may change due to this: for eg, let the original gradient vector be [0.9, 100.0] pointing mostly in the direction of the second axis, but once we clip it by some value, we get [0.9, 1.0] which now points somewhere around the diagonal between the two axes.
- To ensure that the orientation remains intact even after clipping, we should clip by norm rather than by value.

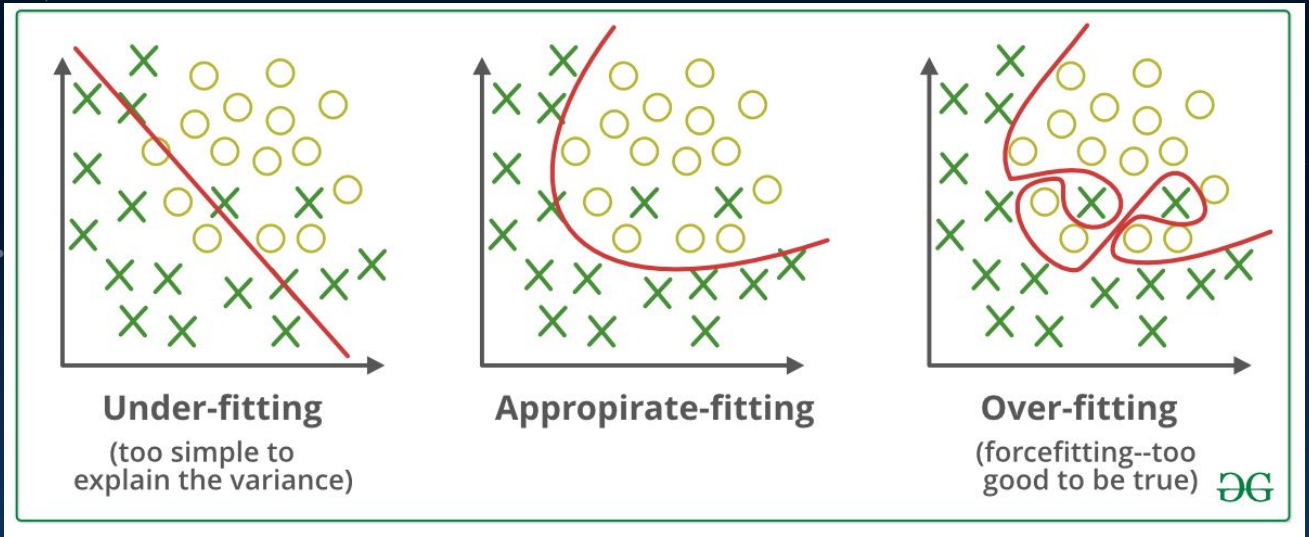
  optimizer = keras.optimizers.SGD(clipnorm = 1.0)
- Now the whole gradient will be clipped if the threshold we picked is less than its $\ell 2$ norm. For eg: if clipnorm=1 then the vector [0.9, 100.0] will be clipped to [0.00899, 0.999995] , thus preserving its orientation.

Part II

Regularization, Optimizers, Hyperparameters and tuning of the same

Regularization

Overfitting is a phenomenon that occurs when a Machine Learning model is constraint to training set and not able to perform well on unseen data.



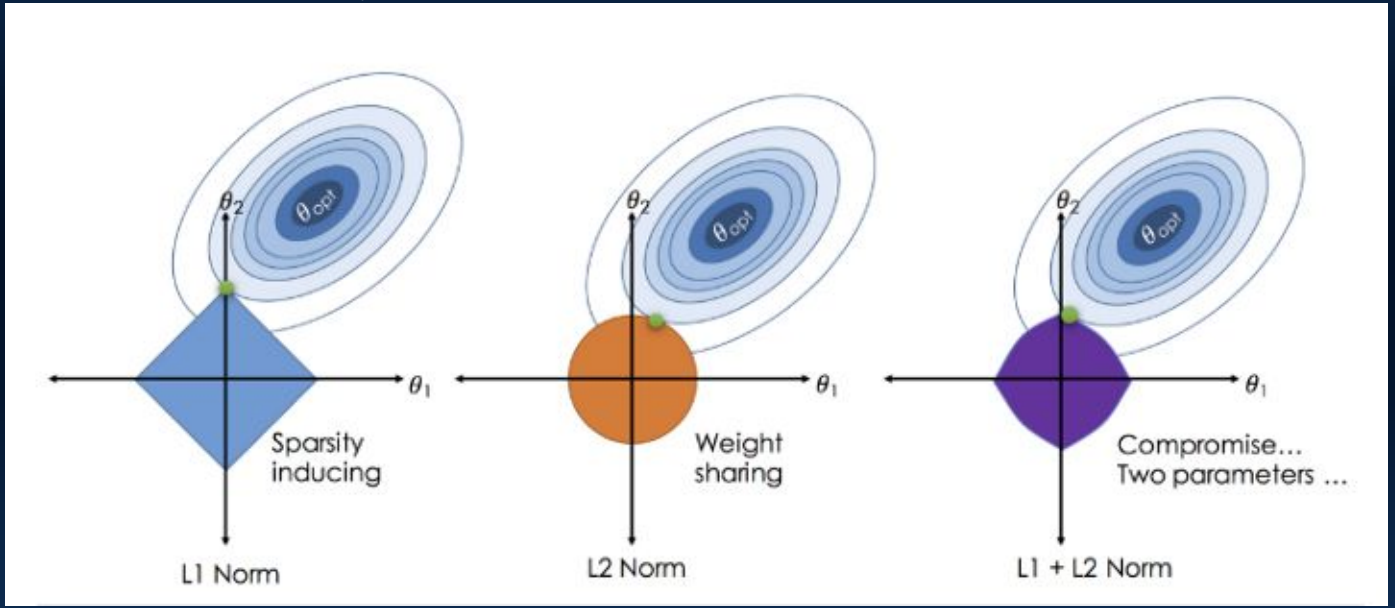
Regularization

Regularization aims to address a few typical model problems by minimizing model complexity, punishing the loss function, and reducing model overfitting.
Regularization can be viewed as a method to reduce model overfitting and variance by demanding some additional bias and requiring the search for an ideal penalty hyperparameter.

The commonly used regularization techniques are :

1. L1 regularization
2. L2 regularization
3. Dropout regularization

Regularization



Regularization

The penalty added by Lasso is equal to the magnitude of the coefficients in absolute values. Thus, this restricts the magnitude of the regression equation's coefficients. Remember that this could result in sparse models with some coefficients becoming zero.

Ridge is also introducing a penalty term, but in this instance, the punishment term will be equal to the square of the magnitude of the coefficients, meaning that all coefficients are actually decreased by the same factor but are not necessarily eliminated.

The basic function of an elastic net is to combine L1 and L2, with the addition of a parameter called alpha that determines their relative importance. Alpha should have a value between 0 and 1. Elastic Net models are equivalent to Lasso models if alpha is equal to 1, or identical to Ridge models if alpha is equal to 0.

How L1 Regularization brings Sparsity

Sparse data -

A variable with sparse data is one in which a relatively high percentage of the variable's cells do not contain actual data. Such "empty," or NA, values take up storage space in the file.

L1 regularizer :

$$||w||1=(|w1|+|w2|+ . . . + |wn|$$

L2 regularizer :

$$||w||2=(|w1|2+|w2|2+ . . . + |wn|2)1/2$$

(where w1,w2 … wn are ‘d‘ dimensional weight vectors)

Regularization

Now while optimization, that is done based on the concept of Gradient Descent algorithm, it is seen that if we use L1 regularization, it brings sparsity to our weight vector by making smaller weights as zero. Let's see how but first let's understand some basic concepts

Sparse vector or matrix: A vector or matrix with a maximum of its elements as zero is called a sparse vector or matrix.

$$\begin{pmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 22 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 45 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{pmatrix}$$

Regularization

Gradient Descent Algorithm :

$W_t = W_{(t-1)} - \eta * \left(\partial L(w)/\partial(w) \right)_{W(t-1)}$

(where η is a small value called learning rate)

As per the gradient descent algorithm, we get our answer when convergence occurs. Convergence occurs when the value of Wt doesn't change much with further iterations, or we can say when we get minima i.e. $\left(\partial(\text{Loss})/\partial w \right)_{W(t-1)}$ becomes approximately equal to 0 an thus, Wt ~ Wt-1.

Regularization

Let's assume we have a linear regression optimization problem(we can take that for any model), with its equation as:

argmin(W) [loss + (regularizer)]

i.e. Find that W which will minimize the loss function

Here we will ignore the loss term and only focus on the regularization term for making it easier for us to analyze our task.

Case1 (L1 taken) :

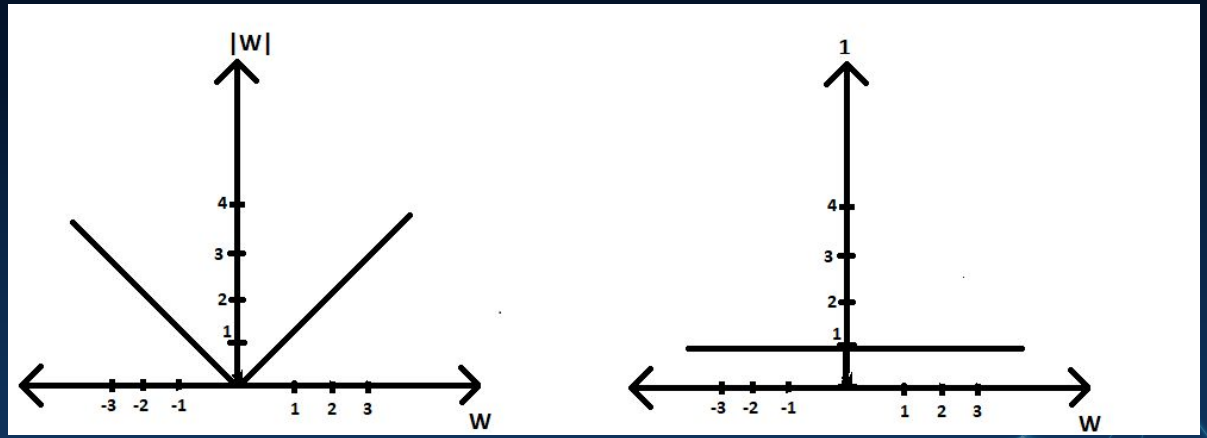
Optimisation equation= argmin(W) |W|

$(\partial|W|/\partial w) = 1$

thus, according to GD algorithm $W_t = W_{t-1} - \eta * 1$

Regularization

Here as we can see our loss derivative comes to be constant, so the condition of convergence occurs faster because we have only η in our subtraction term and it is not being multiplied by any smaller value of W. Therefore our Wt tends towards 0 in a few iterations only. But this will get hinder in our condition to convergence to occur as we will see in the next case.



Regularization

Case2 (L2 taken) :

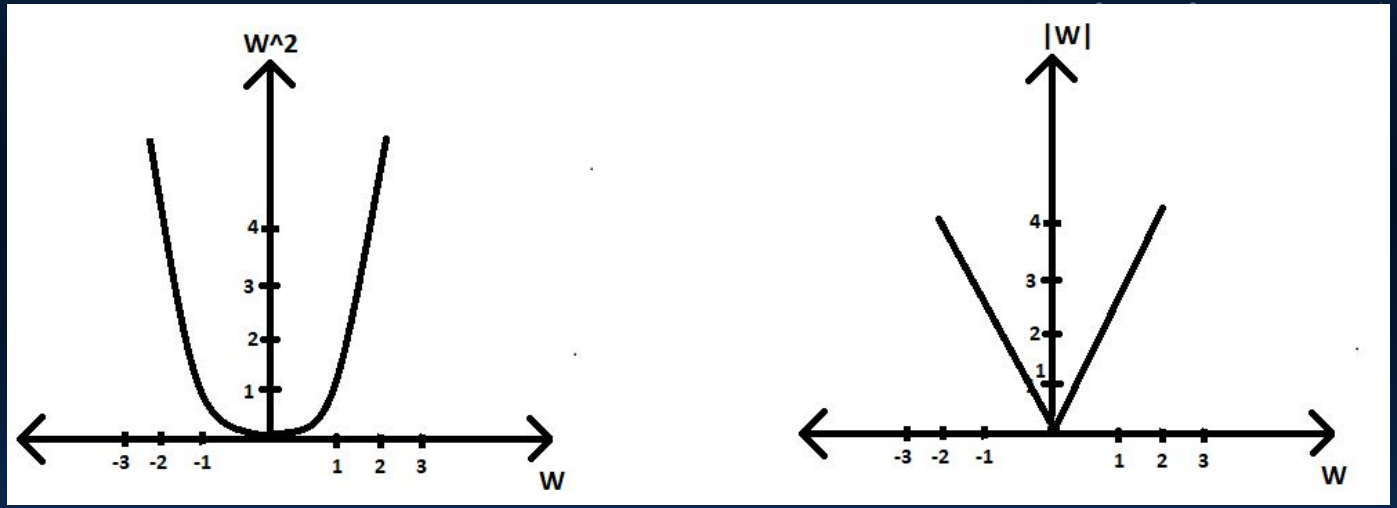
Optimisation equation=argmin(W) |W|2

(∂|W|2/∂W) = 2|W|

thus, according to GD algorithm Wt = Wt-1 - 2*η*|W|

Hence, we can see that our loss derivative term is not constant and thus for smaller values of W, our condition of convergence will not occur faster(or maybe at all) because we have a smaller value of W getting multiplied with η and thus making the whole term to be subtracted even smaller. Therefore, after a few iterations, our Wt becomes a very small constant value but not zero. Hence, not contributing to the sparsity of the weight vector.

Regularization



Use of Regularization

In general, regularization is adopted universally as simple data models generalize better and are less prone to overfitting. Examples of regularization, included;

- K-means: Restricting the segments for avoiding redundant groups.
- Neural networks: Confining the complexity (weights) of a model.
- Random Forest: Reducing the depth of tree and branches (new features)

S.No	L1 Regularization	L2 Regularization
1	Panelizes the sum of absolute value of weights.	penalizes the sum of square weights.
2	It has a sparse solution.	It has a non-sparse solution.
3	It gives multiple solutions.	It has only one solution.
4	Constructed in feature selection.	No feature selection.
5	Robust to outliers.	Not robust to outliers.
6	It generates simple and interpretable models.	It gives more accurate predictions when the output variable is the function of whole input variables.
7	Unable to learn complex data patterns.	Able to learn complex data patterns.
8	Computationally inefficient over non-sparse conditions.	Computationally efficient because of having analytical solutions.

Regularization

Conclusion

In order to prevent overfitting, regularization is most-approaches mathematical technique, it achieves this by panelizing the complex ML models via adding regularization terms to the loss function/cost function of the model.

- L1 regularization gives output in binary weights from 0 to 1 for the model's features and is adopted for decreasing the number of features in a huge dimensional dataset.
- L2 regularization disperse the error terms in all the weights that leads to more accurate customized final models.

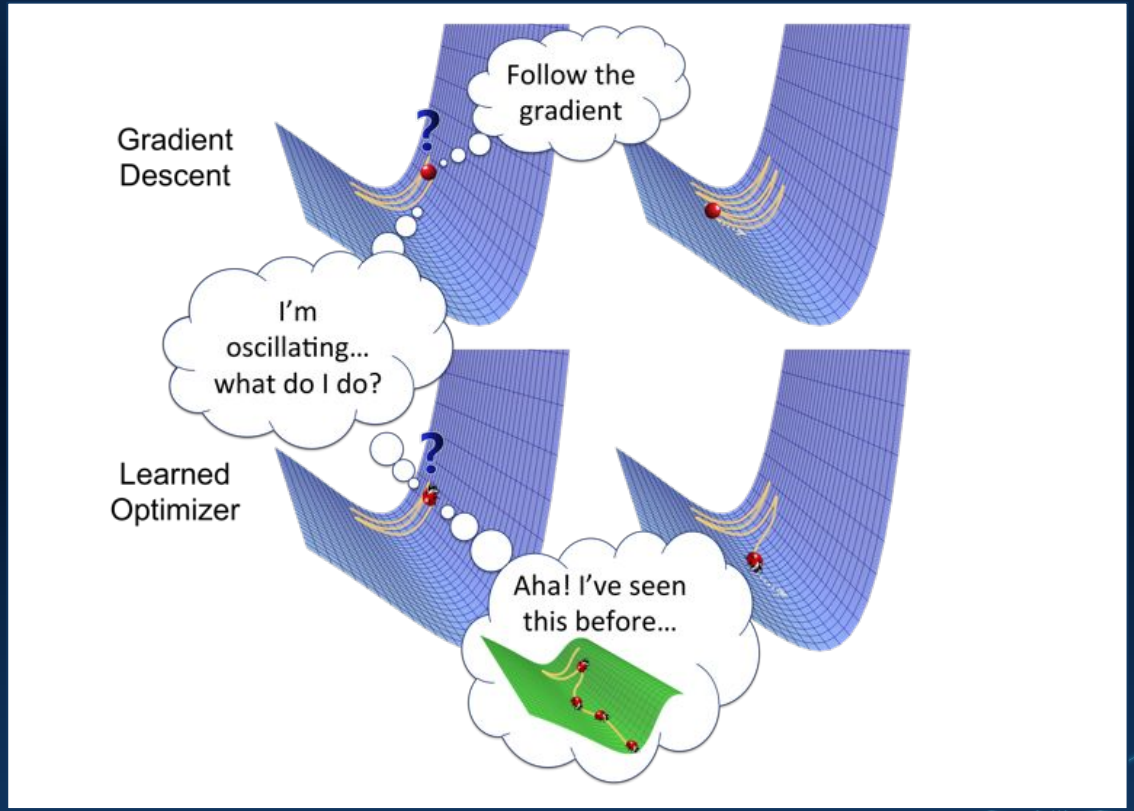
Optimizers

Optimizers are algorithms or methods used to minimize an error function(loss function)or to maximize the efficiency of production. Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights & Biases. Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.

Types of optimizers -

Different types of optimizers and how they exactly work to minimize the loss function.

Optimizers


Gradient Descent
Follow the gradient
?
I'm oscillating... what do I do?
Learned Optimizer
?
Aha! I've seen this before...

Various Optimization Algorithms For Training Neural Network

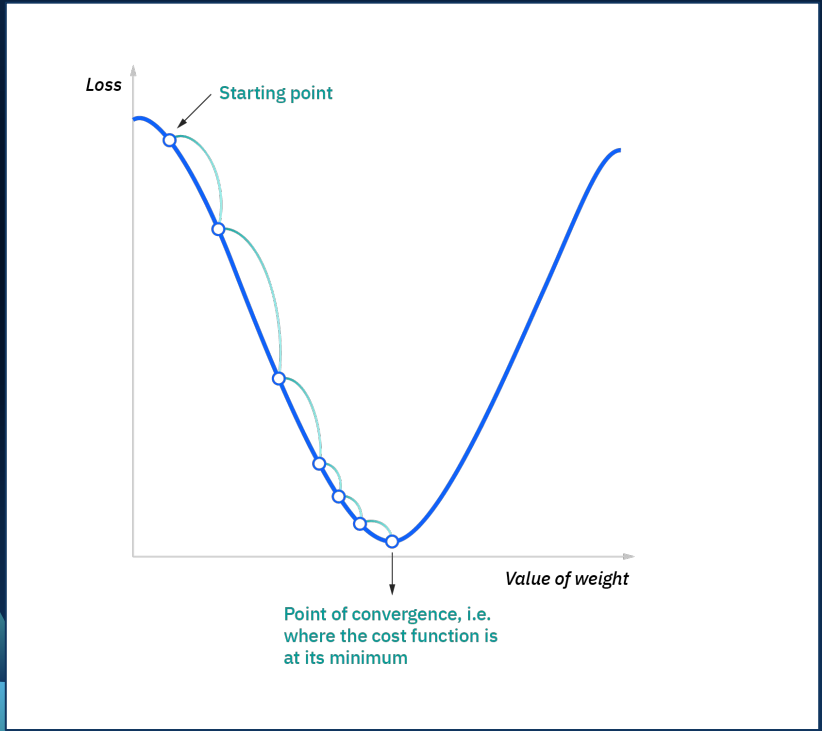
- Gradient Descent
- Stochastic Gradient Descent
- Mini Batch Gradient Descent
- SGD with Momentum
- AdaGrad(Adaptive Gradient Descent)
- Adam(Adaptive Moment Estimation)
- adaDelta

Gradient Descent

Gradient Descent is the most basic but most used optimization algorithm. It's used heavily in linear regression and classification algorithms. Backpropagation in neural networks also uses a gradient descent algorithm.

Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function. It calculates that which way the weights should be altered so that the function can reach a minima. Through backpropagation, the loss is transferred from one layer to another and the model's parameters also known as weights are modified depending on the losses so that the loss can be minimized.

Gradient Descent



$$W_{new} = W_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

Gradient Descent

Advantages of Gradient Descent

1. Easy to understand
2. Easy to implement

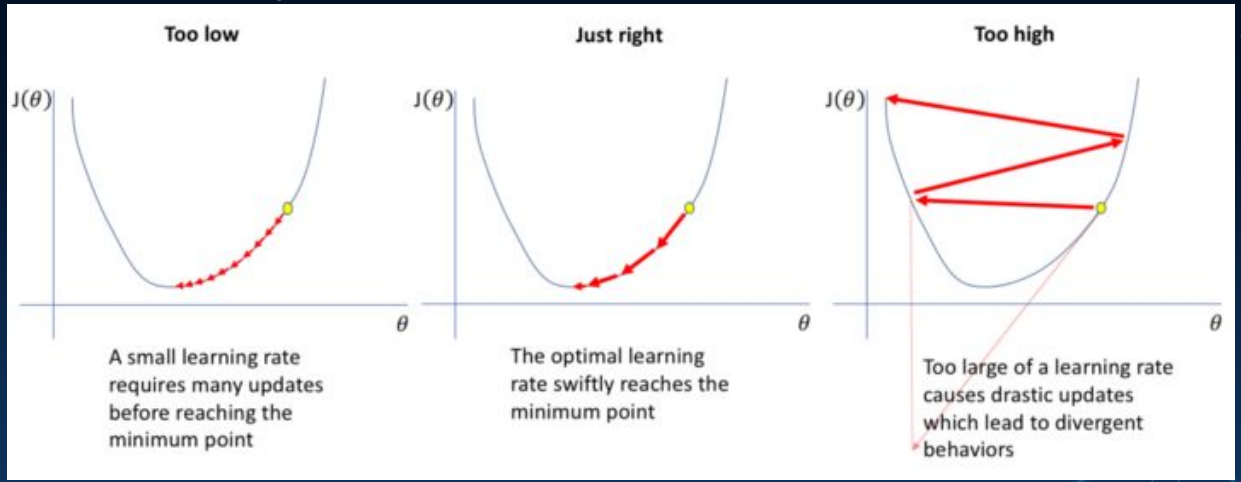
Disadvantages of Gradient Descent

1. Because this method calculates the gradient for the entire data set in one update, the calculation is very slow.
2. It requires large memory and it is computationally expensive.

Gradient Descent

Learning Rate

How big/small the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.



Stochastic Gradient Descent

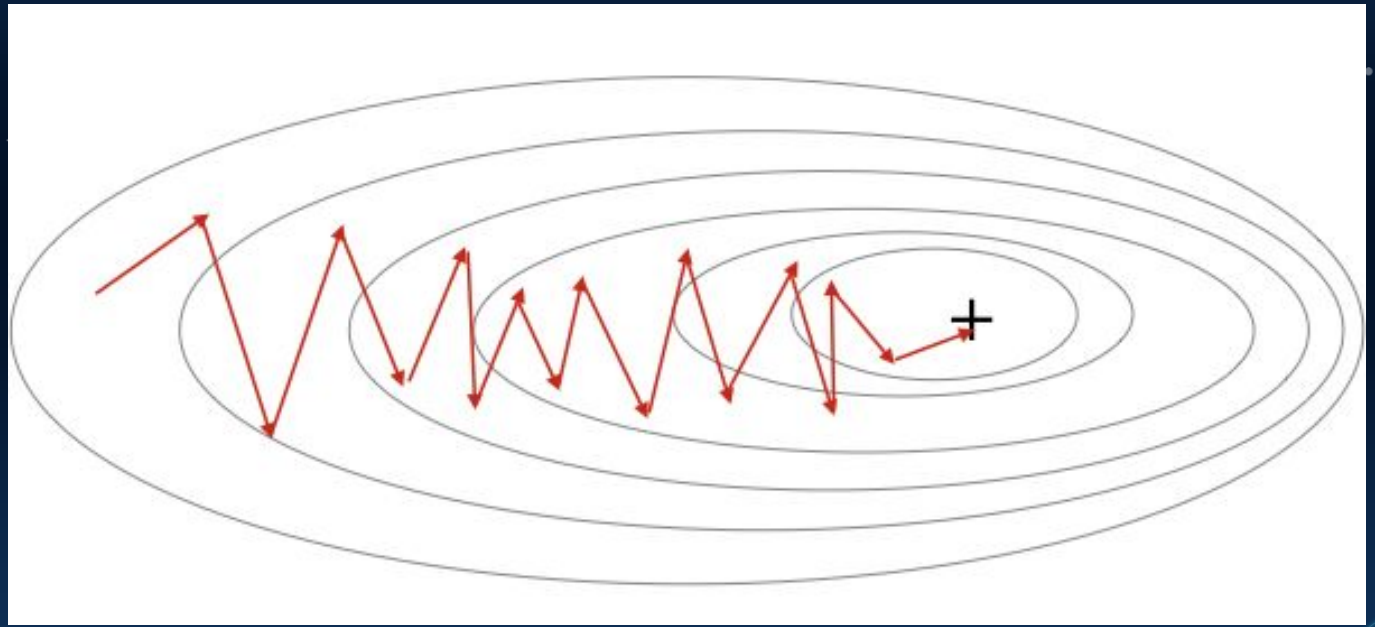
It's a variant of Gradient Descent. It tries to update the model's parameters more frequently. In this, the model parameters are altered after computation of loss on each training example. So, if the dataset contains 1000 rows SGD will update the model parameters 1000 times in one cycle of dataset instead of one time as in Gradient Descent.

$$\text{for n} = 1 : \text{N}$$

$$w_i^n := w_i^{n-1} - \eta \frac{\partial E_n(\text{w})}{\partial w_i}$$

$$:= w_i^{n-1} - \eta (y(x_n, \text{w}) - t_n) \frac{\partial y(x_n, \text{w})}{\partial w_i}$$

Stochastic Gradient Descent



Stochastic Gradient Descent

Advantages of Stochastic Gradient Descent

1. Frequent updates of model parameter
2. Requires less Memory.
3. Allows the use of large data sets as it has to update only one example at a time.

Disadvantages of Stochastic Gradient Descent

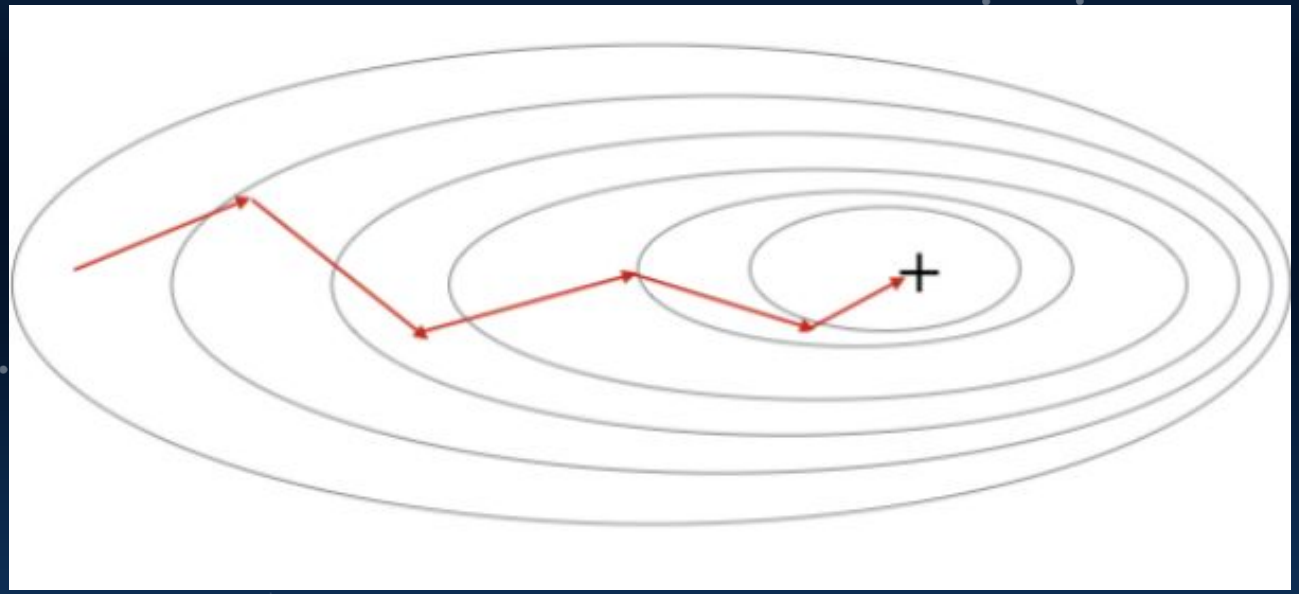
1. The frequent can also result in noisy gradients which may cause the error to increase instead of decreasing it.
2. High Variance.
3. Frequent updates are computationally expensive.

Mini-Batch Gradient Descent

Mini-Batch Gradient Descent

It’s best among all the variations of gradient descent algorithms. It is an improvement on both SGD and standard gradient descent. It updates the model parameters after every batch. So, the dataset is divided into various batches and after every batch, the parameters are updated.

Mini Batch Gradient Descent



Mini Batch Gradient Descent

Advantages of Mini Batch Gradient Descent:

1. It leads to more stable convergence.
2. more efficient gradient calculations.
3. Requires less amount of memory.

Disadvantages of Mini Batch Gradient Descent

1. Mini-batch gradient descent does not guarantee good convergence,
2. If the learning rate is too small, the convergence rate will be slow. If it is too large, the loss function will oscillate or even deviate at the minimum value.

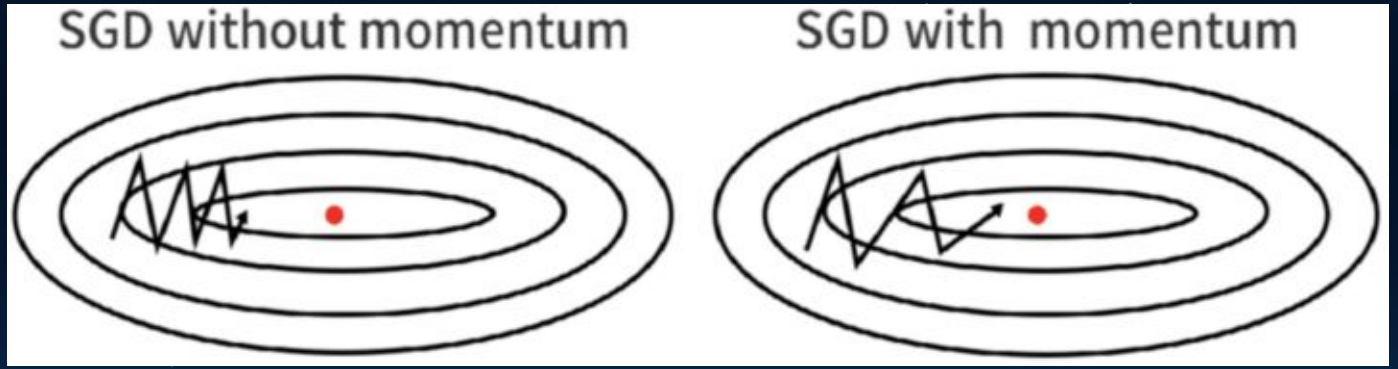
SGD with Momentum

SGD with Momentum

SGD with Momentum is a stochastic optimization method that adds a momentum term to regular stochastic gradient descent. Momentum simulates the inertia of an object when it is moving, that is, the direction of the previous update is retained to a certain extent during the update, while the current update gradient is used to fine-tune the final update direction. In this way, you can increase the stability to a certain extent, so that you can learn faster, and also have the ability to get rid of local optimization.

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

SGD with Momentum



SGD with Momentum

Advantages of SGD with momentum

1. Momentum helps to reduce the noise.
2. Exponential Weighted Average is used to smoothen the curve.

Disadvantage of SGD with momentum

1. Extra hyperparameter is added.

AdaGrad(Adaptive Gradient Descent)

AdaGrad(Adaptive Gradient Descent)

In all the algorithms that we discussed previously the learning rate remains constant. The intuition behind AdaGrad is can we use different Learning Rates for each and every neuron for each and every hidden layer based on different iterations.

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

AdaGrad(Adaptive Gradient Descent)

Advantages of AdaGrad

1. Learning Rate changes adaptively with iterations.
2. It is able to train sparse data as well.

Disadvantage of AdaGrad

1. If the neural network is deep the learning rate becomes very small number which will cause dead neuron problem.

AdaDelta

AdaDelta

Adadelta is an extension of Adagrad and it also tries to reduce Adagrad's aggressive, monotonically reducing the learning rate and remove decaying learning rate problem. In Adadelta we do not need to set the default learning rate as we take the ratio of the running average of the previous time steps to the current gradient.

Advantages of Adadelta

1. The main advantage of AdaDelta is that we do not need to set a default learning rate.

Disadvantages of Adadelta

1. Computationally expensive

Adam (Adaptive Moment Estimation)

Adam (Adaptive Moment Estimation) works with momentums of first and second order. The intuition behind the Adam is that we don't want to roll so fast just because we can jump over the minimum, we want to decrease the velocity a little bit for a careful search. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta, Adam also keeps an exponentially decaying average of past gradients M(t).
M(t) and V(t) are values of the first moment which is the Mean and the second moment which is the uncentered variance of the gradients respectively.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Adam (Adaptive Moment Estimation)

Here, we are taking mean of M(t) and V(t) so that E[m(t)] can be equal to E[g(t)] where, E[f(x)] is an expected value of f(x).
To update the parameter:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Update the parameters
The values for β1 is 0.9 , 0.999 for β2, and (10 x exp(-8)) for ‘ϵ’.

Adam (Adaptive Moment Estimation)

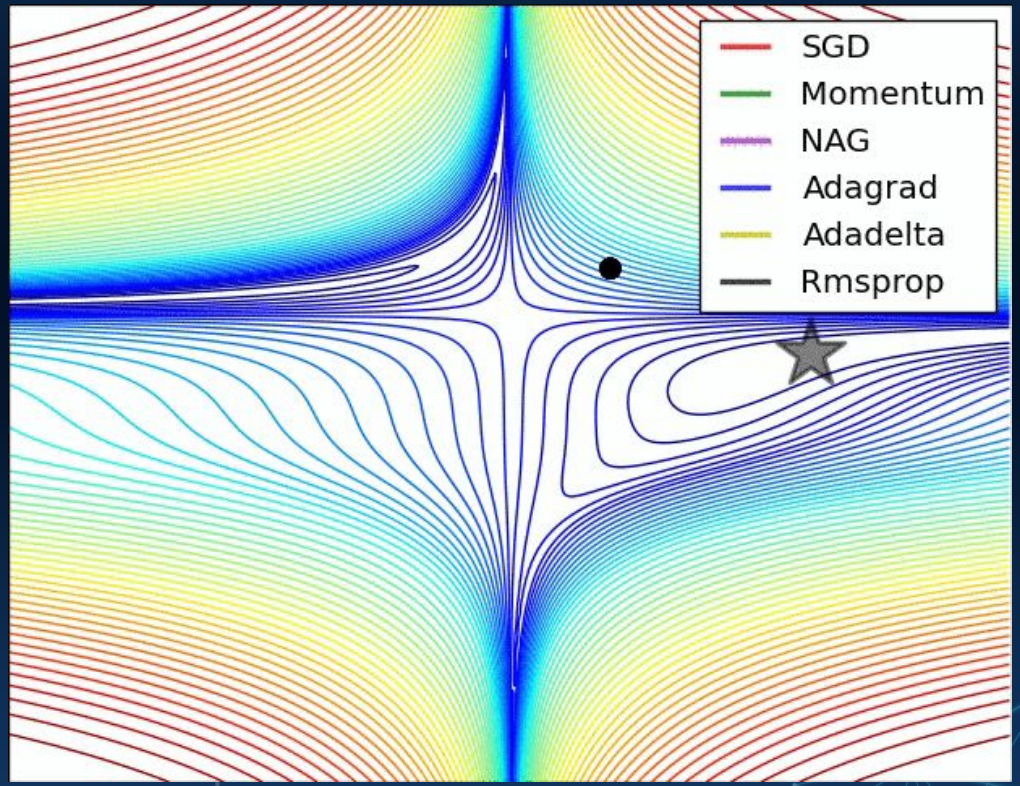
Advantages:

1. The method is too fast and converges rapidly.
2. Rectifies vanishing learning rate, high variance.

Disadvantages:

Computationally costly.

Optimizers Comparison



Optimizers

Conclusions

Adam is the best optimizers. If one wants to train the neural network in less time and more efficiently than Adam is the optimizer.
For sparse data use the optimizers with dynamic learning rate.
If, want to use gradient descent algorithm than min-batch gradient descent is the best option.

Hyperparameter

Hyperparameter tuning is an important part of developing a machine learning model.

What are hyperparameters? — The what
Parameter vs. hyperparameter

- Parameters are estimated from the dataset. They are part of the model equation. The equation below is a logistic regression model. Theta is the vector containing the parameters of the model.

$$h(x) = \frac{1}{1 + e^{-\theta^T x}}$$

- Hyperparameters are set manually to help in the estimation of the model parameters. They are not part of the final model equation.

Hyperparameter

Examples of hyperparameters in logistic regression

1. Learning rate (α). One way of training a logistic regression model is with gradient descent. The learning rate (α) is an important part of the gradient descent algorithm. It determines by how much parameter theta changes with each iteration.

$$\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \cdot \sum_{i=1}^{m} (h(x)^i - y^i) \cdot x_j$$

Hyperparameter

2. Regularization parameter (λ). The regularization parameter (λ), is a constant in the “penalty” term added to the cost function. Adding this penalty to the cost function is called regularization. There are two types of regularization — L1 and L2. They differ in the equation for penalty

$$L1 = \lambda \sum_{j=1}^{k} |\theta_j|$$

$$L2 = \lambda \sum_{j=1}^{k} \theta_j^2$$

Hyperparameter

In a linear regression, the cost function is simply the sum of squared errors. Adding an L2 regularization term and it becomes:

$$cost = \sum_{i=1}^{m} (y^i - h(x)^i)^2$$

$$cost_{regularization} = \sum_{i=1}^{m} (y^i - h(x)^i)^2 + \lambda \sum_{j=1}^{k} \theta_j^2$$

Hyperparameter

In logistic regression, the cost function is the binary cross entropy, or log loss, function. Adding a L2 regularization term and it becomes:

$$cost = -\sum_{i=1}^{m} y^i log(h(x^i)) + (1 - y^i) log(1 - h(x^i))$$

$$cost_{regularization} = -\sum_{i=1}^{m} y^i log(h(x^i)) + (1 - y^i) log(1 - h(x^i)) + \lambda \sum_{j=1}^{k} \theta_j^2$$

How do you go about tuning hyperparameters?

There are several strategies for tuning hyperparameters.

Two of them are Grid Search and Random Search.

Hyperparameter

Grid search is a tuning technique that attempts to compute the optimum values of hyperparameters. It is an exhaustive search that is performed on a the specific parameter values of a model. The model is also known as an estimator.
Grid search exercise can save us time, effort and resources.

Python Implementation

We can use the grid search in Python by performing the following steps:
1. Install sklearn library
2. Import sklearn library
3. Import your model (from sklearn.svm import SVC)

Hyperparameter

4. Create a list of hyperparameters dictionary
This is the key step.

Let’s consider that we want to find the optimal hyperparameter values for:

- kernal: We want the model to train itself on the following kernels and give us the best value amongst linear, poly, rbf, sigmoid and precomputed values
- C: we want the model to try the following values of C: [1,2,3,300,500]
- max_iter: we want the model to use the following values of max_iter: [1000,100000] and give us the best value.

We can create the required dictionary:

```
parameters = [{
'kernel': ['linear', 'poly', 'rbf', 'sigmoid', 'precomputed'],
'C': [1,2,3,300,500],
'max_iter': [1000,100000]}]
```

Hyperparameter

5. Instantiate GridSearchCV and pass in the parameters

```
clf = GridSearchCV(
      SVC(), parameters, scoring='accuracy'
  )
clf.fit(X_train, y_train)
```

Note: we decided to use the precision scoring measure to assess the performance.

6. Finally, print out the best parameters:

```
print(clf.best_params_)
```

That’s all.

Hyperparameter

Random Search
Random searches are very similar to grid searches.
However, instead of testing every combination of hyperparameters, random searches only test a certain number of combinations that are selected randomly.
At first glance, random searches may seem unappealing. After all, if you can't test every hyperparameter combination, you are unlikely to find the best one.
However, this approach does come with certain perks.
Firstly, since the random search tests fewer model architectures, it requires less time and less computation to obtain results.
Although the random search may not necessarily find the best possible set of hyperparameters, it can provide a model that comes close to the ideal model in terms of performance.

Hyperparameter

Why tuning hyperparameters is important?

Effect of regularization
Effect of learning rate (and regularization)

Thank You!