Soft Computing

Soft Computing refers to the collection of Computational techniques in computer science, artificial intelligence, machine learning which attempt to study, model and analyze very Complex phenomena.

Complex phenomena or Complex System
- Complex System in Biology
- Complex System in Medicine
- Complex System in Humanities
- Complex System in Management

Soft Computing	Hard Computing (with Zadeh)
1) For Complex System	For Conventional System
2) Tolerant of Imprecision	Intolerant of imprecision
3) Tolerant of Uncertainty	Intolerant of Uncertainty
4) Tolerant of partial truth	Intolerant of partial truth
5) Tolerant of approximation	Intolerant of approximation

Principal Components of Soft Computing

1) Fuzzy logic
2) Neural Networks
3) Evolutionary Computation
4) Machine Learning
5) Probabilistic Reasoning

Artificial Neural Network

↓

How human brain works ?

Garry Kasparov, the world chess champion

↓

After his defeat in New york in 1997 he said.

"If we were playing a real competitive match, I would tear down "Deep Blue" into pieces".

↓

Turning point for intelligent System

The IBM supercomputer called Deep Blue was capable of analysing 200 million positions a second

Artificial Neural Network (ANN) is a Sub-part of Machine Learning also

Machine Learning

Machine learning enables Computer/machine to
- Learn from experience ✓
- Learn from example ✓
- Learn from environment ✓

Neural Network

The brain consists of a densely interconnected set of nerve cells, or basic information processing units, called neurons.

Neuron → Basic information processing unit

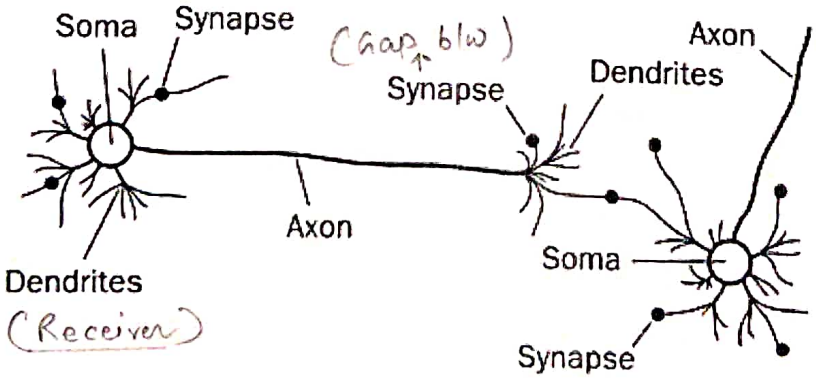
Human brain incorporates nearly 100 billion neurons & 60 Trillion connections Synapses between neuron

100 Billion Neurons

60 Trillion Connection

Signals are propagated from one neuron to other through Complex electro chemical reaction

Structure of Biological Neuron

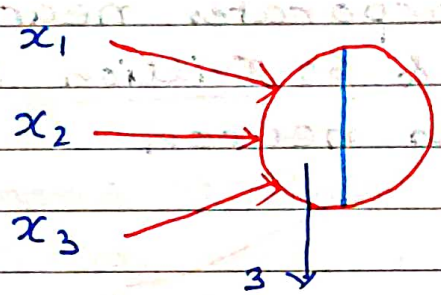


1) **Dendrites** → Input

It receives signals from neighboring neurons and carry them towards the cell body.

2) **Soma or Nucleus** → Neuron

It receives & accumulates the signals coming from dendrites.



It 'fires' when the amount of accumulated signal crosses a threshold, sending a spike along its axon.

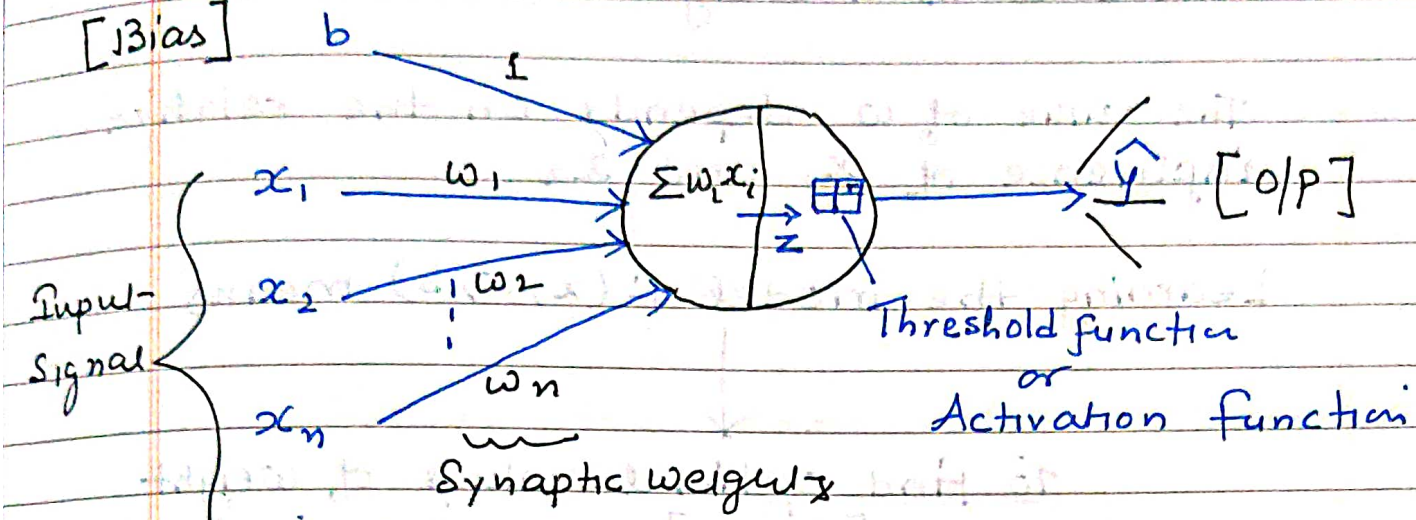
3) **Axon** → output

It gets spike from Soma and passes it on to the neighboring neurons through axon terminals. (To the adjacent dendrite of the neighboring neurons)

Note : Synapse :- weight

Small gap b/w axon terminal of one neuron and the adjacent dendrite of the neighboring neuron.

Exploring the Artificial Neuron



1) $$z = b + \sum_{i=1}^{n} \omega_i x_i$$

2) $$\hat{y} = f(z)$$ f → Activation function

Biological Neuron	Artificial Neuron
Cell	Neuron
Dendrites	Weights or interconnection
Soma	Net input -
Axon	Output -

Example

Real estate problem

↓ Whether a property will get a buyer or not depends upon [Suppose market - Condition is not good]

- Number of Bedroom in apartment - x_1
- Floor in which the apartment is located. x_2

Goal → To learn the model $f(x, w, b)$
Where w is a weight vector

The value of w depends on the relative influence of x_1 and x_2

Learning the model $f(x, w, b)$ means

↓

To find optimal values of weight vector $w = [w_1\ w_2]$ and offset or bias b

$$\text{Input vector} = \begin{bmatrix} x_1 \\ x_2 \\ b \end{bmatrix}$$

$$\text{weight vector} = \begin{bmatrix} w_1 \\ w_2 \\ 1 \end{bmatrix}$$

Sign function

$$f(x, w, b) = \begin{cases} 0 & \text{if } x^T \cdot w + b < 0 \\ 1 & \text{if } x^T \cdot w + b \geq 0 \end{cases}$$

$$x^T \cdot w + b$$

$$\begin{bmatrix} x_1 & x_2 & b \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ 1 \end{bmatrix} = w_1 x_1 + w_2 x_2 + b$$

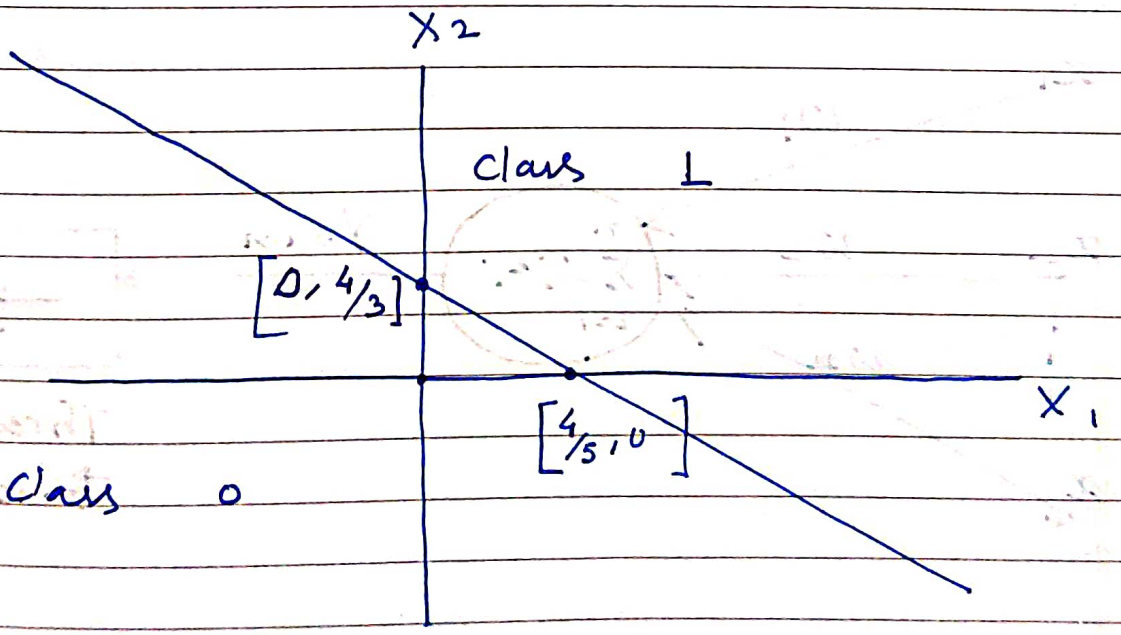
$$F(x, w, b) = \begin{cases} 0 & \text{if } w_1x_1 + w_2x_2 + b < 0 \\ 1 & \text{if } w_1x_1 + w_2x_2 + b \geq 0 \end{cases}$$

Assume the learn values of parameter

$$\text{vector } w = \begin{bmatrix} 5 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ 1 \end{bmatrix}$$

$$\text{bias } b = -4$$

$$F(x, w, b) = \begin{cases} 0 & \text{if } 5x_1 + 3x_2 - 4 < 0 \\ 1 & \text{if } 5x_1 + 3x_2 - 4 \geq 0 \end{cases}$$



Early Implementations of ANN

McCulloch-Pitts Model of Neuron 1943

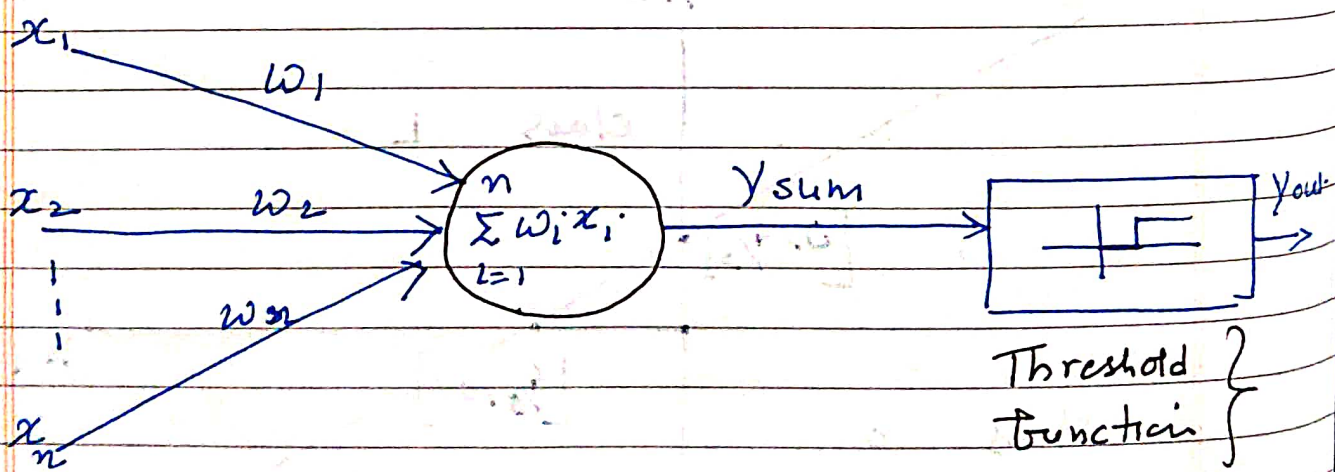
MP neuron is the earliest artificial neuron model.

MP neuron has two types of inputs

- Excitatory (weight of positive magnitude)
- Inhibitory (weight of Negative Magnitude)

The inputs of the MP-neuron could be either 0 or 1.

Threshold function as activation function



Simple MP neurons can be used to design logical operations.

A person carries an umbrella if it is sunny or it is raining.
There are 4 given situations. we need to decide when John will carry the umbrella.

Situation 1 : It is not raining, nor it is sunny
Situation 2 : It is not raining, but it is sunny
Situation 3 : It is raining, and it is not sunny
Situation 4 : It is raining as well as sunny

x_1 → is It raining ?
x_2 → is it Sunny ?

x_1 → 1 [It is raining]
→ 0 [It is not raining]

x_2 → 1 [It is sunny]
→ 0 [It is not sunny]

Truth table

Situation	x_1	x_2	Ysum	Actual Yout
1	0	0	0	0
2	0	1	1	1
3	1	0	1	1
4	1	1	2	1

$$Ysum = \sum_{i=1}^{2} w_i x_i = w_1 x_1 + w_2 x_2$$

Assume $w_1 = w_2 = 1$

Logical – OR

$$Y_{out} = f(Y_{sum}) = \begin{cases} 1 & x \geq 1 \\ 0 & x < 1 \end{cases}$$

Example-2

Implement AND function using McCulloch-Pitts neuron (take binary data)

Truth Table

x_1	x_2	Y
0	0	0
0	1	0
1	0	0
1	1	1

Assume $w_1 = w_2 = 1$

x_1	x_2	Y_{sum}/Y_{in}	Y_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

threshold value $\theta \geq nw - p$

$n = 2$

$w = 1$

$P = 0$

$\theta \geq 2 \times 1 - 0$

$\theta \geq 2$

Thus the output of neuron Y can be written as

$$Y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq 2 \\ 0 & \text{if } y_{in} < 2 \end{cases}$$

Example -3

Implement AND NOT function using McCulloch-Pitts neuron

Truth Table

x_1	x_2	$y\ (x_1, \bar{x}_2)$
0	0	0
0	1	0
1	0	1
1	1	0

Case 1: Assume both weights w_1 and w_2 are excitatory.

$w_1 = w_2 = 1$

x_1	x_2	y_{in}/y_{sum}	Y	
0	0	0	0	
0	1	1	0	→ Conflict
1	0	1	1	
1	1	2	0	

Case 2: Assume one weight as excitatory and the other as inhibitory

$w_1 = 1 \quad w_2 = -1$

x_1	x_2	y_{in} / y_{sum}	y_{out}
0	0	0	0
0	1	−1	0
1	0	1	1
1	1	0	0

$$\theta \geq nw - p$$
$$\theta \geq 2 \times 1 - 1$$
$$\theta \geq 1$$

Thus, the output of neuron y can be written as

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq 1 \\ 0 & \text{if } y_{in} < 1 \end{cases}$$

Example-4

Implement XOR function using McCulloch–Pitts neuron

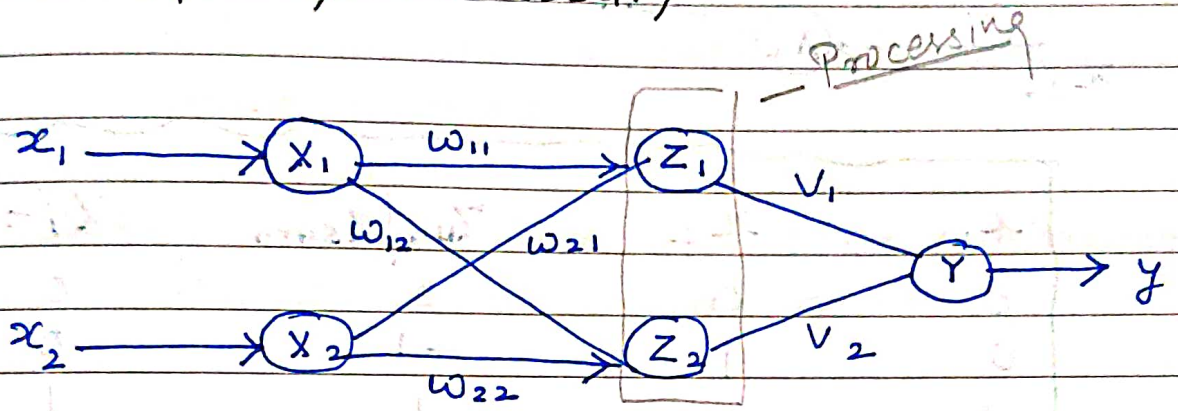
Table :

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

$$y = x_1 \bar{x}_2 + \bar{x}_1 x_2$$
$$y = z_1 + z_2$$

Where $z_1 = x_1 \bar{x}_2$ (function 1)
$z_2 = \bar{x}_1 x_2$ (function 2)
$y = z_1$ (OR) z_2 (function 3)

Note: A single layer net is not sufficient to represent the function. An intermediate layer is necessary.



First function ($z = x_1 \bar{x}_2$)

x_1	x_2	z_1
0	0	0
0	1	0
1	0	1
1	1	0

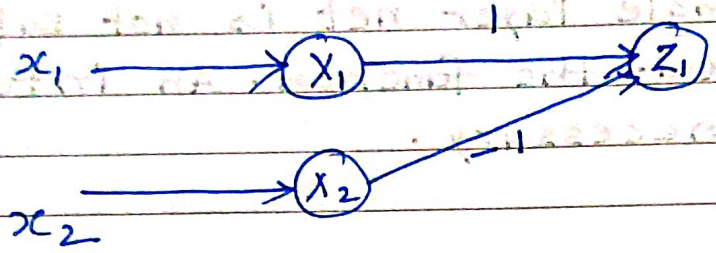
Case 1: Assume both weights as excitatory i.e. $w_{11} = w_{21} = 1$

x_1	x_2	z_{in}/z_{sum}	$z_{out} = z_1$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	2	0

It is not possible to obtain function z_1 using these weights.

Case 2 : Assume one weight as excitatory and other as inhibitory i.e.

$$w_{11} = 1 \quad w_{21} = -1$$



$$Z_{in} = w_{11}x_1 + w_{21}x_2$$

x_1	x_2	Z_{in}/Z_{sum}	$Z_1 = Z_{out}$
0	0	0	0
0	1	-1	0
1	0	1	1
1	1	0	0

$\theta \geq 1$ for Z_1 neuron

Second function $(Z_2 = \bar{x}_1 \bar{x}_2) = \bar{x}_1 x_2$

x_1	x_2	Z_2
0	0	0
0	1	1
1	0	0
1	1	0

Case 1 : Assume both weights as excitatory

$$w_{12} = w_{22} = 1$$

x_1	x_2	z_{2in}	$z_2 = z_{out}$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	2	0

Not possible to obtain function z_2 using these weights

Case 2 : Assume one weight as excitatory and the other as inhibitory i.e.

$$w_{12} = -1 \quad w_{22} = 1$$

x_1	x_2	z_{2in}	$z_2 = z_{out}$
0	0	0	0
0	1	1	1
1	0	-1	0
1	1	0	0

$\theta \geq 1$ for z_2 neuron

~~Case 3 :~~

Third function ($y = z_1 z_2$)

x_1	x_2	z_1	z_2	y
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

Net input is calculated using

$$y_{in} = z_1 v_1 + z_2 v_2$$

Case 1: Assume both weights as excitatory

i.e $v_1 = v_2 = 1$

x_1	x_2	z_1	z_2	y_{in}	y
0	0	0	0	0	0
0	1	0	1	1	1
1	0	1	0	1	1
1	1	0	0	0	0

$\theta \geq 1$

$v_1 = v_2 = 1$

$w_{11} = w_{22} = 1$

$w_{12} = -1$

$w_{21} = -1$

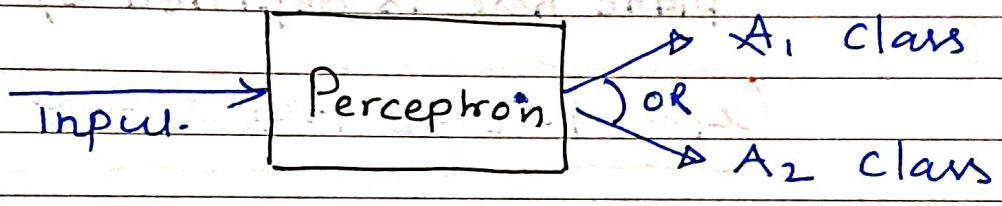
— for X-OR

Perceptron Network
(Single Layer feed forward Network)

The operation of Rosenblatt's perceptron is based on the MC neuron model. The model Consists of a linear combiner followed by hard limiter (Threshold based activati

Linear Combiner + Hard limiter

Aim of perceptron is to classify inputs $x_1, x_2, x_3 - - - x_n$

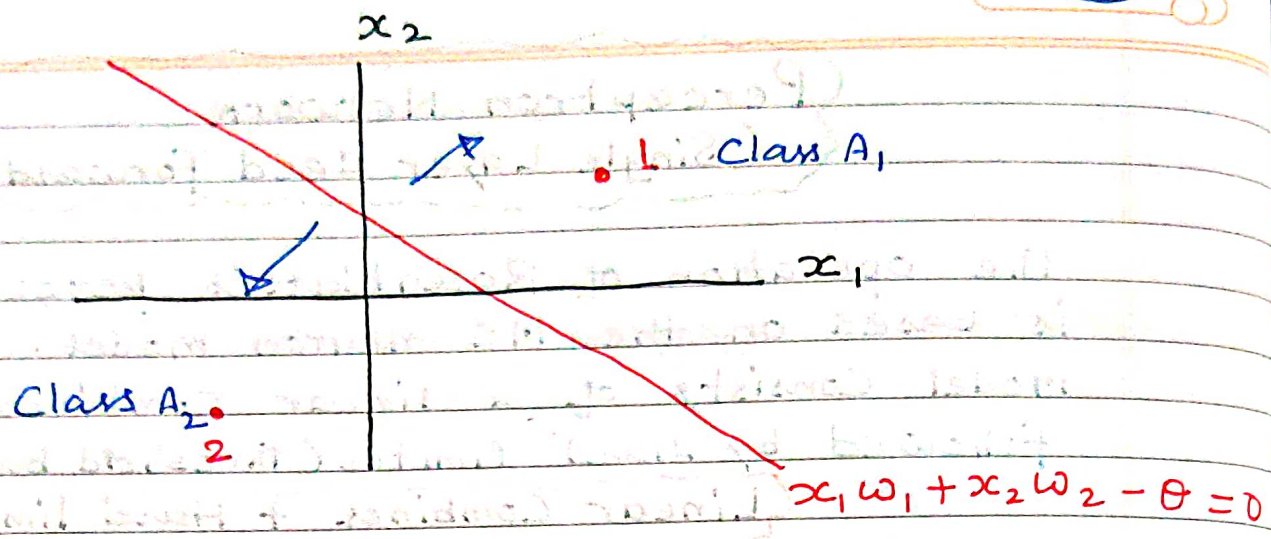


Note: n dimensional space is divided by a hyperplane into two decision regions.

Hyperplase is defined by the linearly Separable function

$$\sum_{i=1}^{n} x_i w_i - \Theta = 0$$

Note: For the case of two inputs x_1 and x_2, the decision boundary takes the form of straight line



Note: With three inputs x_1, x_2, x_3 (three dimension), the separating plane here is defined by the equation.

$$x_1w_1 + x_2w_2 + x_3w_3 - \theta = 0$$

Perceptron learning for classification.

Learning means making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron.

Step 1: Initialize random weights usually in the range $[-0.5, 0.5]$

↓

update the weights iteratively to obtain the output consistent with training examples.

Step 2: At iteration p, the actual output-

Actual o/p = $Y(p)$

Desired o/p = $Y_d(p)$

error = $Y_d(p) - Y(p)$

Note: iteration p here refers to the pth training example.

$$e(p) = Y_d(p) - Y(p)$$

if $e(p)$ is positive → We need to increase $Y(p)$

if $e(p)$ is Negative → We need to decrease $Y(p)$

[Each perceptron input- Contributes $x_i(p) \times w_i(p)$ to the total input- $X(p)$.]

Case 1: if $x_i(p) \geq 0$ (Positive) then increase in $w_i(p)$ → increase $Y(p)$

Case 2: if $x_i(p) < 0$ (Negative) then increase in $w_i(p)$ → decrease $Y(p)$

Perceptron Learning Rule:

$$W_i(p+1) = w_i(p) + \alpha \times x_i(p) \times e(p)$$

α → Learning Rate < 1

$$Y(P) = \text{step}\left[\sum_{i=1}^{n} x_i(P) \cdot \omega_i(P) - \theta\right]$$

$Y_d(P)$ = Desired o/p

$$\omega_i(P+1) = \omega_i(P) + \Delta\omega_i(P)$$

$$\Delta\omega_i(P) = \alpha \times x_i(P) \times e(P)$$

Example of perceptron learning : the logical operation AND

Threshold $\theta = 0.2$
Learning rate $\alpha = 0.1$

Initial random weight :
$\omega_1 = 0.3 \quad \omega_2 = -0.1$

After updation of fifth epoch, weight updation stopped.

Final weight -
$\omega_1 = 0.1 \quad \omega_2 = 0.1$

Separating function or separating line

Epoch	Inputs x_1	Inputs x_2	Desired output Y_d	Initial weights w_1	Initial weights w_2	Actual output Y	Error e	Final weights w_1	Final weights w_2
1	0	0	0	0.3	−0.1	0	0	0.3	−0.1
	0	1	0	0.3	−0.1	0	0	0.3	−0.1
	1	0	0	0.3	−0.1	1	−1	0.2	−0.1
	1	1	1	0.2	−0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	−1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	−1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	−1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1

Threshold: $\theta = 0.2$; learning rate: $\alpha = 0.1$.

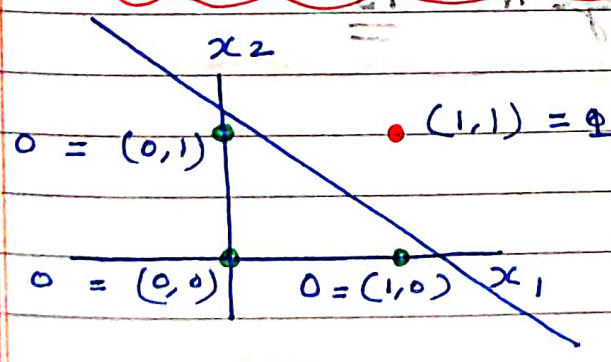
$$x_1 w_1 + x_2 w_2 = \theta$$

$$0.1 x_1 + 0.1 x_2 = 0.2$$

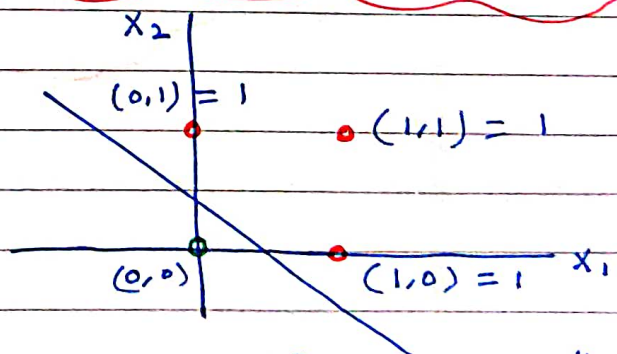
or

$$x_1 + x_2 = 2$$

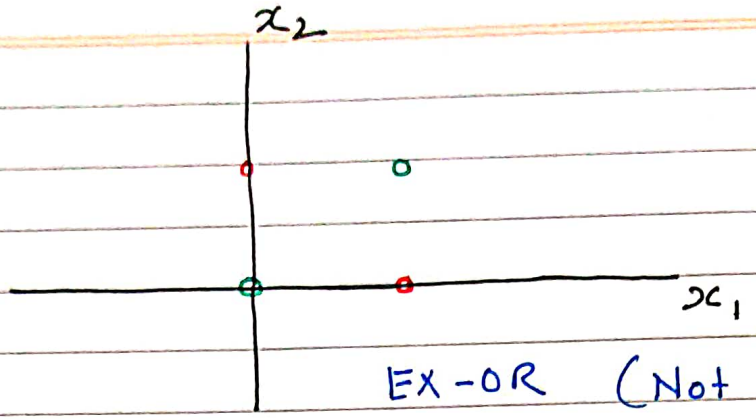
Two dimensional plot of basic logical operations



AND



OR — Homework



EX-OR (Not linearly separable)

Single layer perceptron can classify only linearly separable patterns.

Multi-Layer perceptron (MLP)

Read from PPT

Neural Network from scratch

What is Neuron

Numerical Exp - 10 → S.N Deepa
Page No. 93

Gradient Descent Algorithm

→ Towards Data Science

Gradient Descent is an iterative first-order optimization algorithm

↳ used to find a local minimum/maximum of a given function.

Note:- GD is used to minimise a cost/loss function (e.g. in Linear Regression)

GD is not for all function.

Two specific requirements:

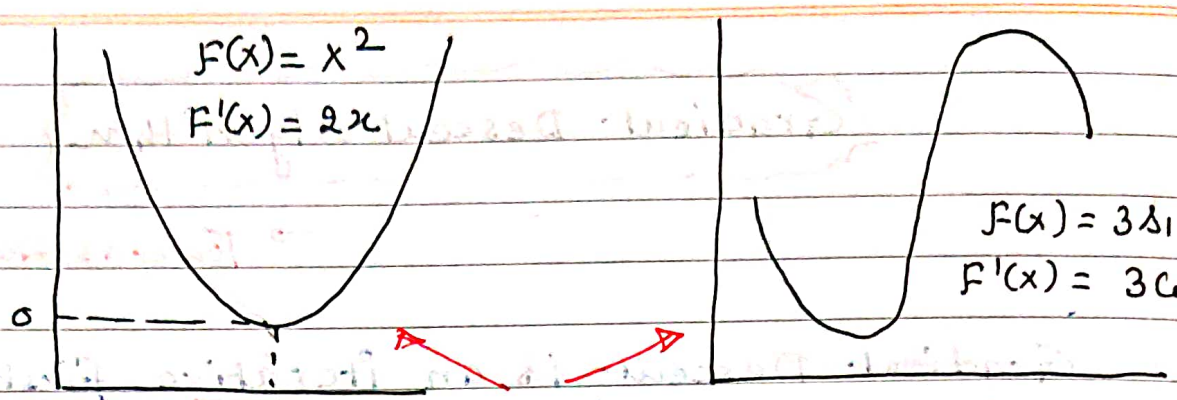
↳ Differentiable
↳ Convex

Differentiable →

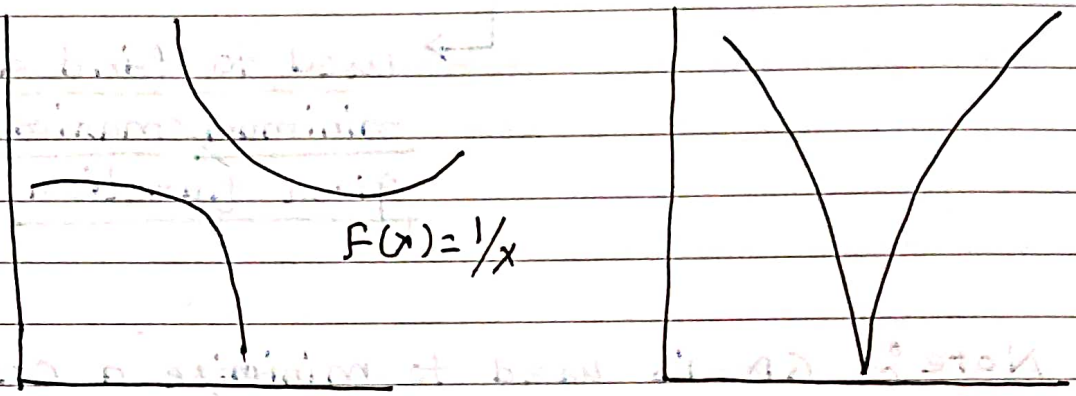
↳ A function is said to be differentiable if the derivatives of the function exists at all points in its domain.

↳ A differentiable function does not have any break, cusp, or angle.

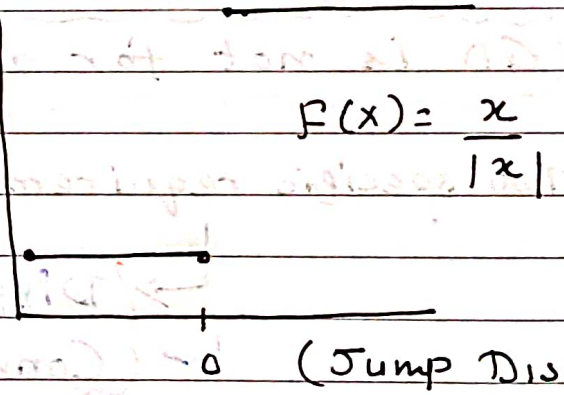
↳ A differentiable function is always continuous but every continuous function is not differentiable.



→ Differentiable ←



Non Differentiable



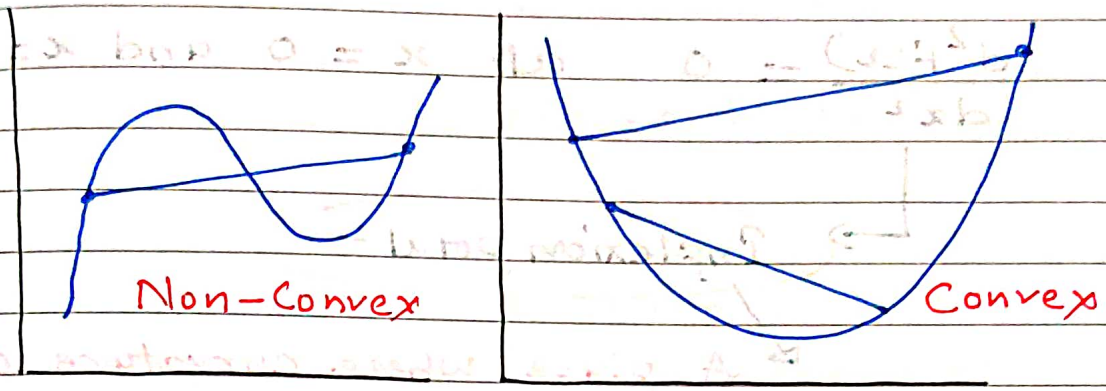
Convexity

If second derivative of a Univariate function is bigger than 0.

$$\frac{d^2 f(x)}{dx^2} > 0$$

Note →!

For a univariate function, if a line segment connecting two function's point lays on or above its curve (it does not cross it)



Example

$$f(x) = x^2 - x + 3$$

$$\frac{df(x)}{dx} = 2x - 1$$

$$\frac{d^2f(x)}{dx} = 2 > 0$$

Second derivative + ve

Strictly Convex

Example

$$f(x) = x^4 - 2x^3 + 2$$

$$\frac{df(x)}{dx} = 4x^3 - 6x^2 = x^2(4x - 6)$$

$$\frac{df(x)}{dx} = 0 \quad \text{at} \quad x = 0 \quad \text{and} \quad x = 1.5$$

Candidates for minima or maxima

Second derivative

$$\frac{d^2F(x)}{dx^2} = 12x^2 - 12x$$
$$= 12x(x-1)$$

$$\frac{d^2F(x)}{dx^2} = 0 \quad \text{at} \quad x = 0 \text{ and } x = 1$$

→ Inflexion point -

↘ A place where curvature changes sign.

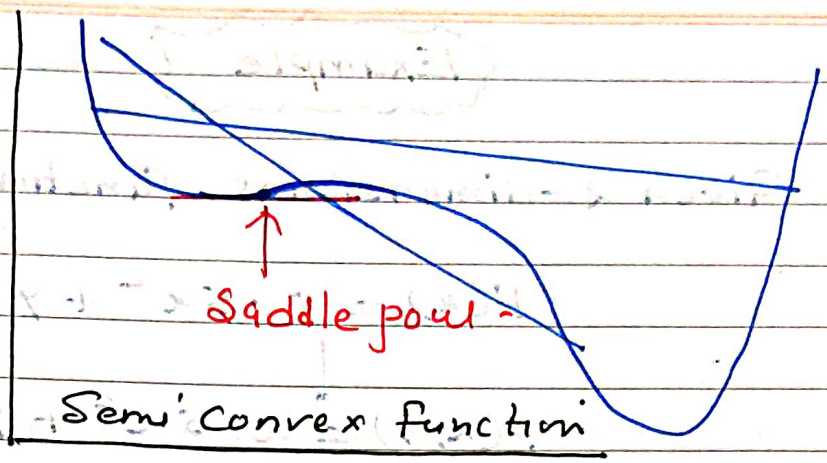
↘ It changes from convex to Concave

for $x < 0$	function is Convex
for $0 < x < 1$	function is Concave
for $x > 1$	function is Convex

$x = 0$ has both first & second derivative equal to zero meaning this is a saddle point.

$x = 1.5$ is a global minimum

↘ Algorithm stuck at saddle point.



Gradient -

Gradient is a slope of a curve at a given point in a specified direction.

For Univariate function

→ first derivate at a point.

for multivariate function

It is a vector of derivatives in each main direction (along variable axes)

Given n - dimensional function $f(x)$ at given point P.

$$\nabla f(P) = \begin{bmatrix} \frac{\partial f(P)}{\partial x_1} \\ \frac{\partial f(P)}{\partial x_2} \\ \vdots \\ \frac{\partial f(P)}{\partial x_n} \end{bmatrix}$$

Example

Given 2-dimensional function

$$f(x) = 0.5x^2 + y^2$$

$$\downarrow$$

$$f(x,y) = 0.5x^2 + y^2$$

find gradient at point P(10,10)

$$\frac{\partial f(x,y)}{\partial x} = x$$

$$\frac{\partial f(x,y)}{\partial y} = 2y$$

$$\nabla f(x,y) = \begin{bmatrix} x \\ 2y \end{bmatrix}$$

$$\nabla f(10,10) = \begin{bmatrix} 10 \\ 20 \end{bmatrix}$$

Slope is twice steeper along the y-axis.

Gradient Descent Algorithm

Gradient Descent Algorithm iteratively calculates the next point using Gradient at the current position.

$$P_{n+1} = P_n - \eta \nabla F(P_n)$$

P_{n+1} = Next point -
P_n = Current point -
η = Learning rate or step size for scaling the gradient -.
∇ = Gradient -

Note

Smaller η learning rate the longer Convergence
Larger η learning rate may not converge to optimal point. (Jump around)

Types of Gradient Descent -

- Batch GD
- Mini-Batch GD
- Stochastic GD
- Stochastic GD with Momentum.

↳ GD is used to find model parameter (w, b)

↳ If number of examples are sufficient -
↓
Model parameter get effectively tuned
↓
Hence Error get reduced.

If volume of data is very large

Each iteration takes hours or day

Entire data should be loaded in memory for each iteration.

This approach of entire data is "Batch Gradient Descent"

Mini-Batch Gradient Descent -

Entire training data is split into smaller set of training records.

e.g → Training data set = 320 records
5 mini batch
Each Mini Batch = 64 records.

If Mini Batch = 1 record

Stochastic Gradient Descent -

Noted Point - 1

SGD works based on individual records. Calculates the error & update the parameters (w, b) based on each training record.

Noted point - 2

BGD calculates the error based on each training record
updates the parameters after evaluating all training record.

Noted point - 3

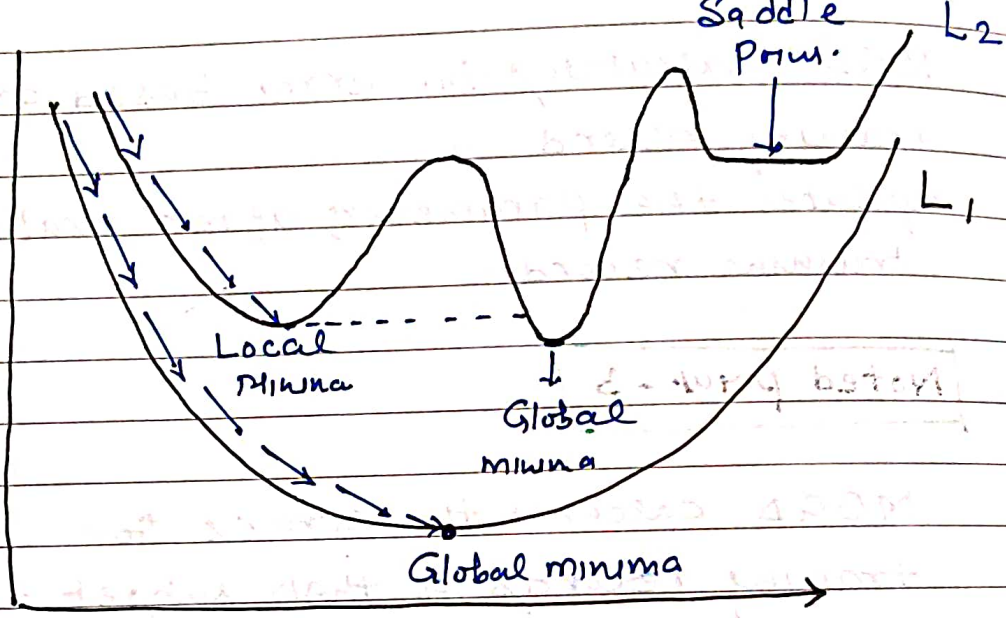
MBGD calcules the errors for individual training records of that subset.
updates the parameters after evaluating all training record of that subset.

Approach	Error calculation	Parameter update
BGD	Each training record	After all records
MBGD	Each record of batch	After all records of batch
SGD	Each training record	Each record

Challenges in Batch Gradient Descent-

→ Loading the entire training records in memory

→ Gradient descent may stuck at local minima in the case of non convex loss function.



Stochastic Gradient Descent-

In case of stochastic gradient descent, the possibility to get trapped in local minima is much less.

↳ For each training record t_1, t_2, t_3, etc., there is a separate loss curve L_1, L_2, L_3, etc.

↳ This leads to higher probability to reach global minima.

↳ One downside of SGD is that the model parameters update is so frequent that it is computationally expensive.

↳ Takes higher training time for larger training datasets.

Effect of Learning rate

Case 1 : If the learning rate is large

↳ Gradient descent happens very fast.

↳ Possibility of overshooting the minima.

Case 2 : If the learning rate is small

↳ Training process may take long time.

↳ Algorithm designed for specific number of iterations, there is a possibility of missing out the minima.

Note :

In reality gradient descent never reach the minima.

↳ It reaches to the point close to minima

↳ where loss value does not change much

Algorithm should run for a specific number of iterations (called epoch) but stop when error change stop.

Other Optimization Algorithms

1) Gradient Descent with Momentum
2) Adagrad
3) Adadelta
4) RMSprop
5) Adam

Gradient Descent with Momentum

↳ It is designed to accelerate the optimization process.

↳ Momentum is designed to accelerate learning especially in the face of high curvature

↳ Momentum is helpful when search space is flat or nearly flat. (Zero gradient)

↳ Gradient changes a lot over small regions

In Physics, acceleration in a direction can be accumulated from past updates. (History or Momentum)

In Normal gradient Descent -

$$W_{new} = W_{old} - \alpha \cdot \partial W$$

For gradient descent with Momentum, the weight update in the i^{th} iteration is done as follows :

$$m_{wi} = \beta_1 m_{wi-1} + (1-\beta_1)\partial w$$

$$W_i = W_{i-1} - \alpha \cdot m_{wi}$$

Similarly, the bias update in the i^{th} iteration

$$m_{bi} = \beta_1 \cdot m_{wi-1} + (1-\beta_1)\partial w$$

$$b_i = b_{i-1} - \alpha \cdot m_{bi}$$

Dampens the vertical oscillation problem
Standard value $\beta_1 = 0.9$

Adaptive Gradient (Adagrad)

↳ Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function.

↳ A limitation of gradient descent is that it uses the same step size (Learning rate) for each input variable.

↓

This can be a problem for objective functions that have different amount of curvature in different dimensions.

AdaGrad allows the step size in each dimension used by the optimization algorithm to be automatically adapted based on gradient seen for the variable seen over the course of the search.

Problem → Learning rate (step size) remains same in all iteration & same in all direction → Solution — AdaGrad

Assumptions

↳ During the initial iterations of GD, the learning rate should be higher.

↳ closer to minima or after some iterations learning rate needs to be reduced.

AdaGrad Algo → Adaptive learning rate method based on an accumulated value of historical gradient.

" Learning rate is scaled by the inverse of the square root of sum of squares of the historical gradient of the parameters "

The weight update for the ith iteration is done as follows:

$$\alpha_i = \frac{\alpha}{\sqrt{\delta + \sum_{K=1}^{i-1} \partial w_K^2}}$$

- α_i represent the learning rate to be used in the ith iteration.
- $\delta \approx 10^{-5} - 10^{-7}$ very small number used to prevent division by zero.

$$w_i^\circ = w_{i-1} - \alpha_i \cdot \partial w$$

Similarly, the bias update is done as follows.

$$\alpha_i^\circ = \frac{\alpha}{\sqrt{\delta + \sum_{K=1}^{i-1} \partial b_K^2}}$$

$$b_i = b_{i-1} - \alpha_i \cdot \partial b$$

Major problem Over the iteration $\sum \partial w^2$ becomes bigger & bigger so learning rate keeps decaying.

→ Algorithm may stuck.

Adadelta Algorithm

Instead of taking the squared gradients of all past iteration, it takes the gradients for a specific window or number of iteration.

↳ If window size = 3, it will take the squared gradients only for 3 preceding iteration.

Unpublished

Root Mean Square Propagation (RMSProp)

↳ The RMSProp algorithm was developed by Geoffrey Hinton.

↳ Tries to address the learning rate decay problem of Adagrad algorithm.

↳ In Adagrad, learning rate for neurons having high variance decays even more drastically.

↳

↳ RMSProp algorithm does not take a flat cumulative value of the past gradients.

↳ It uses a decay factor β_2 which helps to give more weightage to the recent gradients & less weightage to the older gradients.

→ The value β_2 is adopted is 0.999

for RMSProp algorithm, the weight update for the i^{th} iteration.

$$S\omega_i = \beta_2 S\omega_{i-1} + (1-\beta_2) \cdot \partial w_i^2$$

$$\alpha_i = \frac{\alpha}{\sqrt{\delta + S\omega_i}}$$

$$W_{i_{new}} = W_{i_{old}} - \alpha_i \cdot \partial W$$

The bias update for the i^{th} iteration is done as follows:

$$Sb_i = \beta_2 \cdot S_{b_{i-1}} + (1-\beta_2) \cdot \partial b_i^2$$

$$\alpha_i = \frac{\alpha}{\sqrt{\delta + S_{b_i}}}$$

$$b_i = b_i - \alpha_i \cdot \partial b$$

Adaptive Moment Estimation (Adam)

Adam = RMSProp + SGD with momentum

→ In GD with momentum, the parameter update in each iteration is done as a combination of parameter update in previous iteration + Gradient in current iteration.

The momentum term is calculated as follows :

$$m_{wi} = \beta_1 \cdot m_{wi-1} + (1-\beta_1) \cdot \partial w$$

Similarly in RMSProp, the learning rate is gradually adjusted.

$$S_{wi} = \beta_2 \cdot S_{wi-1} + (1-\beta_2) \cdot \partial W_i^2$$

$$\alpha_i = \frac{\alpha}{\sqrt{\delta + S_{wi}}}$$

Combining the effect of momentum + learning rate adjustment..

$$W_i = W_{i-1} - \alpha_i \cdot m_{w_i}$$

$$\downarrow$$

$$W_i = W_{i-1} - \alpha_i \cdot m_{w_i}$$

Note : When we start

m & s $\longrightarrow$ small close to zero

$\downarrow$

Down the iteration, m & S values grow

$\downarrow$

To Dampen the effect of growing value of m & s.

Adjust the values of m and s

$$m_{\omega_i}^{corrected} = \frac{m_{\omega_i}}{1 - \beta_1^i}$$ ✓

$$S_{\omega_i}^{Corrected} = \frac{S_{\omega_i}}{1 - \beta_2^i}$$ ✓

$$\alpha_i^{corrected} = \frac{\alpha}{\sqrt{\delta + S_{\omega_i}^{corrected}}}$$

$$W_i = W_{i-1} - \alpha_i^{corrected} \cdot m_{\omega_i}^{corrected}$$

Similarly for the bias term

$$m_{b_i} = \beta_1 \cdot m_{b_{i-1}} + (1 - \beta_1) \cdot \partial b$$

$$S_{b_i} = \beta_2 \cdot S_{b_{i-1}} + (1 - \beta_2) \cdot \partial b_i^2$$

$$\alpha_i = \frac{\alpha}{\sqrt{\delta + S_{b_i}}}$$

$$b_i = b_{i-1} - \alpha_i \cdot m_{b_i}$$

$$m_{b_i}^{\text{Corrected}} = \frac{m_{b_i}}{1 - \beta_1 i^0}$$

$$s_{b_i}^{\text{Corrected}} = \frac{s_{b_i}}{1 - \beta_2 i}$$

$$\alpha_i^{\text{Corrected}} = \frac{\alpha}{\sqrt{\delta + s_{b_i}^{\text{Corrected}}}}$$

$$b_i = b_{i-1} - \alpha_i^{\text{Corrected}} \cdot m_{b_i}^{\text{Corrected}}$$

Loss Functions

Loss functions can be classified into two major categories

- Regression Loss
- Classification Loss

To predict output in terms of Continuous value.

To predict output in terms of class. (categorical)

Exp: Given large set of hand written images, categorize them into one of 0-9 digits. — Classification ✓

Exp: Predict the price of room for given floor area, size of room etc.
↳ Regression ✓

Regression Loss

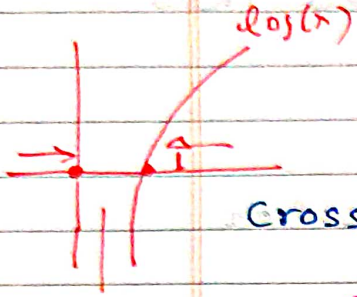
Mean Square Error / Quadratic Loss / L2 Loss

Mean square error is measured as the average of squared difference between predictions and actual observation.

$$MSE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}$$

Classification Loss

Cross entropy loss

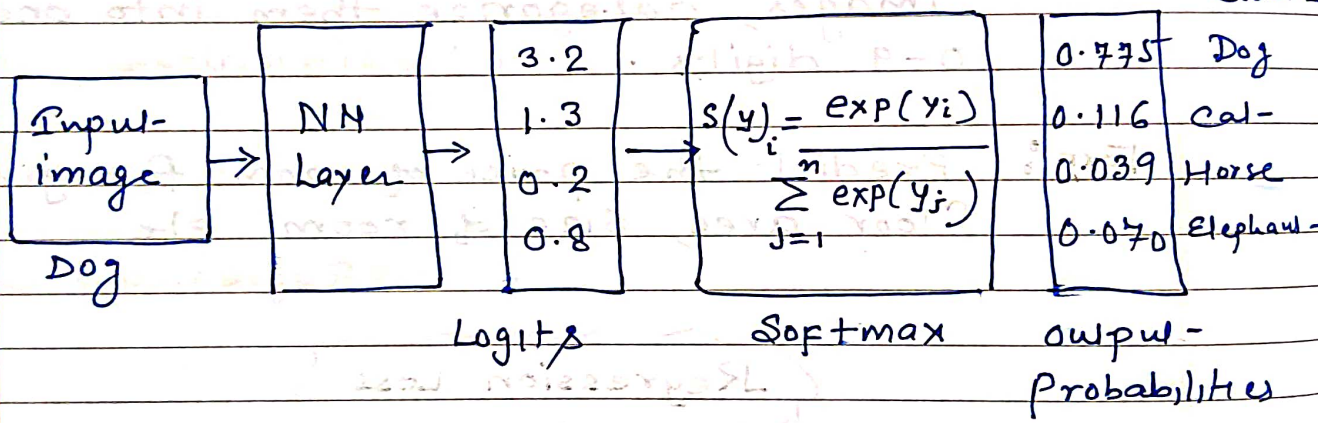


Cross entropy Loss = L_{CE}

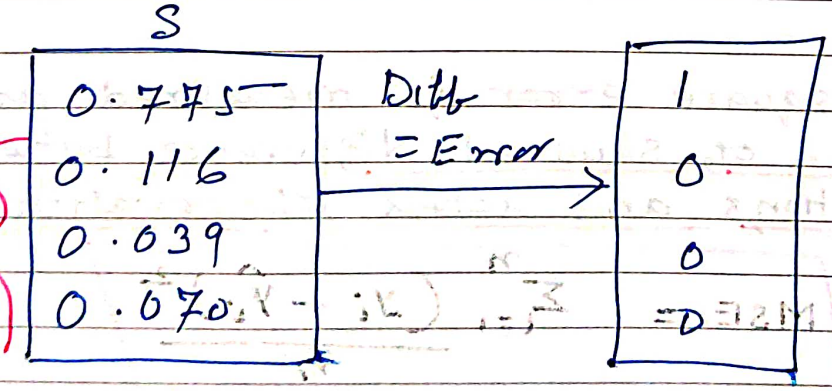
Negative sign
$\log(P(x)) < 0$
for $P(x)$ in $(0,1)$

$$L_{CE} = -\sum_{i=1}^{n} t_i \log(p_i) \text{ for } n \text{ classes}$$

Where t_i is the truth label
p_i is the softmax probability for the i^{th} class.



Softmax converts Logits ⟶ Probabilities



Cross Entropy Does not depend on incorrect

$$L_{CE} = -\sum_{i=1} t_i \log(P_i)$$

$$= -[1 \cdot \log_2(0.775) + 0 \log_2(0.126) + 0 \cdot \log_2(0.070)]$$

$$= -\log_2(0.775)$$

$$\boxed{L_{CE} = 0.3677}$$

Softmax function is continuously differentiable function. This makes it possible to calculate the derivatives of the loss function with respect to every weight.

Assume that after some iterations of model training, the model outputs the following vector of logits.

S	T
0.938	1
0.028	0
0.013	0
0.023	0

$$L_{CE} = -1 \log_2(0.938) + 0 + 0 + 0$$

$$= 0.095$$

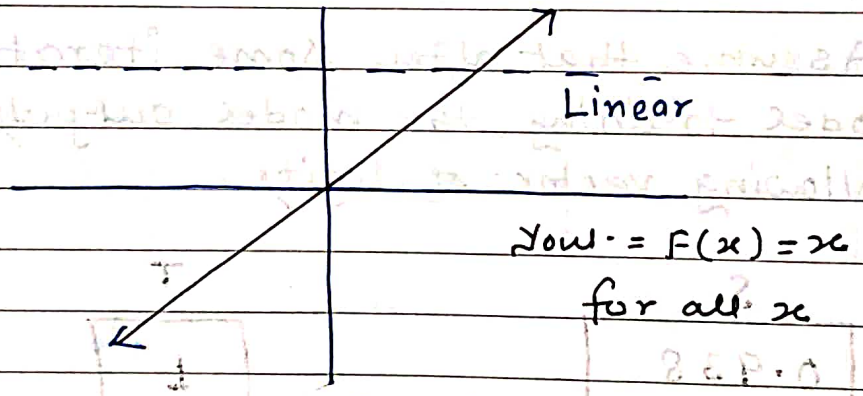
0.095 is less than Previous loss.

Activation function

Activation functions are responsible for transforming the summation of weighted inputs into an activation which is given as the output of a neural network.

- Linear function
- Non-Linear function.

Linear function (Identity function)



However linear function is differentiable but has a constant derivative = 1

→ Gradient is unaffected by the value of the input.

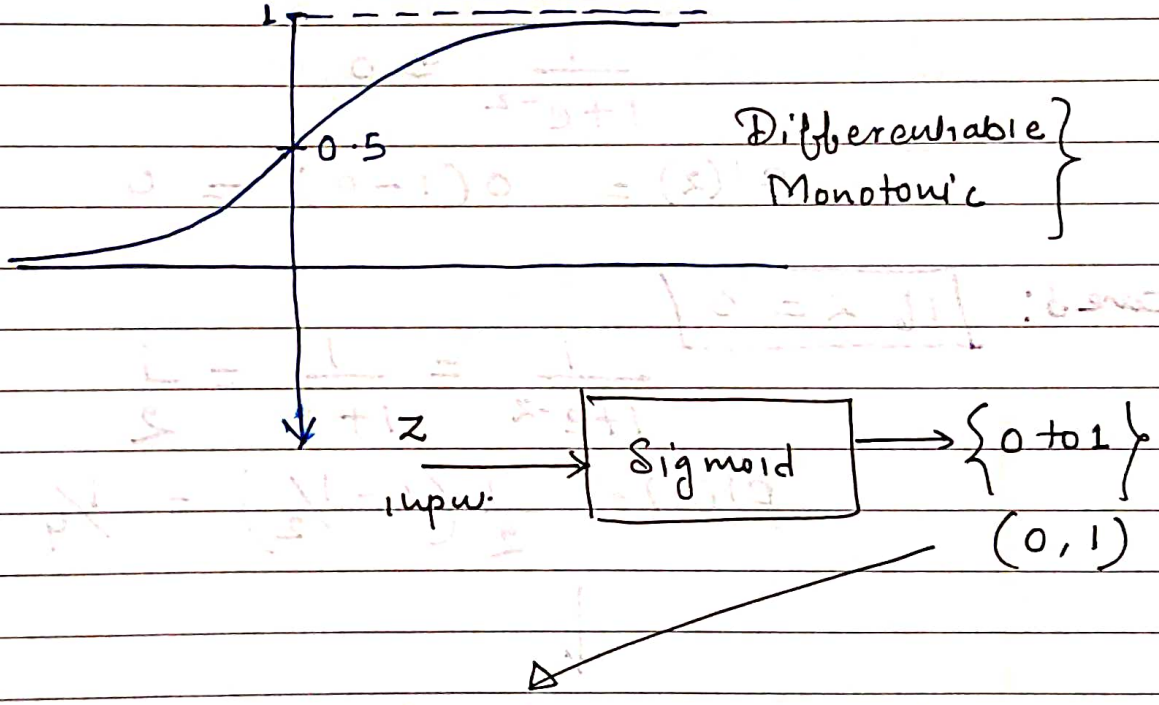
Note According to the universal approximation theorem, with a non-linear activation function, a two-layer neural network can approximate any complex non-linear data pattern

Non-Linear Activation function

→ Sigmoid or Logistic
→ Hyperbolic tangent or tanh function
→ ReLU (Rectified Linear Unit)
→ Leaky ReLU

Sigmoid

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



Suitable for Probability based prediction.

Problem!

Derivative of the function is very low, almost close to zero when z is very large or very small.

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

$$F'(z) = \left(\frac{1}{1+e^{-z}}\right)\left(1 - \frac{1}{1+e^{-z}}\right)$$

Case 1°: **If z is very large**

$$e^{-z} \approx 0$$

$$f'(z) = 1 \cdot (1-1) = 0$$

Case 2: **If z is very small**

$$\frac{1}{1+e^{-z}} \approx 0$$

$$f'(z) = 0(1-0) = 0$$

Case 3: **If z = 0**

$$\frac{1}{1+e^{-z}} = \frac{1}{1+1} = \frac{1}{2}$$

$$f'(z) = \frac{1}{2}\left(1 - \frac{1}{2}\right) = \frac{1}{4}$$

↓

In most of the case, derivative is very small which makes **convergence slow**.

Problem 2: Non zero Centered o/p

Sigmoid o/p always between 0 and 1.

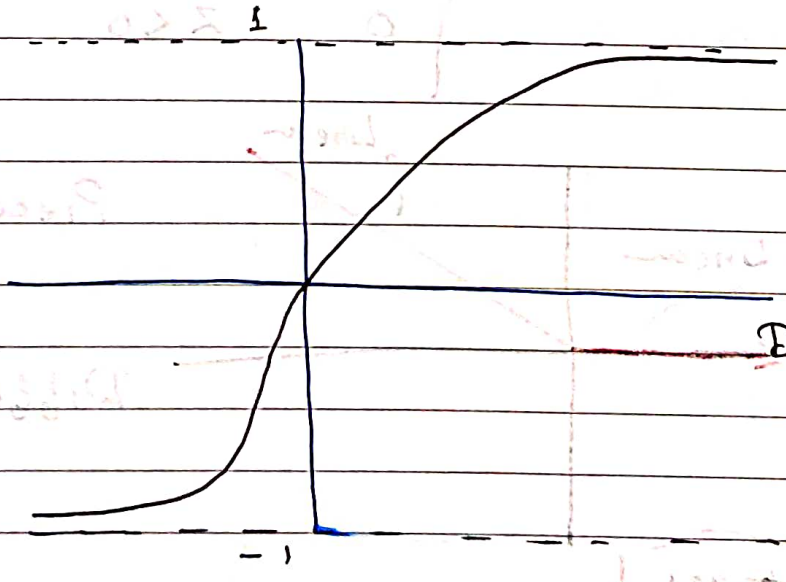
→ During gradient descent -

→ Gradient will be either positive or negative but could not change sign as activation output is always positive.

→ As a result, the gradient updates go too far in different directions which makes optimization harder

Hyperbolic Tangent (Tanh) function

$$\text{Your} = f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



(-1 to 1)
Centered to zero
Differentiable
Monotonic

Since the o/p is Centered around zero so it is preffered in hidden layer.

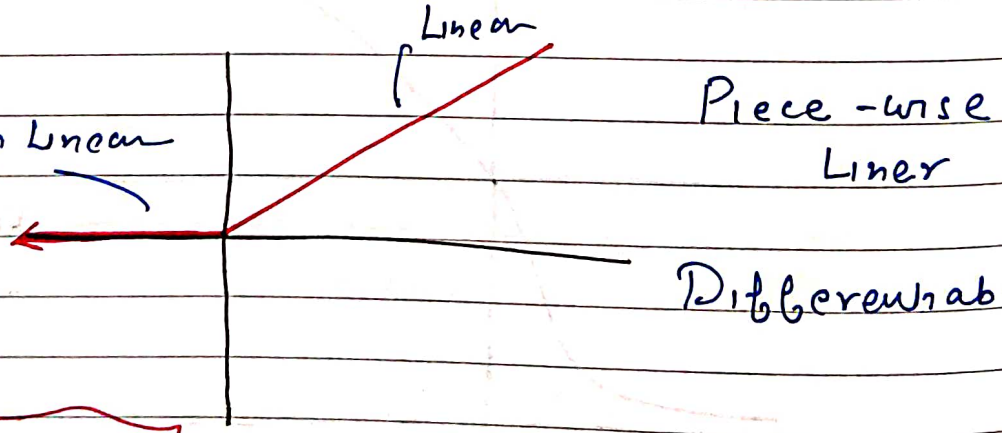
→ Sigmoidal function is preffered in o/p layer in binary classification
↳ 0 to 1 output.

Problem : Slow convergence due to low derivatives for very high and very low value of z.
↳ Bw. faster than Sigmoid

ReLU (Rectified Linear Unit)

→ Most popular activation function for hidden layer.

$$f(z) = \begin{cases} z, & z \geq 0 \\ 0 & z < 0 \end{cases}$$



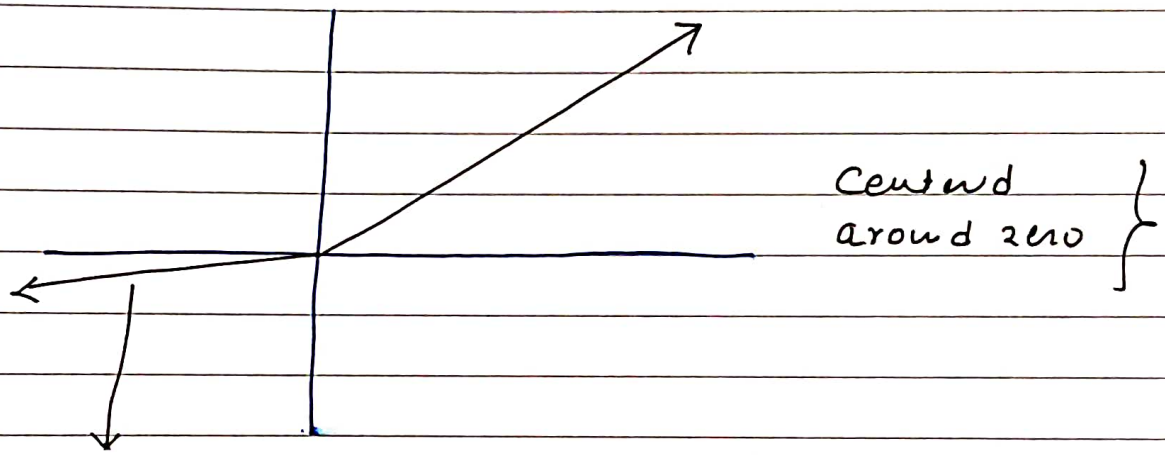
Advantages
→ Derivative is 1. Convergence is faster.

→ Computation is faster since due to characteristic of ReLU, there is a possibility that 50% of neurons to give 0 activations.

→ Less computation required
50% dying neuron.

Solution

Leaky ReLU function



Introdue small negative slope (like 0.01) instead of zero

→ Solve the problem of dead units of neuron.

Softmax function

→ for Multi-class classification.

Logits

y_1
y_2
y_3
$\vdots$
y_n

$$S(y_i) = \frac{\exp(y_i)}{\sum_{j=1}^{n} \exp(y_j)}$$

→ Probability vector