15/12/2025

data collection
↓
data preprocessing
↓
Build the AI model
- Training set
- Test set
- Validation set

70% - Training {Shuffled and fed to the AI model}
30%
- Test (15%)
- validation (15%)
↓
enable early stopping

performance measure - Accuracy
- confusion matrix

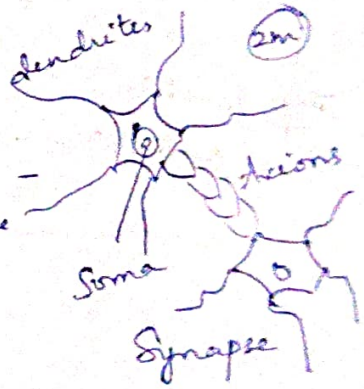
| | P | N |
|---|---|---|
| P | TP | FN |
| N | FP | TN |

$$\text{Accuracy} = \frac{TP + TN}{\underbrace{TP + TN + FP + FN}_{\text{Total}}}$$

## Introduction to the Neural Networks

3 types of neuron
- Sensory neuron
- Motoroy neuron
- Interconnection neuron

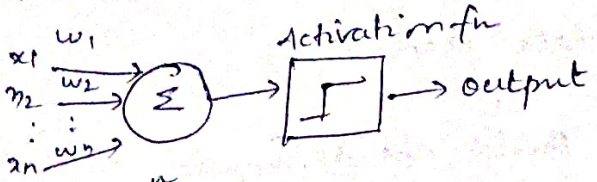
Biological neuron - structure




| Biological Neuron | Artificial Neuron |
|---|---|
| - call it as a neuron | - call it as a computational node |
| - Inputs processed through dendrites | - Inputs |
| - Soma - cell body | - processing centre |
| - Axion | - output |
| - Synapse | - connection or a junction that connect the output of one neuron to the input of the other neuron |
| - Complex | - less complex |
| - more connected and perform very less or complex tasks | - Less connected and perform limited tasks |

Mc Culloch + pitts neuron model

- mathematical model to denote artifical neuron




$$Sum = \sum_{i=1}^{n} x_i w_i$$

output = $\phi$(Sum) → Activation fn

Threshold based activation function

$$f(x) = \begin{cases} 0 & sum \le 0 \\ 1 & otherwise \end{cases}$$

$\phi$(sum)(or) f(sum)

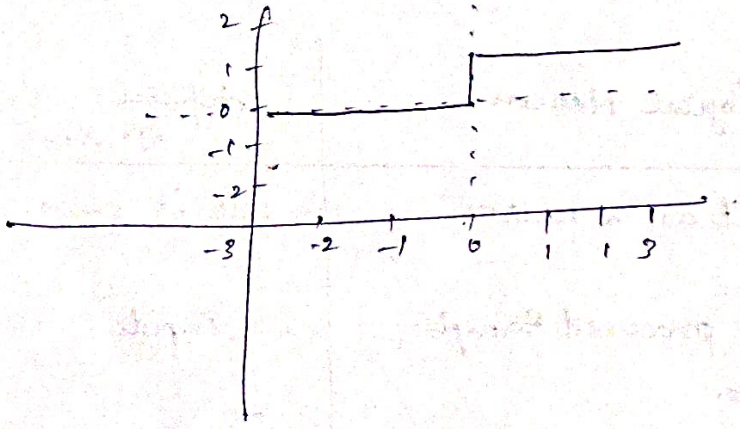
$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$x^T w = \underset{1\times 3}{[x_1 \; x_2 \; x_3]} \underset{3\times 1}{\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}}$$

$$= [x_1 w_1 + x_2 w_2 + x_3 w_3]_{1\times 1}$$

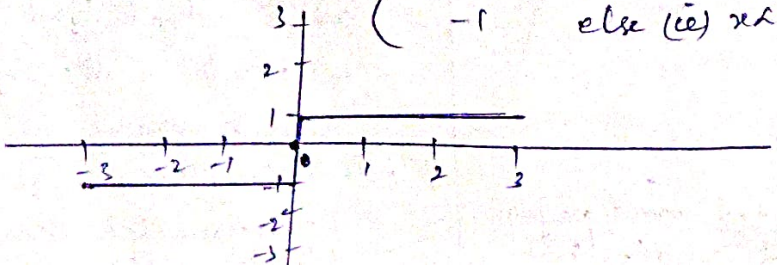
Step function:

$$out(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & otherwise \end{cases} \Big/ \begin{cases} 0 & x \le 0 \\ 1 & otherwise \end{cases}$$




Signum activation fn: (-1, 0, 1)

$$out(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{else (ie) } x < 0 \end{cases}$$



$x = [1 \quad 1 \quad 2]$

$w = [0.5 \quad 0.5 \quad 0.6]$

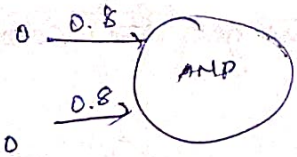
$\sum x_i w_i = 0.5 + 0.5 + 1.2 = 2.2 \geq 1$

$Out(x) = 1$

| $x_1$ | $x_2$ | $y$(AND) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$T = 0.5$

$T = 1$




$Out(0,0) = 0 \times 0.8 + 0 \times 0.8 = 0 \not\geq 1 = 0$

$Out(0,1) = 0 \times 0.8 + 1 \times 0.8 = 0.8 \not\geq 1 = 0$

$Out(1,0) = 1 \times 0.8 + 0 \times 0.8 = 0.8 \not\geq 1 = 0$

$Out(1,1) = 1 \times 0.8 + 1 \times 0.8 = 1.6 > 1 = 1$

---

16/12/2025

## Perceptron model

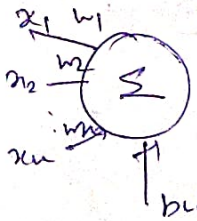
- binary classifier
- used to solve the linearly separable problems

### Steps - perceptron learning algo :

1. $\{(x_1, t_1), (x_2, t_2), (x_3, t_3) \ldots (x_n, t_n)\}$

   $c_1 = +1 \quad c_2 = -1$

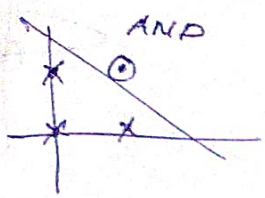
2. Initialize the weights randomly $\in [-1, 1]$
3. Initialize the learning rate $\in [0, 1] \Rightarrow \eta$
4. $(x_i, t_i)$ for each input, do the forward propagation




$Sum = \sum_{i=1}^{n} x_i w_i + bias$

$\Rightarrow Sum = \sum_{i=0}^{n} x_i w_i$

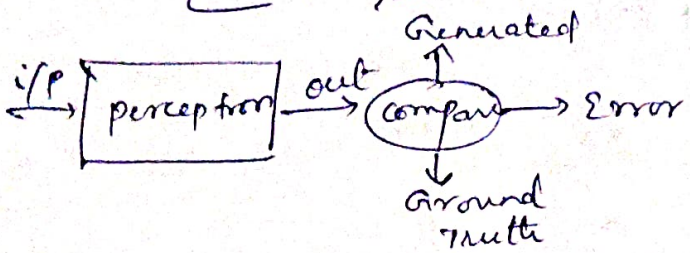
$x_0 = 1$



Then, the activation function must be applied → Signum

out = Sign(net)

(ie) $out = \begin{cases} 1 & \text{if } net \geq 0 \\ -1 & \text{otherwise} \end{cases}$




5) find the error: err = t - out

$w_i = w_i + \Delta w_i$

6) update: $\Delta w_i = error * \eta * x_i$

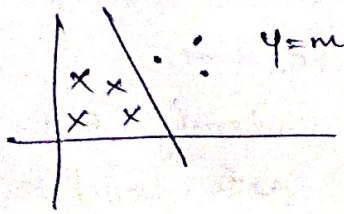
7) Repeat the above steps till we get the convinced error. (1-6)

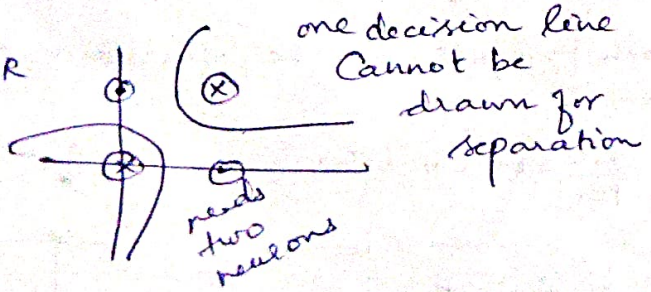
Eg

Err(8) = t - out (ground truth - predicted)

| | | | |
|---|---|---|---|
| t = 1 | out = 1 | err = 0 | no update |
| t = -1 | out = -1 | err = 0 | no update |
| t = 1 | out = -1 | err = 1+1 = 2 | update (increase weight) $\Delta w_i = error * \eta * x_i$ |
| t = -1 | out = 1 | err = -1+1 = -2 | update (decrease weight) |

Perceptron as a binary classifier:



Problem:

| | $i_1$ | $i_2$ | t |
|---|---|---|---|
| $x_1$ | 0 | 0 | -1 |
| $x_2$ | 0 | 4 | +1 |
| $x_3$ | 1 | 1 | +1 |
| $x_4$ | 1 | -3 | -1 |

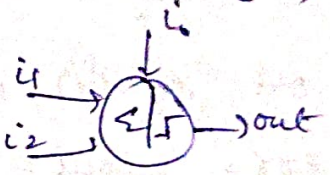
$w_0 = w_1 = w_2 = 1 ; \eta = 1$

$x_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ $x_2 = \begin{pmatrix} 0 \\ 4 \end{pmatrix}$ $x_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ $x_4 = \begin{pmatrix} 1 \\ -3 \end{pmatrix}$

Augument 1 to all i/ps:

~~$x_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ $x_2 = \begin{pmatrix} 0 \\ 4 \\ 1 \end{pmatrix}$ $x_3 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ $x_4 = \begin{pmatrix} 1 \\ -3 \\ 1 \end{pmatrix}$~~

~~$x_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ $x_2 = \begin{pmatrix} 1 \\ 4 \\ 0 \end{pmatrix}$~~




$x_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ $x_2 = \begin{pmatrix} 1 \\ 0 \\ 4 \end{pmatrix}$ $x_3 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ $x_4 = \begin{pmatrix} 1 \\ 1 \\ -3 \end{pmatrix}$

$x_p = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ $w = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$

$y_1 = w^T x_p = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

$= 1 \times 1 + 1 \times 0 + 1 \times 0$

$= 1$

$t = -1$ , $out = 1$

$err = -1 - 1 = -2$

$w_0 = w_0 + \Delta w_0$ ; $w_0 = (1\ 1\ 1) + 1 \times -2 \times (1\ 0\ 0)$

$w = w + \Delta w$ $= (1\ 1\ 1) + (-2\ 0\ 0)$

$= (-1\ 1\ 1)$

$x_2 = \begin{pmatrix} 1 \\ 0 \\ 4 \end{pmatrix}$ $w = \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}$

$$S_2 = (-1 \;\; 1 \;\; 1)\begin{pmatrix}1\\0\\4\end{pmatrix}$$

$$= -1+0+4$$

$$= 3$$

$t = +1$, $out = sign(3) = 1$

$err = t - out = 1-1 = 0$

So, no update in weight

$x_3 = \begin{pmatrix}1\\1\\1\end{pmatrix}$ $w = \begin{pmatrix}-1\\1\\1\end{pmatrix}$

$S_3 = w^T x_3$

$$= (-1 \;\; 1 \;\; 1)\begin{pmatrix}1\\1\\1\end{pmatrix}$$

$$= -1+1+1$$

$$= 1$$

$out = sign(1) = 1$

$err = t - out = 1-1 = 0$

So, no update in weight

$x_4 = \begin{pmatrix}1\\1\\-3\end{pmatrix}$ $w = \begin{pmatrix}-1\\1\\1\end{pmatrix}$

$S_4 = w^T x_4$

$$= (-1 \;\; 1 \;\; 1)\begin{pmatrix}1\\1\\-3\end{pmatrix}$$

$$= -1+1-3$$

$$= -4+1$$

$$= -3$$

$out = sign(-3) = 0$

$err = t - out = -1-0 = -1$

$w = w + \Delta w$

$w = \begin{pmatrix}-1\\1\\1\end{pmatrix}$

$out = sign(-3) = -1$

$err = t - out$

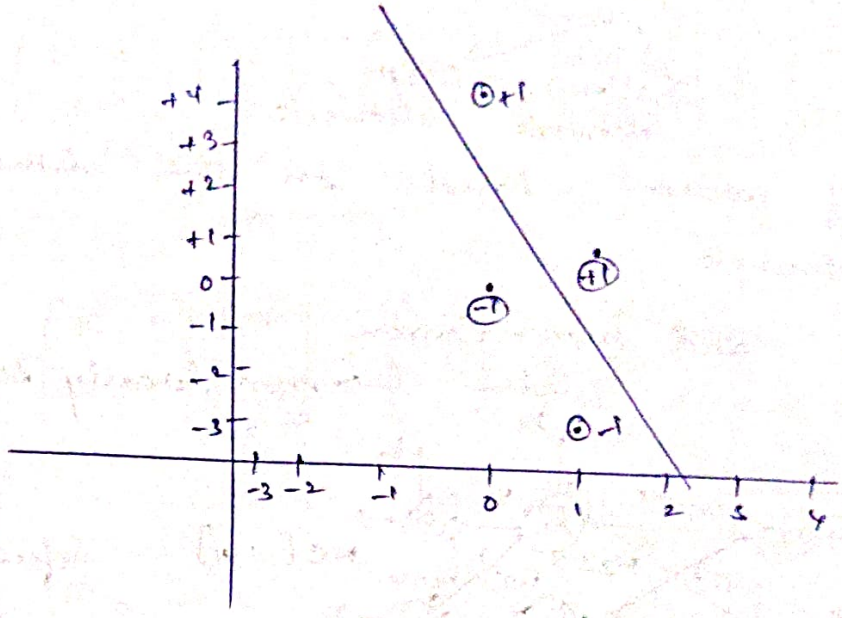
$err = -1-(-1)$

$err = 0$

Hence, no update

$\eta \to 1$, then the epoch converges quickly..

$\eta$ should be as small as possible

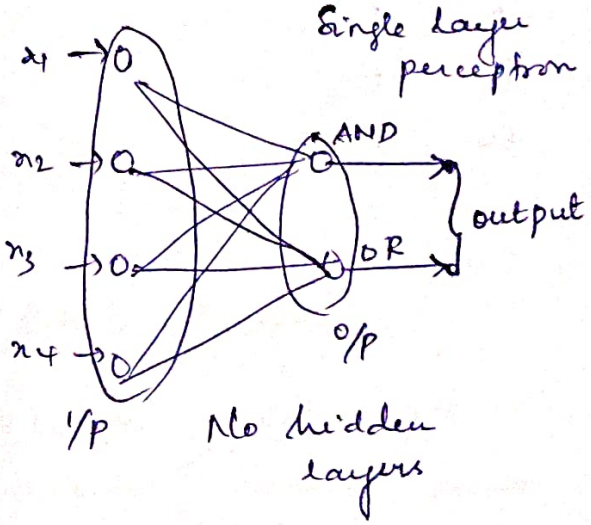



OR : $-1$ $1$ $1$ $1$ $\eta = 1$

$t_1$ $t_2$ $t_3$ $t_4$

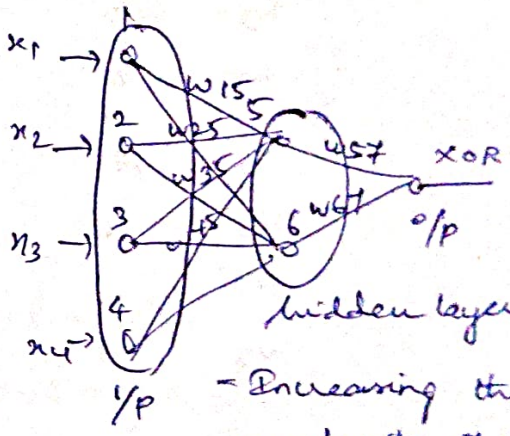
---

Single layer perceptron



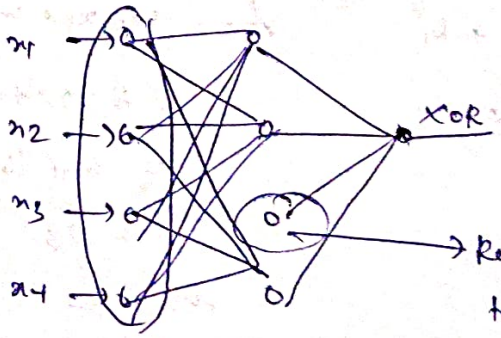

No hidden layers

Multi-layer Perceptron




hidden layer

- Increasing the complexity of the N/w




Regularization to prevent overfitting

one/two/more hidden layer

↓

Shallow network

more than two

↓

Deep Neural Networks

22/8/2022

MLP

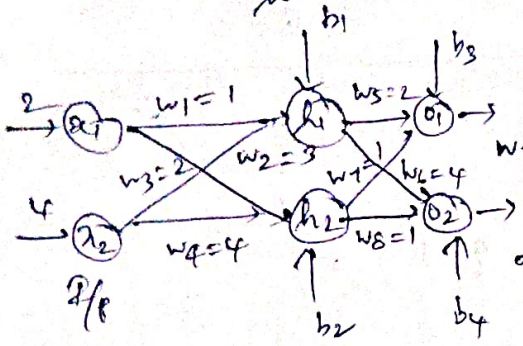
Shallow Network

Deep Network

No. of neurons - based on the dimensionality of the i/p sample

Convex and non-Convex

hidden → non-linearly separable pblms




$w \in [-1, +1]$

feed foward neural N/ws

Acyclic graph

| $w_i$ | value |
|---|---|
| $w_1$ | 1 |
| $w_2$ | 3 |
| $w_3$ | 2 |
| $w_4$ | 4 |
| $w_5$ | 2 |
| $w_6$ | 4 |
| $w_7$ | 1 |
| $w_8$ | 1 |

| $b_i$ | value |
|---|---|
| $b_1$ | 1 |
| $b_2$ | 1 |
| $b_3$ | 1 |
| $b_4$ | 1 |

Solution:

$h_1 = w_1x_1 + w_3x_2 + b_1$

$= 1 \times 2 + \underset{2\times4}{8} + 1$

$= 2 + 8 + 1$

$h_1 = 11$

$h_2 = w_2x_1 + w_4x_2 + b_2$

$= 3 \times 2 + 4 \times 4 + 1$

$= 6 + 16 + 1$

$= 23$

$h_2 = 23$

$O_1 = h_1w_5 + h_2w_7 + b_3$

$= 11 \times 2 + 23 \times 1 + 1$

$= 22 + 23 + 1$

$= 46$

$O_2 = h_1w_6 + h_2w_8 + b_4$

$= 11 \times 4 + 23 \times 1 + 1$

$= 44 + 23 + 1$

$= 68$

By matrix representation:

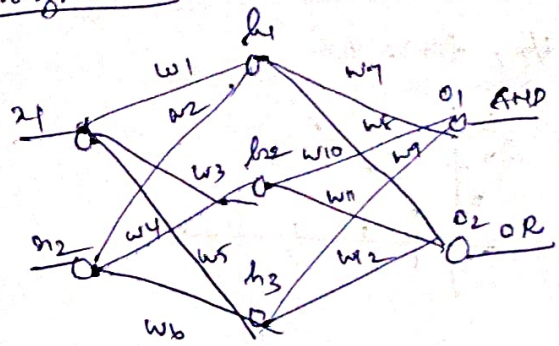
$$\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2+8+1 \\ 6+16+1 \end{bmatrix} = \begin{bmatrix} 11 \\ 23 \end{bmatrix}$$

$2\times2 \quad 2\times1$

$2\times1$

$$\begin{bmatrix} o_1 \\ o_2 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 11 \\ 23 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 22+23+1 \\ 44+23+1 \end{bmatrix} = \begin{bmatrix} 46 \\ 68 \end{bmatrix}$$

Assignment:




| $w_i$ | value |
|---|---|
| $w_1$ | 1 |
| $w_2$ | 2 |
| $w_3$ | 1 |
| $w_4$ | 3 |
| $w_5$ | 1 |
| $w_6$ | 4 |
| $w_7$ | 1 |
| $w_8$ | 2 |
| $w_9$ | 1 |
| $w_{10}$ | 1 |
| $w_{11}$ | 3 |
| $w_{12}$ | 3 |

$b_i = 1$ for $i = 1$ to 3

$b_4 = 2$

$b_5 = 1$

Activation functions:

↳ To introduce the non-linearity

↳ To learn more complex problems & patterns

↳ giving and not giving the neurons

Linearly separable

properties

↳ homogeneity (scaling)

$$f(x*k) = k\,f(x)$$

(Imp) ↳ Additive property (superposition)

$$f(a,b) = f(a) + f(b)$$

if the i/ps not proportional to the o/ps, then they are said to be non-linearly separable problems, doesn't satisfy the superposition properties.

exponential - $ae^x$

logarithmic - $\ln \log x$

Trigonometric - $\cos x$

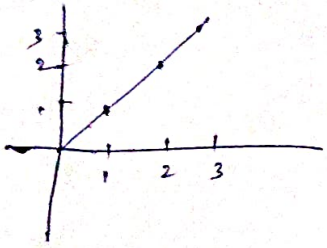
polynomial - $x^2 + 5x + 6$

Activation fn → linear
→ Non-linear

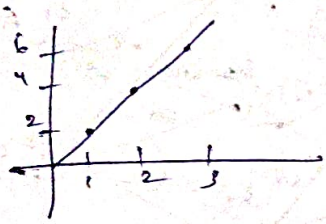
## Linear activation function

### ① identity

$\forall x$, $f(x) = x$




$\forall x$, $f(x) = 2x$

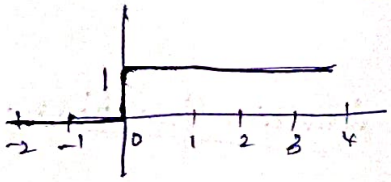


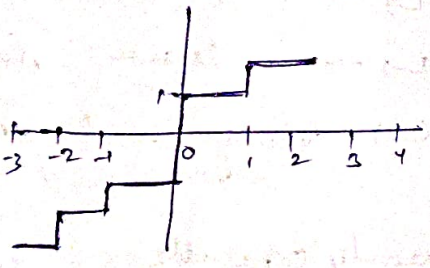

$f(x) = ax$

$f'(x) = a$

### ② Binary step Activation function / threshold activation f

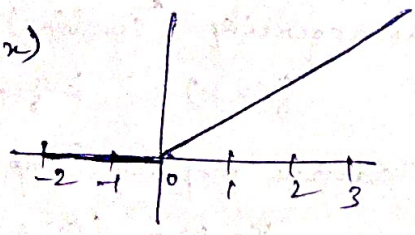
$$f(x) = \begin{cases} 0 & \text{if } x \le 0 \\ 1 & \text{if } x > 1 \end{cases}$$




### ③ ReLU Activation function

$f(x) = \max(0, x)$




not fixed - dead - sparse

↳ Make the models converge quickly

↳ Better than sigmoid and tanh activation fns

$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$

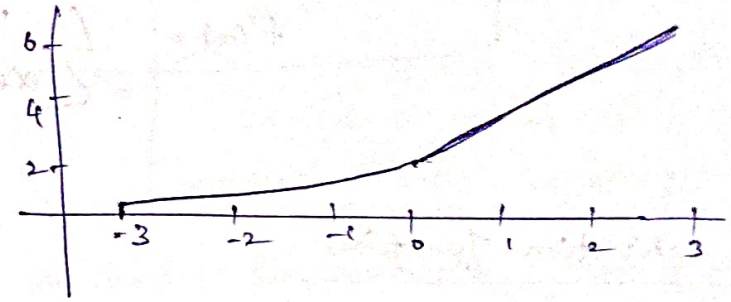
- vanishing gradient problem

$x = -4, \quad f(x) = 0$

$x = 17, \quad f(x) = 17$

④ Leaky ReLU Activation fn

$$f(a,x) = \begin{cases} \beta x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

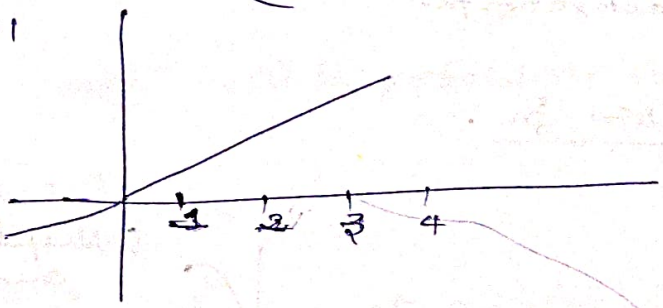



$\beta = 0.01$

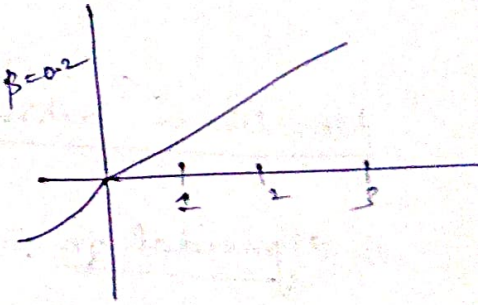
⑤ Parametric ReLU Activation fn

$$f(a,x) = \begin{cases} \beta x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

$\beta = 0.1$




$\beta = 0.2$

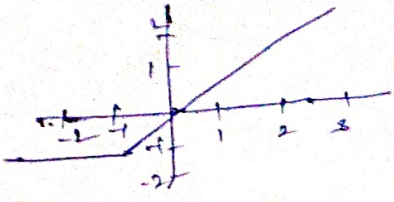



⑥ Randomized ReLU Activation fn

β value can chosen randomly

used for regularization

⑦ Shifted ReLU Activation fn



$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

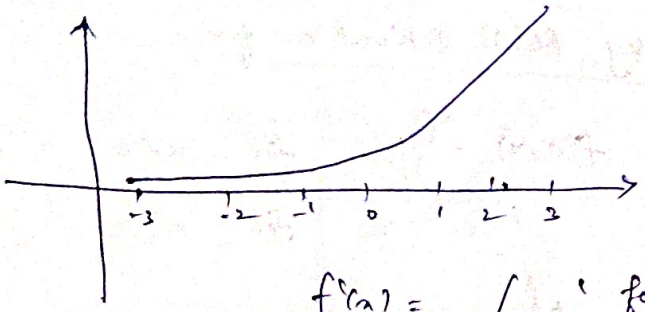
⑧ Exponential linear unit Activation fn (ELU) :

$$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

→ true value

$f(\alpha, x) \in [-\infty, \infty)$

log fns for -ve value




$$f'(x) = \begin{cases} 1 & \text{for } x \geq 0 \\ \alpha e^x & \text{for } x < 0 \end{cases}$$

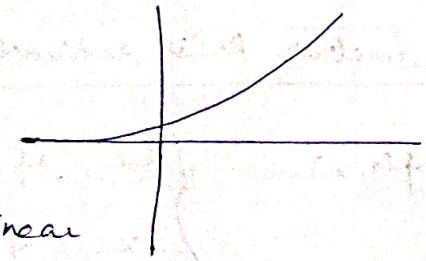
⑨ Softmax plus activation function

$\in (0, \infty)$

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

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

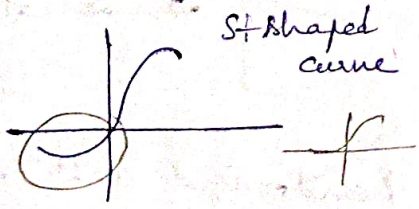
- used as a non-linear activation fn



Non-linear activation fn :

1) Sigmoidal fn :

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




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

$x = 4$

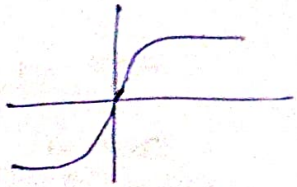
$$f(x) = \frac{1}{1 + e^{-4}} \simeq 0.98$$

2) Tanh (hyperbolic tangent activation fn)

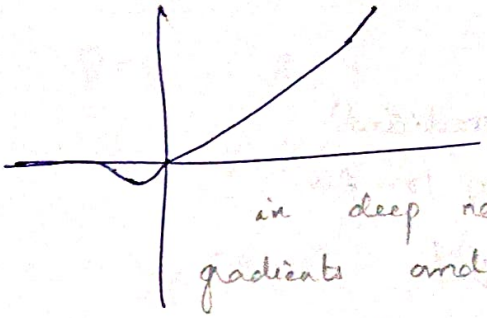
$$f(x) = \frac{2}{1+e^{-2x}} - 1$$

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

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



3) Swish activation fn :



$$f(x) = x * \sigma(\beta x)$$

- used to improve learning in deep neural N/ws by providing smooth gradients and avoiding dead neurons.

Soft-max fn used for multiclass classification problem

↳ 4) Softmax fn :

Let inputs → $x_i$ for $i = 1$ to $n$

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

output will be probabilistic value

$x_i = [4, 2, 1]$

$$f(x_1) = \frac{e^4}{e^4 + e^2 + e^1} \simeq 0.8438$$

$$f(x_2) = \frac{e^2}{e^4 + e^2 + e^1} \simeq 0.1142$$

$$f(x_3) = \frac{e^1}{e^4 + e^2 + e^1} \simeq 0.042$$

## Loss functions

Loss → difference b/w Actual & predicted

cost fn → Average of all loss fns

Loss → $L(y, \hat{y})$

Cost → $\frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{y}_i)$ ... mean error

1. Regression loss fn
2. classification loss fn

### ① Regression loss fn

error = $y - \hat{y}$ (actual - predicted)

mean error = Average of loss fns

MAE = $\frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$

(Ex)

$y = [5 \quad 6 \quad 8 \quad 10 \quad 12]$

$\hat{y} = [4 \quad 7 \quad 9 \quad 10 \quad 8]$

| $y$ | $\hat{y}$ | $y - \hat{y}$ | $\lvert y - \hat{y} \rvert$ | $(y_i - \hat{y})^2$ |
|---|---|---|---|---|
| 5 | 4 | 1 | 1 | 1 |
| 6 | 7 | −1 | 1 | 1 |
| 8 | 9 | −1 | 1 | 1 |
| 10 | 10 | 0 | 0 | 0 |
| 12 | 8 | 4 | 4 | 16 |
| | | 3 | 7 | 19 |

ME = 3/5 = 0.6 MAE = 7/5 = 1.4 MSE = 19/5 = 3.8

MAE - sensitive to the outliers, not differentiable

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

- easy to interpret
- not robust to outliers

RMSE = $\sqrt{MSE}$

SVM
- suitable for binary classification
- Robust to noise

## Huber loss

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

Avg Huber loss - Avg of huber loss over the sample

MAE ≤ Huber loss ≤ MSE

Huber loss is a regression that combines
- Mean Squared Error(MSE) for small errors
- Mean Absolute Error(MAE) for large errors

It is robust to outliers

$y = 6 \quad \hat{y} = 4 \quad \delta = 4 \quad : \quad y - \hat{y} = 6 - 4 = 2$

$Huber(y, \hat{y}) \Rightarrow y - \hat{y} = 2 \le 4$

$\frac{1}{2}(6-4)^2 = \frac{1}{2} \times 2^2 = 2$

Intuition:
Small errors → behave like MSE (smooth learning)
Large errors → behave like MAE (less sensitive to outliers)
Goal: Be accurate but not get disturbed by extreme values

## Hinge loss $\in [-1, 1]$

$$Hinge(D) = \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y\,h(y))$$

↳ mostly used for binary classification

$y = [1 \quad 1 \quad -1 \quad -1]$

$\hat{y} = [0.5 \quad 1.5 \quad 0.8 \quad -1.4]$

| $y$ | $\hat{y}$ | $yh(y)$ | $1 - yh(y)$ | o/p |
|---|---|---|---|---|
| 1 | 0.5 | 0.5 | 0.5 | 0.5 |
| 1 | 1.5 | 1.5 | -0.5 | 0 |
| -1 | 0.8 | -0.8 | 1.8 | 1.8 |
| -1 | -1.4 | 1.4 | -0.4 | 0 |

Hinge loss is a loss function mainly used for binary classification, especially in SVMs. It focuses on creating a clear margin b/w classes

$Hinge(D) = \frac{1}{4}\Sigma(0.5, 0, 1.8, 0)$

$= \frac{1.8 + 0.5}{4} = 0.575$

Robust to outliers means that a method or loss function does not get affected by extreme or unusual data points

29/12/25

## ② Classification based Error functions

outcome → categorical

Two types
- Binary cross entropy / log loss function
- categorical cross entropy

### Binary cross entropy

$$L(y, \hat{y}) = -(y \log \hat{y} + (1-y) \log \hat{y})$$

$y=1, \hat{y}=0.3$

$$L(1, 0.3) = -(1 \log 0.3 + (1-1) \log 0.3)$$

$$= -\log 0.3$$

$$= 1.7369 \quad 0.5229$$

### Multivariate / categorical cross entropy function

$$L(y, \hat{y}) = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

N → total no. of possible outcomes

$y_i$ → one shot vector of the encoded actual value (actual outcomes)

$\hat{y}_i$ → one shot vector of the predicted value

Consider three items red, green and blue

r(1 0 0) g(0 1 0) b(0 0 1)

Let's assume that possible o/p of the neural n/w is probability → P(0.5, 0.4, 0.1)

Find the cross entropy for red

$$CE(P) = -(1 * \log(0.5) + 0 * \log(0.4) + 0 * \log(0.1))$$

$$= -(-0.301)$$

$$= 0.301$$

$$CE(g) = -(0 * \log 0.5 + 1 * \log 0.4 + 0 * \log 0.1)$$

$$= 0.3979$$

$$CE(b) = -(0 * \log 0.5 + 0 * \log 0.4 + 1 * \log 0.1)$$

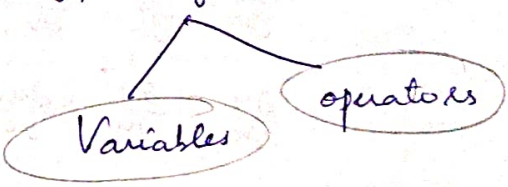
$$= 1$$

## Role of computational graph and chain rules

### Computational graph

- Acyclic graph that shows the graphical representation of the mathematical model
- To show how the derivatives are calculated during the backpropagation process

Two types of nodes



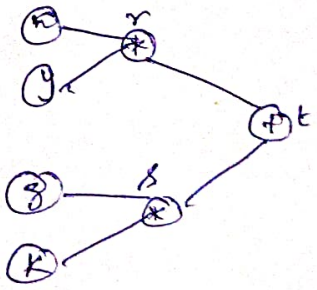

(i/ps, bias, weights)

Consider a simple example:

$$f = xy + zk$$

$$r = xy \quad s = zk$$

$$f = r + s$$




### chain rule:

- used to find derivative of the computational functions
- functions may be univariate or multivariate
- Composite: Combination of both fcns

$$g(x, y, m, r, s)$$

$$u = g(f(x))$$

### Chaining

↳ Sequence of combination of the functions is called the chaining process

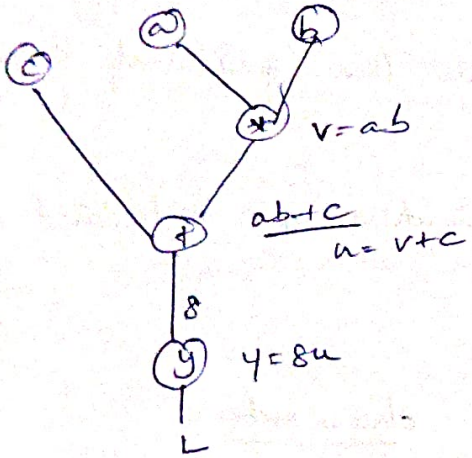
Consider derivative of $h(x) = u$ then, derivative $\{g(h(x))\}$:

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial a} \times \frac{\partial a}{\partial x}$$

The partial derivative of the Composite function is product of partial derivative of u w.r.t v and derivative of v with respect to x.

Neural N/ws:

$$f3(f2(f1(f(x))))$$




$f(a,b,c) = (a+b+c)8$

1) $v = ab$

2) $u = v+c$

3) $y = 8u$

$$\frac{\partial y}{\partial u} = 8 \quad \frac{\partial u}{\partial v} = 1 \quad \frac{\partial u}{\partial c} = 1$$

$$\frac{\partial y}{\partial v} = \frac{\partial y}{\partial u} \times \frac{\partial u}{\partial v} = 8 \times 1 = 8$$

$$\frac{\partial y}{\partial c} = \frac{\partial y}{\partial u} \times \frac{\partial u}{\partial c} = 8 \times 1 = 8$$

$$\frac{\partial v}{\partial a} = b \quad \frac{\partial v}{\partial b} = a$$

$$\frac{\partial y}{\partial a} = \frac{\partial y}{\partial u} \times \frac{\partial u}{\partial v} \times \frac{\partial v}{\partial a}$$

$$= 8 \times 1 \times b = 8b$$

$$\frac{\partial y}{\partial b} = \frac{\partial y}{\partial u} \times \frac{\partial u}{\partial v} \times \frac{\partial v}{\partial b}$$

$$= 8 \times 1 \times a = 8a$$

## Backpropagation Algorithm :

Two stages
- forward stage
- Backward stage

Algo : 1. Read the input
2. Assign the input to the input layer

3. Compute all the neurons of hidden layers

$$net_j = w_0 + \sum_{i=1}^{n} x_i w_i$$

$$O_j = \frac{1}{1+e^{-net_j}}$$

4. Compute the difference between the predicted o/p and the target o/p as an error

o/p layer : $\delta_j = (t_j - o_j)\, o_j (1 - o_j)$

hidden layer : $\delta_k = o_j (1-o_j) \sum \delta_k w_{jk}$

5) $\Delta w_{ij} = \eta \delta_j o_i$
$w_{ij} = w_{ij} + \Delta w_{ij}$ } update the weights

$\theta_j = \theta_j + \Delta \theta_j$ → update the bias

---

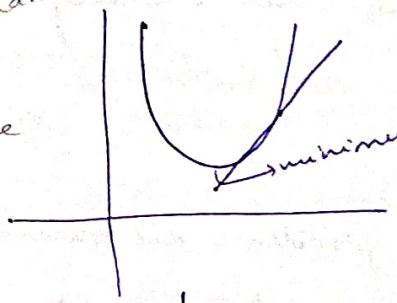
## Optimization Algorithms

Optimization algo's search for critical points because
- Minimum loss occurs at a critical point
- updates naturally flow as gradient →
- They indicate possible solution

Problems
↓
Two types
- Constrained
- unconstrained

parameters
- model parameters
- hyperparameters
  ↓
  parameters not learned from the data, but associated with the algo's

$\min f(x;\theta)$ → model parameters
$x \in X$ search space




$f'(x) = \frac{dy}{dx} = 0$ ↳ critical points (variables)

$f''(x) = \frac{d^2y}{dx^2}$

$f(x) = x^2 + 2x + 1$

$f'(x) = 2x + 2$

$2x + 2 = 0$

$2x = -2$

$x = -1$

$f''(x) = 2 > 0$

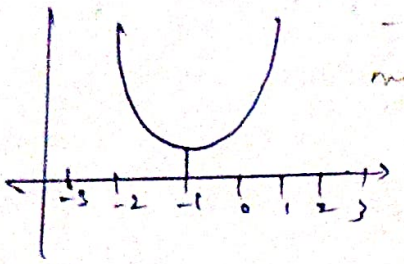
+ve → local minimum

$f(-1) = (-1)^2 + 2(-1) + 1 = 0$

Substitute the critical points in the 2nd order derivatives

1) +ve → critical point is local minimum
2) −ve → local maximum
3) 0 → Inconclusive

## Gradient descent algo




- an optimization algo used to minimize a loss (cost) function by iteratively updating model parameters in a direction that reduces the error the most.

Multivariate function - $f(x_1, x_2, \ldots x_n)$

$$\nabla f(x) = \left( \frac{\partial f(x_1, x_2, \ldots x_n)}{\partial x_1}, \frac{\partial f(x_1, x_2, \ldots x_n)}{\partial x_2}, \ldots \frac{\partial f(x_1, x_2, \ldots x_n)}{\partial x_n} \right)$$

↓ first order derivative

→ one shot vector

## Gradient descent algorithm:

1) Sophisticate (stochastic) gradient
2) Adaptive gradient
3) Adam optimizer

### Optimization pblm

A problem where we must choose the best solution from a set of possible solutions (search space) by maximizing or minimizing an objective function, subject to certain constraints ✓

## Features of optimizer

↳ Minimizing the loss function
↳ efficiency
↳ handling of non-convex functions
↳ To provide a complex loss landscape
↳ Adaptation to the data
↳ To improve the convergence of the model ✓

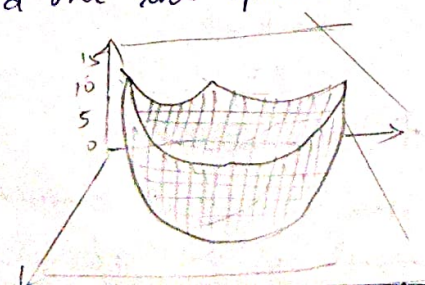
Decision Variables - The values we are allowed to change
(eg) weights w, bias b, learning rate

## Search space:

Search space - Set of all possible values of the decision variables contains all candidate solutions

↳ optimization problems are framed as search space
↳ All the candidate outputs are framed as search space
↳ optimization problems will find out the optimal or best soln from the search space

Candidate soln = one possible answer from the search space



Two types of optimization pblms

Convex (have a clear boundary b/w the classes)
eg Linear regression, SVM (with linear kernel)

Non-Convex (multiple minimum - need to find the global minimum)
→ 1st order optimization algo
→ 2nd order optimization algo
eg Neural networks, k-means clustering

local minimum - The point where the function value is smaller than the nearby values

Global minimum - The point where the fn val is smallest over the entire search space

decision boundary complex

## Hyper parameter tuning

↳ tune the hyperparameters so that the model's performance can be improvised

values set before training, not learned from the data

eg learning rate

↳ It tells how much data that model sees before updating weights

Batch size - No. of samples used per weight update

## Basic Algorithms

↳ Random search
↳ Grid search

## Random search :

↳ sample the input randomly from the distribution
↳ check for all combination of inputs, evaluate the model and check the results
↳ pick the best combination of inputs that generalizes the model well
↳ less memory required, can be done parallelly, thus computationally intensive.

## Grid search algo :

↳ explore all possible combinations
↳ Take the hyperparameters with discrete set of values
↳ Find all combinations of input

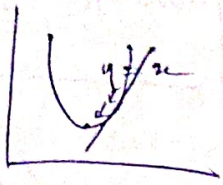
Parameters

A (1,2) B (3,4)

↳ (1,2) (3,4)
(1,4) (2,3)

# Tuning the model parameters

## 1. Gradient descent algorithm




- Meaning of Gradient - slant/slope of the surface
- minimize the loss function

$f(x_1, x_2)$ direction $\left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}\right)$

magnitude $\sqrt{\left(\frac{\partial y}{\partial x_1}\right)^2 + \left(\frac{\partial y}{\partial x_2}\right)^2}$

$f(x,y) = x+y$

gradient vector $= \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$ → direction of the gradient

$= (1,1)$

magnitude $= \sqrt{1+1} = \sqrt{2}$

$dir = \frac{\nabla f(x,y)}{\|\nabla f(x,y)\|} = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$

represented with norm values

Steps

1. $w_0$
2. $\Delta w_0 = w_0 - \frac{\partial f}{\partial w_0} \times \eta$
3. $\Delta w_{t+1} = w_t + \eta \frac{\partial f}{\partial w_t}$

(eg) $x^2 + 2x + 1$ . $f(x) = x^2 + 2x + 1$

$x_0 = 1 \quad \eta = 0.1$ $f'(x) = 2x$

$f''(x) = 2$

$x_{t+1} = x_t - \eta \frac{\partial f}{\partial x_t}$

$x_1 = x_0 - 0.1 \times \frac{\partial f}{\partial x_0}$

$x_1 = 1 - 0.1 \times (2(1)+2)$

$x_1 = 1 - 0.1 \times (4)$

$x_1 = 1 - 0.4$

$x_1 = 0.6$

$x_2 = x_1 - \eta \frac{\partial f}{\partial x_1}$

$= 0.6 - 0.1(2(0.6)+2)$

$= 0.6 - 0.1(3.2)$

$= 0.6 - 0.32$

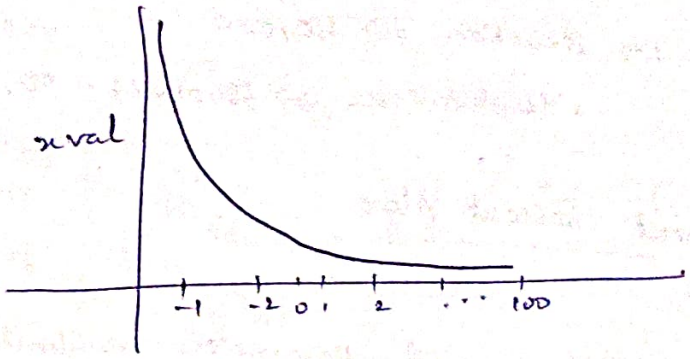
$= 0.28$

$x_3 = x_2 - \eta \frac{\partial f}{\partial x_2}$ $\quad f'(x) = 2(0.28) + 2$

$= 0.28 - (0.1) \times 2.56$ $\quad = 0.56 + 2 = 2.56$

$= 0.024$

Loss functions keep on decreasing each iteration




Batch Gradient descent

1) Take all training samples
2) Compute predictions for all samples
3) Calculate total or avg loss
4) Compute gradient using entire data
5) update param - model parameters are updated using the entire training dataset at once in each iteration

## 2. Batch gradient descendant

Algo

- Initialize the model parameters randomly and the learning rate
- Repeat until the iteration converges or number of iterations is reached

↳ For entire dataset, compute the cost function

$$\nabla J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla J(\theta, x_i, y_i)$$

where m is the number of samples in the training set

↳ $\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t)$

(2m) - advantage and disadvantage of gradient descent algo

**Advantages**

↳ converge to a global minima is smoother compared to stochastic gradient descendant

↳ stable convergence

↳ Convergence is guaranteed for convex functions not for non-convex functions

↳ Suitable for small datasets

Disadvantages

↳ computationally intensive for large datasets

↳ memory requirement is higher.

eg Consider 100 data points with five features

computations = $5 \times 100 = 500$

100 iteration => 50,000

1 iteration - 1sec => $50,000 \times 1 = 50,000$ sec.

2) Stochastic gradient descent Algo : - model updates its parameters after seeing one training sample at a time

Algo

- Initialize the model parameters randomly
- Initialize the learning rate randomly
- Repeat until iterations converge or no. of iterations is reached
  - ↳ Shuffle the dataset
  - ↳ Training Sample randomly, compute gradient of the loss of the current point w.r.t model parameter

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t)$$

working

- Init model params
- pick one training sample
- predict and compute loss
- update params, repeat for all samples (one epoch)
- continue for multiple epoch

Advantages

↳ Frequent updates

↳ memory requirement is less

↳ Adaptive for online learning

Disadvantages

↳ computationally very intensive

↳ data comes one by one

↳ model updates immediately after each sample

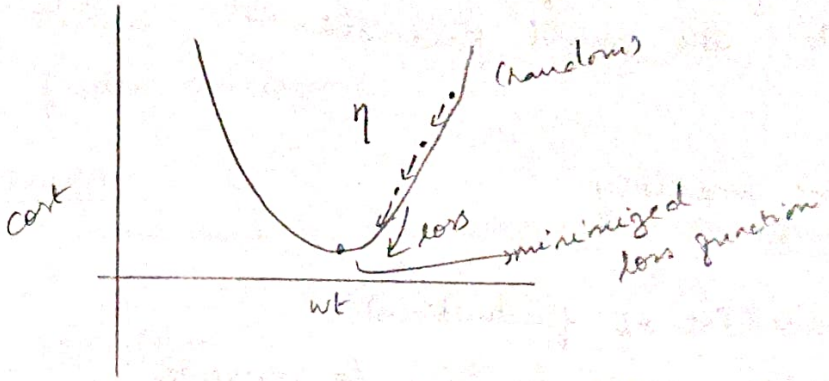
↳ No need to store all past data

05/02/2026

## Steps in Gradient descent :-

1. Initialize all the model parameters randomly
2. Find out gradient of the loss fn with respect to each of the parameters
3. update the model parameters in order to minimize the loss function

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t)$$




## Gradient descent with Armijo Goldstein condition

- To reduce the objective function and loss function using line search algorithm

$$\delta(x^{t-1} + \eta \nabla f(x^{t-1})) - f(x^{t-1}) \geq c\eta \nabla f(x^{t-1})$$

$$c \in \{0-1\}$$

## Gradient descendent with full Armijo Goldstein condition

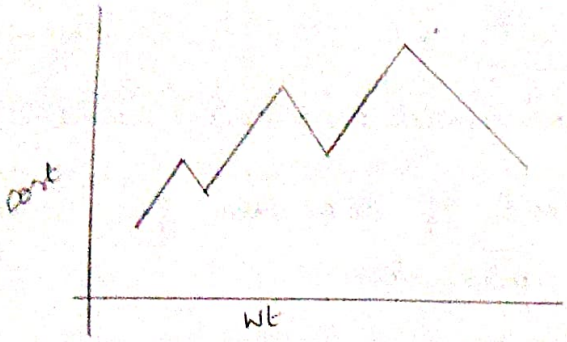
- uses 2nd order gradient

$$f(x^{t-1} + \eta \nabla f(x^{t-1})) - f(x^{t-1}) \geq c.\eta (\nabla f(x^{t-1}))^2 + \tfrac{1}{2} \eta^2 f(x^{t-1})^T \cdot H(x) \cdot \nabla f(x^{t-1})$$

↳ Hessian matrix

## Variants of gradient descent Algo's

1. Stochastic gradient descent algo
   - update model parameters after each and every sample




not a smooth curve, lot of fluctuations

Steps:

1. Select one training (e.g) randomly
2. Compute gradient of the loss function
3. update the model parameters

Advantages

- less memory

disadvantages

- noise (lot of fluctuations)
- requires more iterations to converge

## Mini Batch Gradient descent Algo

Steps

1) Select the mini-batch with fixed size
2) Compute the mean gradient of the mini batch
3) update the model parameters
4) Repeat all the above steps for all remaining mini-batches

↳ lies b/w Stochastic and batch gradient descent

↳ requires medium amount of memory and less time to converge compared to SGD (Stochastic Gradient Descent)

problem - it may stuck with local minima

## Stochastic gradient descent with momentum

↳ helps to accelerate the convergence rate by smoothing out the noisy gradients of Stochastic Gradient descent, thus reducing the fluctuations and improve the speed of

convergence compared to SGD Algo.

$$V_{t+1} = \beta V_t + (1-\beta)\nabla J(\theta_t) \qquad \beta \in [0, 1]$$

where, $V_t \rightarrow$ momentum at time 't'

$\beta \rightarrow$ momentum coefficient

$$\theta_{t+1} = \theta_t - \eta V_{t+1}$$

Advantages
- mitigate the oscillations
- Reduce the variance
- faster convergence

disadvantages
- we have to tune momentum co-efficient 'β' properly

Advance Optimizers:

1. AdaGrad (Adaptive Gradient descent algo)
- Adapts the learning rate for each and every model parameters based on the historical gradient info
- effective for the sparse (less-frequently used) features

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla J(\theta_t)$$

where, $G_t \rightarrow$ sum of the squared gradients

$\epsilon \rightarrow$ small constant to avoid divide by zero error

Advantage
- adapt the learning rate for each of the parameter to improve the training efficiency

problem
- learning rate decays very quickly - so it leads to slow convergence of the model

2. RMS prop
- Improvement over the AdaGrad with decay factor to prevent the learning rate decreasing too rapidly.

$$E[g^2]_t = \gamma \cdot E(g^2)_{t-1} + (1-\gamma)(\nabla J(\theta_t))^2$$

($\gamma$ → decay factor)

↳ Exponentially weighted Moving Average (EWMA) of the square gradient

$\forall \gamma \in [0, 1]$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E(g^2)_t + \epsilon}} \nabla J(\theta_t)$$

Advantage
- prevent the excessive decay of learning rate

Disadvantage
- Computationally intensive

3. Adam optimizer
- Adaptive gradient with momentum
- combines RMS prop with Momentum
- uses both first order and second order momentum of gradient to adapt learning rate for each parameter

Steps:
- update the first order momentum

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \nabla J(\theta_t)$$

- update the second order momentum

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla J(\theta_t))^2$$

- Bias correction

$$\hat{m_t} = \frac{m_t}{1-\beta_1^t} \qquad \hat{v} = \frac{v_t}{1-\beta_2^t}$$

- update the model parameters

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v_t} + \epsilon}} + \hat{m_t}$$

where, $\beta_1$ and $\beta_2$ are the decay factors for the 1st order and 2nd order momentum respectively

Advantage - Faster convergence  Disadvantage - Heavy computation, requires lot of memory

## Comparison of optimizers:

| optimizer | Advantages | Disadvantages |
|---|---|---|
| ~~SDG~~ SGD | Simple and easy of implement | Slow convergence |
| Mini Batch ~~SDC~~ SGD | Faster than SGD | Computationally expensive, stuck in local minima |
| SGD with momentum | faster convergence, reduced noise | Tune the β parameter properly |
| AdaGrad | Adaptive learning rate for all parameters | learning rate decays rapidly, slow convergence |
| RMS prop | Prevents fast decay of learning rate | Computationally intensive |
| Adam | Fast convergence due to combining momentum and RMSprop | Memory intensive, computationally expensive. |

### Tuning the model params:

### Steps in Gradient descent algo

↳ Init model params with random values

↳ Find the derivative of the univariant function $\left(\frac{\partial f}{\partial \theta}\right)$

$$f(x_1, x_2) = (x_1, x_2)\left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right)$$

$$\text{magnitude(norm) } \nabla f(x) = \sqrt{\left(\frac{\partial f}{\partial x_1}\right)^2 + \left(\frac{\partial f}{\partial x_2}\right)^2}$$

(eg) $f(x_1, x_2) = x + y$

$$\nabla f(x_1, x_2) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right) = (1, 1)$$

$$|\nabla f(x)| = \sqrt{1^2 + 1^2} = 2$$

$$u = \frac{\partial f(x_1, x_2)}{|\nabla f(x_1, x_2)|} = \frac{(1,1)}{\sqrt{2}} = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$$

direction of the gradient

1) $w_0$

2) $\Delta w_0 = w_0 - \frac{\partial f}{\partial w_0}$

3) $w_{t+1} = w_t - \eta \nabla w_0$

(8.2) $f(x) = x^2 + 2x + 1$

$\eta = 0.1 \qquad x_0 = 1$

$f'(x) = 2x + 2$

$x_{t+1} = x_t - \eta f'(x_t)$

$f'(x_0) = 2(1) + 2 = 4$

$x_1 = x_0 - \eta f'(x_0)$

$= 1 - (0.1)\,4$

$= 1 - 0.4$

$= 0.6$

$f'(0.6) = 2(0.6) + 2$

$= 1.2 + 2$

$= 3.2$

$x_2 = x_1 - \eta f'(x_1)$

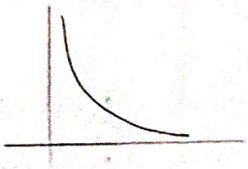
$= 0.6 - 0.1 \times 3.2$

$= 0.28$

$f'(0.28) = 2(0.28) + 2$

$= 0.56 + 2$

$= 2.56$



$x_3 = x_2 - \eta f'(x_2)$

$= 0.28 - 0.1 \times 2.56$

$= 0.28 - 0.256$

$x_3 = 0.024$

## Batch gradient descent

1. $\theta$
2. $\eta$, no. of iterations
3. Repeat until the model converges or the number of iterations reached

4) For entire dataset, compute cost fn

$$\nabla J(\theta) = \frac{1}{m} \cdot \sum_{i=1}^{m} \nabla J(\theta, x_i, y_i)$$

b) update the model parameter

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta)$$

4) return the optimal parameters

Advantage

- converge to global minima in a smoother way compared to Stochastic gradient descent
- stable convergence
- suitable for small dataset

Disadvantage

- computationally intensive if dataset is large
- memory requirement will also be larg if the dataset is large

Stochastic gradient descent algo

1. $\theta$
2. $\eta$, no. of iterations
3. Repeat until the convergence or no. of iterations is reached
   a) shuffle dataset
   b) Randomly choose a sample
   c) $\nabla J(\theta, x, y)$
   d) $\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t)$

Advantages

- Frequent updates support large datasets (as only one sample is randomly chosen)
- requires less memory
- Adaptive for online learning
- overcome the local minima by randomness

Disadvantage

- Due to randomness, noise is introduced, leads to noisy gradient leading to oscillations and fluctuations, leads to fail to reach the local minima.

## Mini Batch gradient descent:

1) $\theta$, $\eta$, no. of iterations
2) no. of Batches (b), k Samples
3) Repeat until convergence or no. of iterations is reached

$$\nabla J(\theta) = \frac{1}{k} \sum_{i=1}^{k} \nabla J(\theta, x^i, y^i)$$

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta)$$

} for each batch b

4) return the finetuned model parameters $\theta$, end.

Advantage
- lies b/w SGD and batch GD
- steady and stable convergence

Disadvantage
- noise due to randomness
- no guarantee of convergence

Consider 10,000 data points

1) SGD = 10,000
2) BGD = 1
3) mini Batch (5) = $\frac{10000}{5}$ = 2000

## Concepts of Momentum

↳ Accelerate the model convergence by collecting the gradient in a variable velocity

↳ velocity is the smoothened version of gradient

↳ Momentum is added with model parameter update

accelerate convergence speed, dampening oscillations, avoid the saddle points, efficient navigation of plato, improves the generalization which leads to avoiding the overfilling

Two types
- Traditional momentum
- Nesterone momentum

Traditional: ⇒ momentum coeff $\in [0,1]$.

$$V_t = \beta \cdot V_{t-1} + (1-\beta) \nabla J(\theta_{t-1})$$

(or)

$$V_{t+1} = \beta \cdot V_t + (1-\beta) \nabla J(\theta_t)$$

$$\theta_{t+1} = \theta_t - \eta V_{t+1}$$

Nesterone:

future $V_{t+1} = \beta V_t + (1-\beta) \nabla J(\theta_t)$

past $V_{t-1} = \beta \cdot V_{t-2} + (1-\beta) \nabla J(\theta_t)$

$$\theta_{t+1} = \theta_t - \eta (V_{t+1} + V_{t-1})$$

adjusting the model parameters based on the past and future

Adaptive Gradient.

- considering Square of Gradients in Normalized Variable for all model parameters

Steps:

1) Init - $\theta$, $\eta$, no. of iterations
2) Repeat until convergence or no. of iterations is reached
   a) $r = r + g \cdot g$
   b) $\nabla\theta = \eta \times \frac{1}{\sqrt{r+\epsilon}} \cdot g$ | $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{A_t+\epsilon}} \nabla J(\theta) V_{t-1}$
   c) $\theta = \theta - \Delta\theta$
3) Return the updated $\theta$

Advantage - adapts l.r for each parameter

disadvantage - l.r decays quickly

Speciality of AdaGrad - personalized l.r for each parameter
- adaptation is based on gradient history
- effective for sparse and high dimensional data

06/01/26

## RMS prop :

↳ improvement over the AdaGrad

↳ using decaying average or Moving average of partial derivative

### RMS prop

* uses decaying avg or moving avg of the partial derivative
* Exponential weighted to moving that gives higher weightage to the recent updates rather than the past history

### AdaGrad

* Sum of square gradients
* Square of partial derivatives grow monotonically, hence leads to slow rate of convergence

* $y_0 = x_0$

$y_n = \rho x_n + (1-\rho)\, y_{n-1}$

$\rho \rightarrow$ decay factor

when $\rho = 1$, $y_n = x_n$

$\rho = 0$, $y_n = y_{n-1}$

$\rho \in [0-1]$

| $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ |
|---|---|---|---|---|
| 5 | 10 | 12 | 14 | 16 |

$\rho = 0.2$

$y_0 = x_0 = 5$

$y_1 = \rho x_1 + (1-\rho)\, y_0$

$y_1 = 0.2 \times 10 + (1-0.2) \times 5$

$= 0.2 \times 10 + 0.8 \times 5$

$= 2 + 4 = 6$

$y_2 = 0.2 \times 12 + (1-0.2) \times 6 = 7.2$

$y_3 = 0.2 \times 14 + (1-0.2) \times 7.2 = 8.56$

algo

1) $\theta, \eta$, no. of iterations, $\epsilon$

2) Repeat until the model converges or the no. of iterations is reached

a) EWMA of the gradient

$$\gamma = \beta\gamma + (1-\beta) g \cdot g$$

b) update the parameters

$$\Delta\theta = \eta \times \frac{1}{\sqrt{\gamma + \epsilon}} \cdot g$$

$$E[g^2]^t = \gamma \cdot E[g^2]^{t-1} + (1-\gamma)(\nabla J(\theta t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]^t + \epsilon}} \cdot (\nabla J(\theta))$$

c) Apply update

$$\theta = \theta - \Delta\theta$$

Specialty

↳ Adaptive learning for each parameter

↳ Reduction in oscillation towards minimum so convergence is stable

↳ Computationally very effective

Adam optimizer (Adaptive Movement Estimation):

↳ This combines both gradient descent with momentum (RMS prop)

↳ Adaptive l.r for each parameter because of RMS prop

↳ Bias correction to remove the noise

1) $m_t = \beta_1 m_t + (1-\beta_1) \nabla J(\theta t)$

$v_t = \beta_2 v_t + (1-\beta_2) (\nabla J(\theta t))^2$

2) Bias correction:

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

Variance correction:

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

3) Model parameter update

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

Algo:

1) Init: $\theta$, $\eta$ and no. of iterations, $\epsilon$, $\rho_1$, $\rho_2$

2) Repeat until convergence or the no. of iterations is reached

a) $m = \rho_1 m + (1-\rho_1) \cdot g$

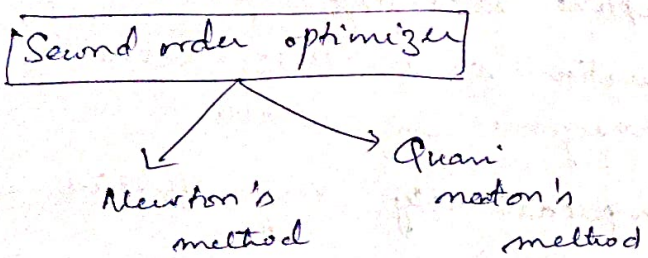
b) $v = \rho_2 \cdot v + (1-\rho_2) \cdot g \times g$

c) $\hat{m} = \frac{m}{1-\rho_1^t}$ $\quad \hat{v} = \frac{v}{1-\rho_2^t}$

d) update

$$\Delta\theta = \eta \cdot \frac{1}{\sqrt{\hat{v} + \epsilon}} \cdot \hat{m}$$

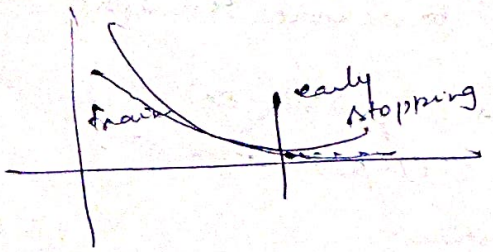
e) $\theta = \theta - \Delta\theta$




## Regularization

↳ done inorder to prevent overfitting (works well in training, but poor performance in test sample)

↳ regularization and Generalization



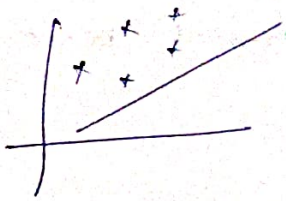
error - diff b/w actual & predicted

bias - Avg of all errors (cost)

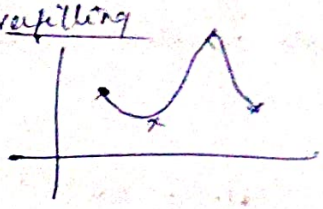
Variance - 2nd order derivative → inconsistency and erratic predictions because of fluctuations

generalization error = bias$^2$ + variance

underfitting :



overfitting



| Condition | model | Remarks |
|---|---|---|
| 1) high bias and high variance | ~~underfitting~~ useless | |
| 2) high bias and low variance | underfitting | |
| 3) low bias and high variance | overfitting | |
| 4) low bias and low variance | Ideal | |

Difference b/w overfitting and underfitting :

| underfitting | overfitting |
|---|---|
| * too simplistic to capture real time scenario and failed to capture the underlying patterns in data | * very complex, fits the data as well as the noise and fluctuation present in the data |
| * characterized by high bias and low variance | * characterized by low bias and high variance |
| * performance is poor during training and testing | * performance good in training, poor in testing |

* poor generalization * poor generalization

$M_1, M_2$ - models } we can choose complex model $m_2$
↓ ↓ provided performance($M_2$) > performance($M_1$)
Simple complex

$M_1 = 2x^2 + 3$

$M_2 = 6x^4 + 7x + 2x^2 + 3$

$L_1 = |x_1| + |x_2| + |x_3| + \cdots + |x_n|$

norm:

$M_1 = 2$

$M_2 = 6 + 7 + 2 = 15$

L2 norm: $\|x\|^2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$

$M_1 : \sqrt{4} = 2$

$M_2 : \sqrt{6^2 + 7^2 + 2^2}$

$L_\alpha = |x|_\alpha$

$= \max(|x_1|, |x_2| \ldots |x_n|)$

$L_p = \max_i \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}$

Regularization ways

↳ Neural N/w Initialization
↳ Neural N/w configuration
↳ Neural N/w Regularization

Weight Initialization

↳ zero weight Initialization
↳ Random Initialization $\in (-0.5, 0.5)$
↳ Xavier(or) Glorot Initialization

$$w \simeq U\left[\frac{-\sqrt{6}}{\{fan_{in}+fan_{out}\}}, \frac{+\sqrt{6}}{\{fan_{in}+fan_{out}\}}\right]$$

$$N(0,1) \sim \frac{\sqrt{1}}{fan_{in}}$$

b) $$\sigma = \frac{2}{\sqrt{fan_{in}+fan_{out}}}$$

4) He uniform distribution

$$w \simeq U\left[-\sqrt{\frac{6}{fan_{in}}}, +\sqrt{\frac{6}{fan_{out}}}\right]$$

5) LeCun Initialization

-uses guassian distribution

$$\sigma = G\left[\frac{\sqrt{3}}{fan_{in}}, \frac{+\sqrt{3}}{fan_{out}}\right]$$

width - no. of neurons in each layer ( maximal no. considered)

depth - no. of hidden layers in the model

$$N_i \leq N_h \leq N_o$$

$$N_h < 2N_i$$

$$N_h = \frac{N_s}{N_i}(\alpha N_i + N_o)$$