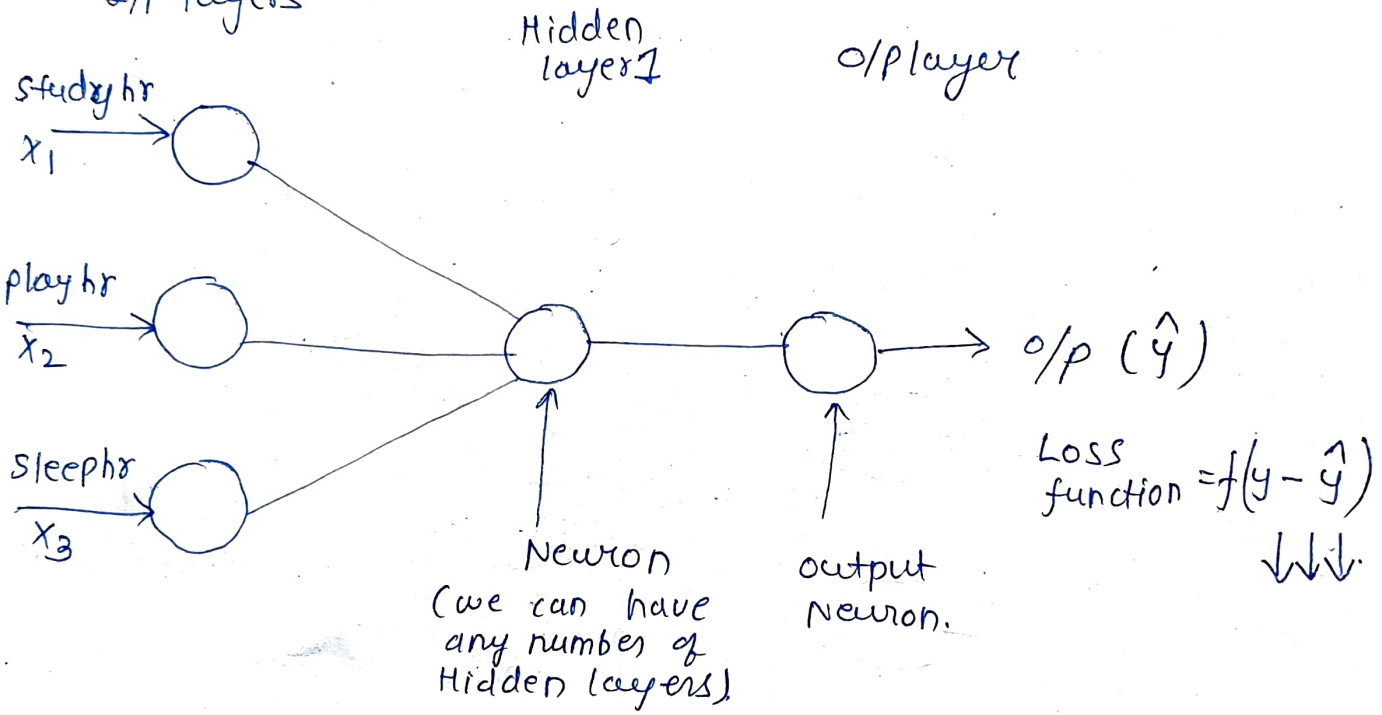
Deep Learning.

why Deep learning ??

1) we have huge amount of data.
2) Advancement in computer Hardware (GPU's)

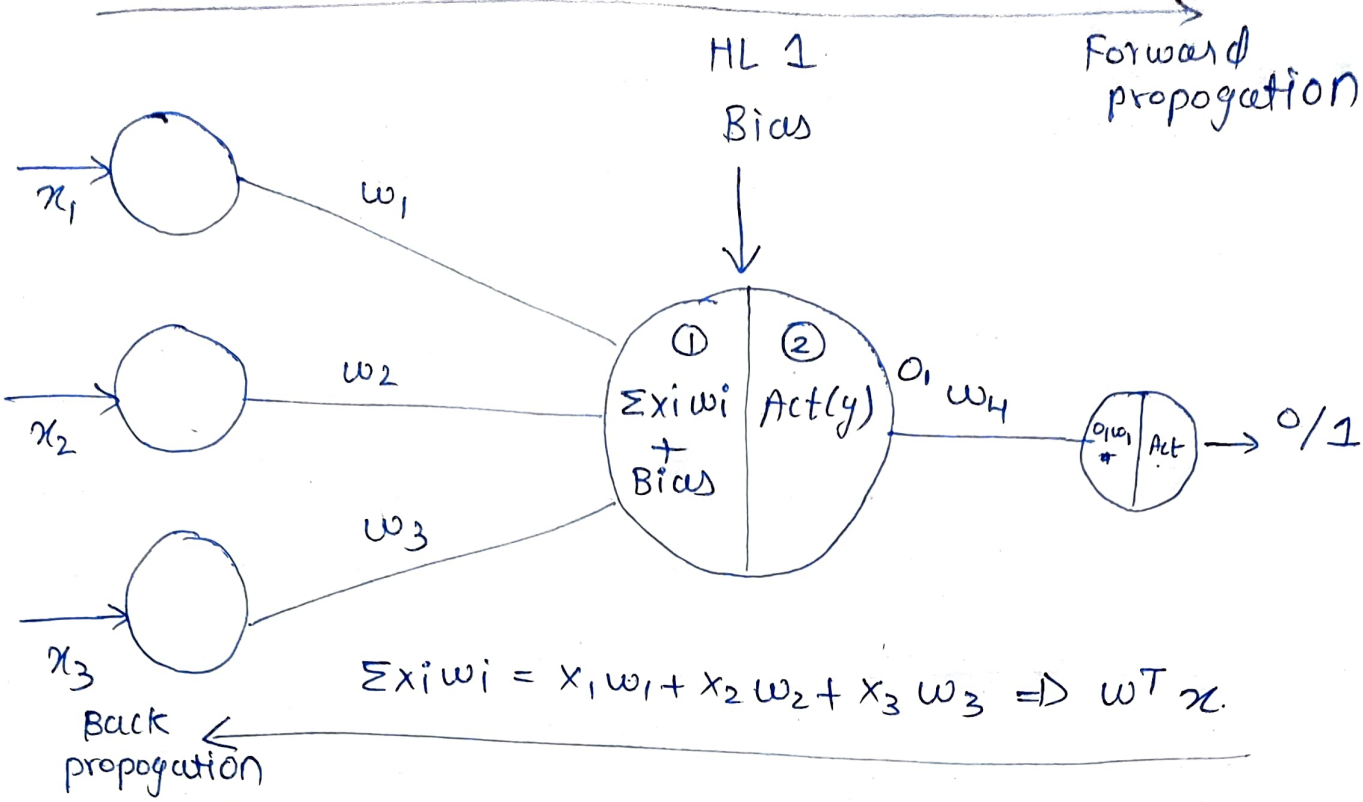
*perceptron

(single layered Neural network)



Dataset (Binary classification)

Study hr	play hr	sleep hr	pass/fail → 1, → 0
7	3	7	1
2	5	8	0
4	3	7	1



$$\Sigma x_i w_i = x_1 w_1 + x_2 w_2 + x_3 w_3 \Rightarrow w^T x.$$

weights are the real values that are attached to each i/p/feature and they convey the importance of that corresponding feature in predicting the final o/p.

weights convey following things

1) Importance of feature in predicting o/p value.
   weight ↑ feature value ↑.

2) Relationship b/w feature and target/output
   Ex: car price, car popularity → o/p Buy/or not.
   Buy car ∝ car popularity
   Buy car ∝ 1/car price
   w_1 (car price) + w_2 (car pop) + b = 0
   price of car ↑ w_1 (car price) ↑ ⇒ we buy car.
   but this we don't want
   so we will assign w_1 a lower value.
   or -ve value.

3) wt plays an important role in changing orientation that seperates data classes.

→ Bias is used to shift activation function to left or right.

→ Bias initialization is automatic by neural network

Sigmoid Activation function (for binary classification)

↳ $\text{sigmoid} = \frac{1}{1+e^{-y}} = \frac{1}{1+e^{-(\sum x_i w_i + b)}}$ $\begin{cases} \geq 0.5 & 1 \\ < 0.5 & 0 \end{cases}$

① Forward propogation

Input → Assign weights → Input x weight + bias

Actual value y ⇓

$\hat{y}$ output ← Activation fn $(\sum x_i w_i + b)$

$\text{Loss} := y - \hat{y}$ (Reduce Loss to close to zero)

for this we need to update wt.

This is done using back propogation. (using optimizers)

② Back propogation

update wt's using optimizers to reduce loss. $(y - \hat{y})$

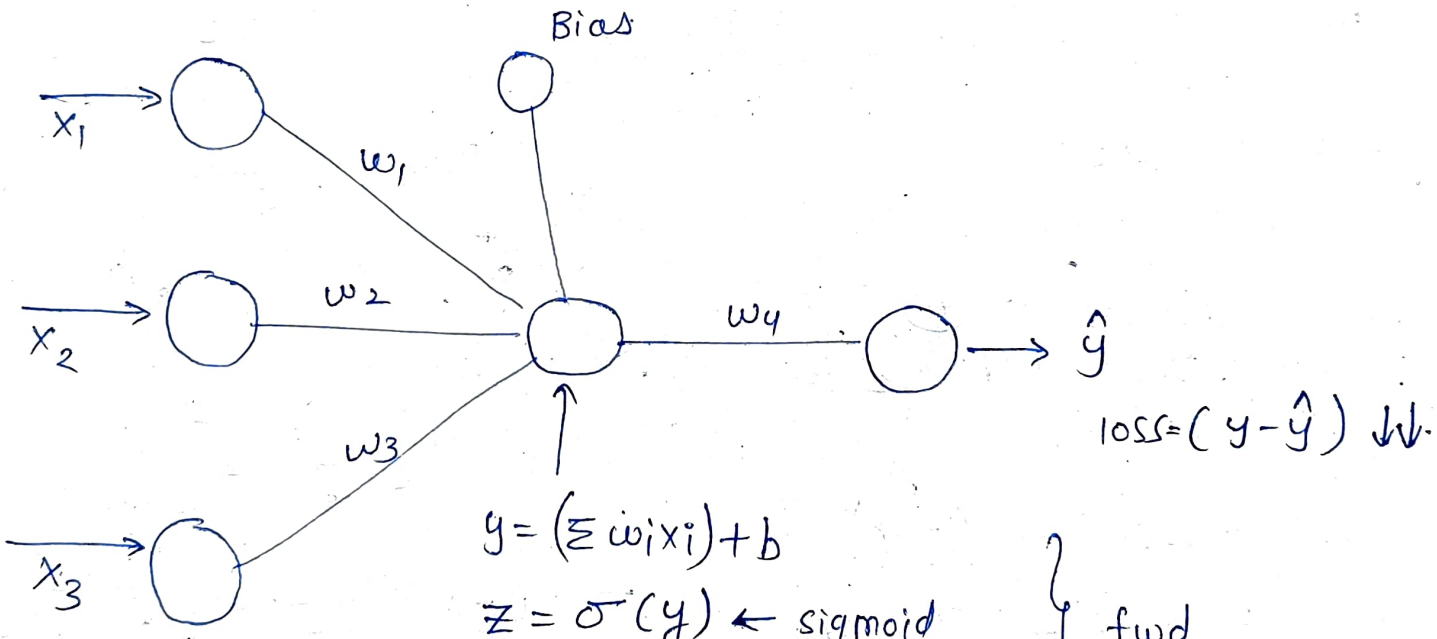
1) I/p layers
2) weights
3) Bias — Bias gets added in each hidden layer.
4) Activation function

} forward propogation.

5) loss function $f(y-\hat{y})$ ↓↓↓
6) optimizers
7) update the wt

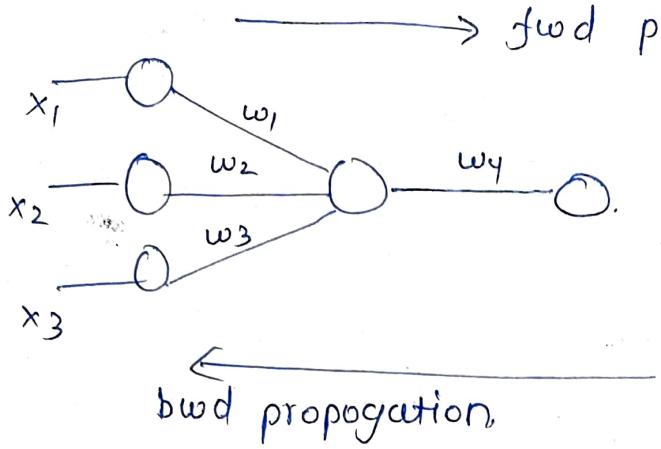
} Backward propogation

F



→ we need to update wts using back propogation

① weight updation formula.
② chain rule of differentiation.

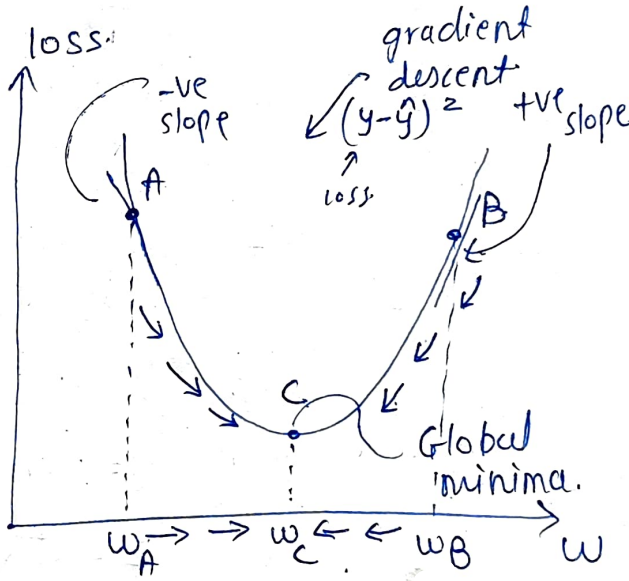


* **weight updation formula**

$$w_{new} = w_{old} - \eta \frac{\partial Loss}{\partial w_{old}}$$

where η = learning rate

$\frac{\partial h}{\partial w_{old}}$ = slope



1) when we are at (A)
we want to reach w_c for global minima

-ve slope

$w_{new} = w_{old} - \eta\,(-ve)$

$= w_{old} + \eta\,(const)$

$\therefore w_{new} > w_{old}$ ⇒ updating this multiple time we will reach w_c.

2) At (B) slope = +ve

$w_{new} = w_{old} - \eta\,(+ve)$

$= w_{old} - \eta\,(const)$

$\therefore w_{new} < w_{old}$ ⇒ updating this multiple times we will reach global minima.

* Importance of learning rate (η)

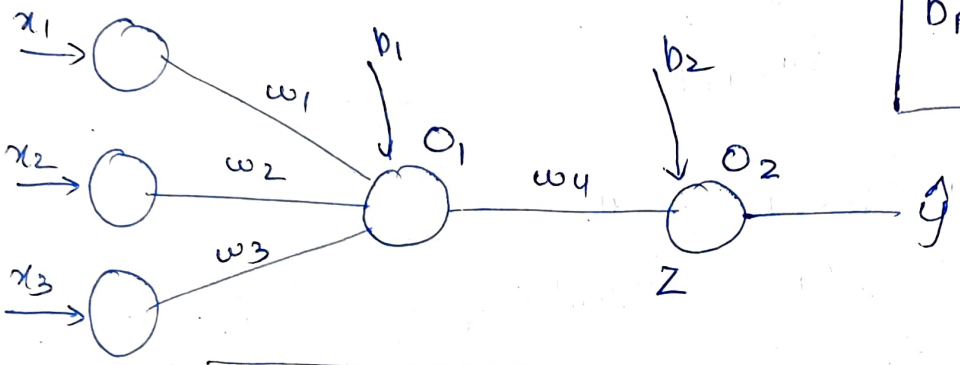
1) It should be a smaller number. So that we can slowly reach the global minima.

2) For larger learning rate we may not reach global minima.

Bias updation formula.

$$b_{new} = b_{old} - \eta \frac{\partial L}{\partial b_{old}}$$

② chain rule of differentiation



$$w_{4\,new} = w_{4\,old} - \eta \frac{\partial loss}{\partial w_{4\,old}}$$

$$\frac{\partial L}{\partial w_{4\,old}} = \frac{\partial L}{\partial O_2} \times \frac{\partial O_2}{\partial w_4} \leftarrow \text{chain rule}$$

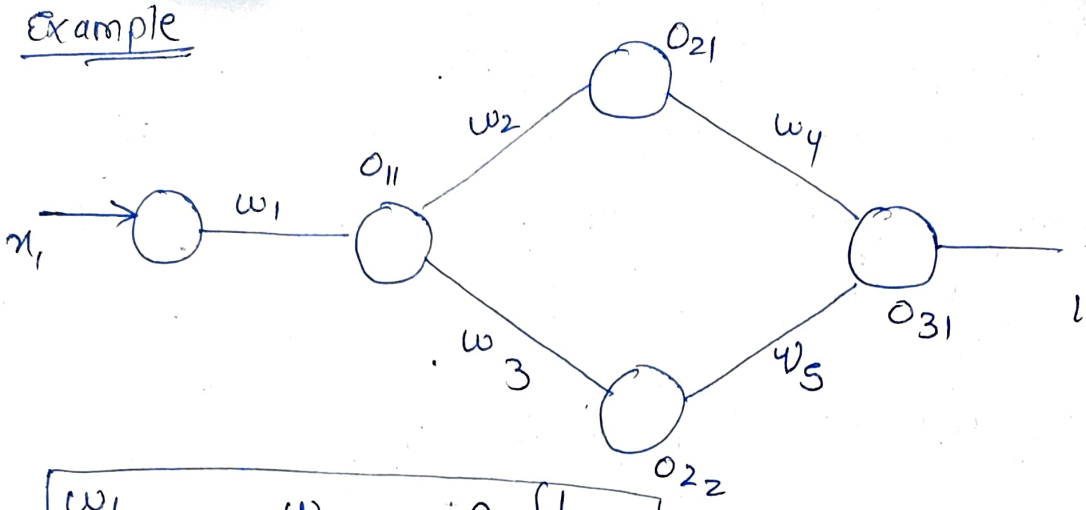
$$w_{1\,new} = w_{1\,old} - \eta \frac{\partial L}{\partial w_{1\,old}}$$

$$\frac{\partial L}{\partial w_{1\,old}} = \frac{\partial L}{\partial O_2} \times \frac{\partial O_2}{\partial O_1} \times \frac{\partial O_1}{\partial w_{1\,old}} \leftarrow \text{chain rule.}$$

$$\Downarrow$$

$$\frac{\partial O_2}{\partial w_4} \times \frac{\partial w_4}{\partial O_1}$$

Example



$$w_{1\,new} = w_{1\,old} - \eta \frac{\delta L}{\delta w_{1\,old}}$$

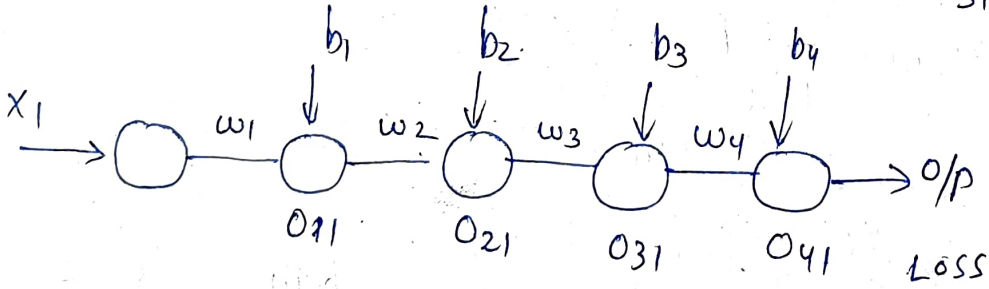
$$\frac{\delta L}{\delta w_{1\,old}} = \left[\frac{\delta L}{\delta O_{31}} \times \frac{\delta O_{31}}{\delta O_{21}} \times \frac{\delta O_{21}}{\delta O_{11}} \times \frac{\delta O_{11}}{\delta w_{1\,old}}\right] + \left[\frac{\delta L}{\delta O_{31}} \times \frac{\delta O_{31}}{\delta O_{22}} \times \frac{\delta O_{22}}{\delta O_{11}} \times \frac{\delta O_{11}}{\delta w_{1\,old}}\right]$$

← chain rule of derivatives.

* Vanishing gradient problem

consider a very deep neural network

sigmoid Activation



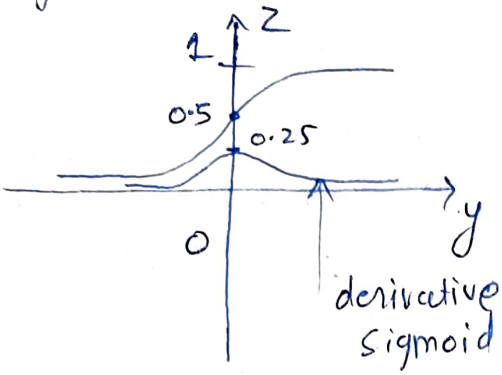
$$\text{Loss} = \frac{1}{2}(y - \hat{y})^2$$

↑ MSE

$$w_{1\,new} = w_{1\,old} - \eta \frac{\delta L}{\delta w_{1\,old}}$$

$$\frac{\delta L}{\delta w_{1\,old}} = \frac{\delta L}{\delta O_{41}} \times \frac{\delta O_{41}}{\delta O_{31}} \times \frac{\delta O_{31}}{\delta O_{21}} \times \frac{\delta O_{21}}{\delta O_{11}} \times \frac{\delta O_{11}}{\delta w_{1\,old}}$$

Sigmoid Activation.



derivative of sigmoid curve.

$$z = \frac{1}{1+e^{-x}} \quad \begin{matrix} \geq 0.5 = 1 \\ < 0.5 = 0 \end{matrix}$$

$$0 < f(\text{sigmoid}) \leq 0.25$$

↑ derivative condition.

$$\frac{\partial L}{\partial w_{old}} = 0.25 \times 0.15 \times 0.10 \times 0.05 \times 0.02$$

derivative value decreases

= very small value.

$$w_{new} = w_{old} - \eta \times (\text{small number})$$

↑ Also a small NO.

$$\therefore \boxed{w_{new} \approx w_{old}}$$ very small

↑ the wt will not get updated or very very slowly updated

This is called vanishing gradient problem

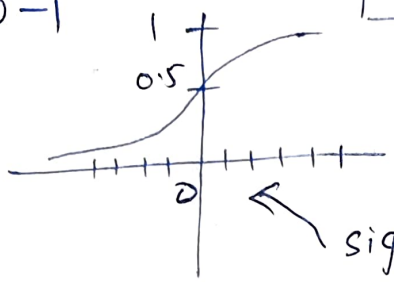
solution is change Activation function

1. sigmoid activation function
2. Tanh
3. ReLU
4. Leaky ReLU
5. PreLU

① sigmoid

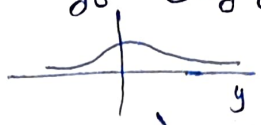
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

0-1



sigmoid is not zero centered curve

So $0 < \int \sigma(x) \leq 0.25$



0-0.25

gradient vanishing problem.

* wt updation is very easy in case of zero centered curves

Adv

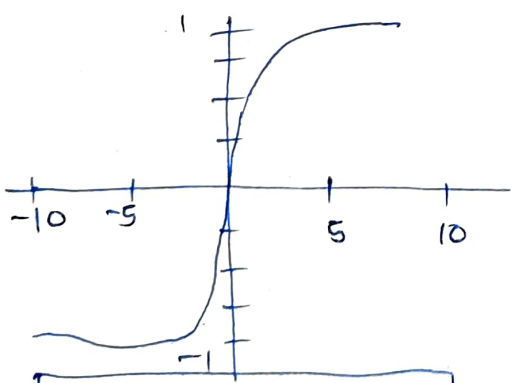
1) smooth gradient, prevents jumps in o/p values
2) since o/p blw 0 and 1, it normalises o/p of each neuron.
3) clear prediction very close to 0 or 1.

Disadv

1) prone to gradient vanishing → we cannot create deep neural network.
2) Function o/p is not zero centred, wt updation issue
3) Sigmoid has exponential operation, which is slower for computers to calculate.

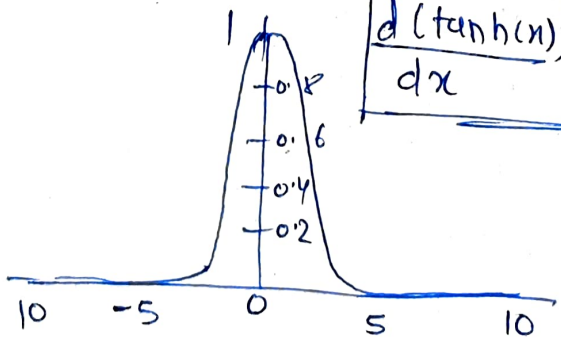
② tanh function (hyperbolic tangent function)

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



tanh(x) => -1 to 1

$$\frac{d(\tanh(x))}{dx} \Rightarrow 0 \text{ to } 1$$



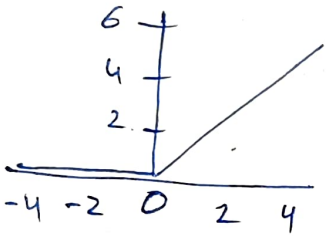
Adv 1) zero centric output.

Disadv 1) still can create vanishing gradient problem.

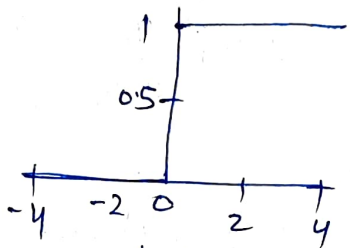
for binary classification — tanh ← hidden layer
sigmoid ← o/p layer.

(3) ReLU function.

↑ Rectified Linear unit.



ReLU(x)



$\frac{d\,ReLU(x)}{dx}$ ← not differentiable at zero.

↳ $ReLU = max(0, x)$

→ not zero centric

* when $\frac{\partial\, Relu}{\partial\, x} = 0$ ↑ neuron is dead. This is a problem.

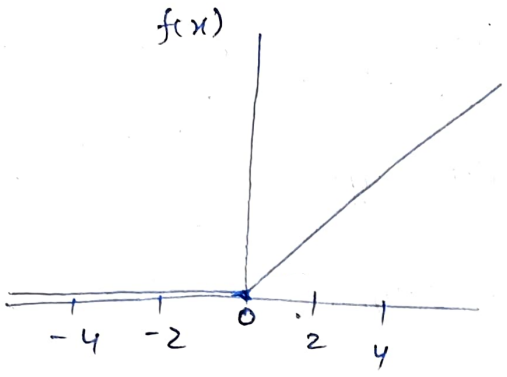
Adv 1) no vanishing gradient
2) Faster calculation
3) when input is +ve, no gradient saturation problem.

Disadv: 1) not a zero centric function since o/p is either 0 or 1.

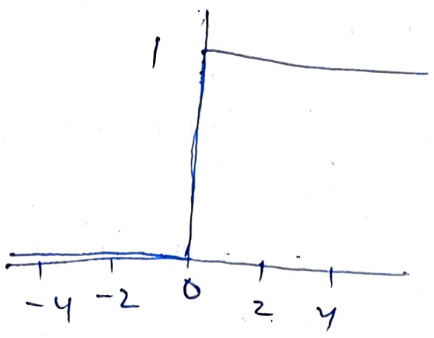
2) when input is -ve, ReLU is inactive completely.
=> ReLU is dead for input <= 0.
↳ gradient = zero } same problem as sigmoid and tanh.

4) Leaky ReLU function

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



$d(f(x))/dx$



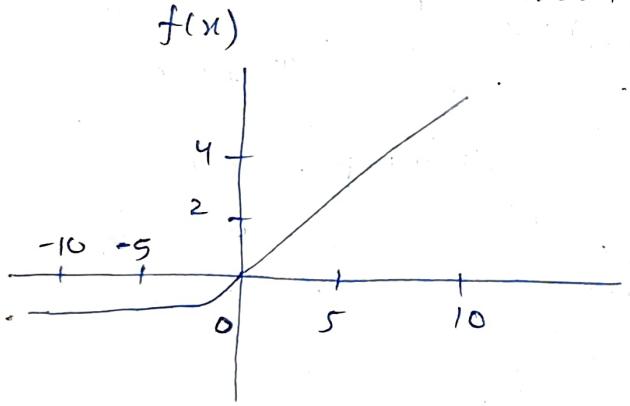
Adv: 1) solves dead neuron problem in ReLU.

5) Parametric ReLU function.

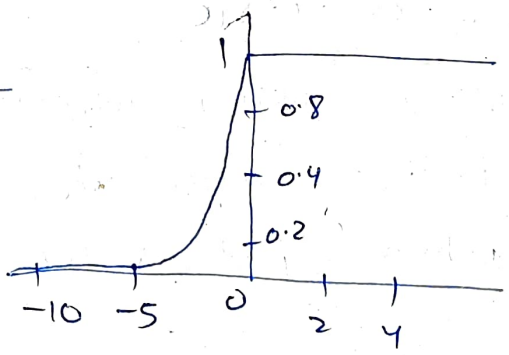
$$f(x) = \begin{cases} y_i & \text{if } y_i > 0 \\ a_i y_i & \text{if } y_i \leq 0 \end{cases}$$

a=0 f(x) → ReLU
a>0 f(x) → Leaky ReLU
a is learnable → PReLU.

6) ELU (Exponential Linear units) function



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



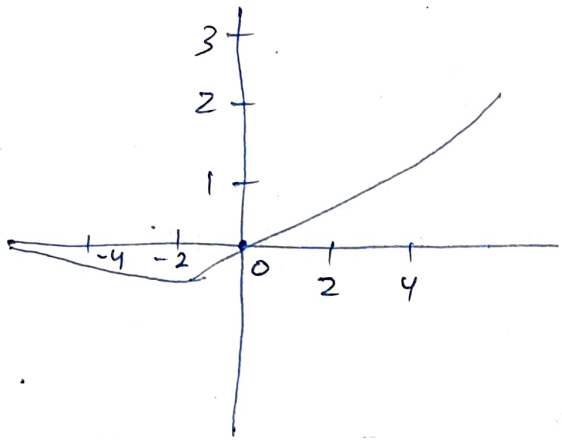
$\frac{df(x)}{dx}$

It solves problems of ReLU.

Adv: 1) No dead ReLU issue
2) zero centred.
3) Better the ReLU (proved)

Disadv 1) computation heavy (due to exponential calcn).

8). SWISH (self gated) function



$f(x) = x \times \text{sigmoid}(x)$

↳ gating in LSTM

9) Max out

$$f(x) = \max(w_1^T x + b_1, w_2^T x + b_2).$$

It is generalised version of ReLU and Leaky ReLU.

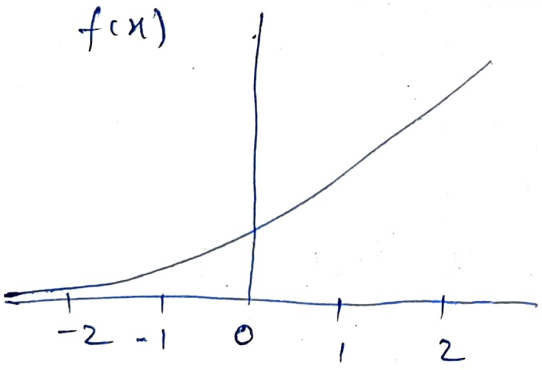
for reLU $w_1, b_1 = 0$

for Leaky reLU $w_1, b_1 = 0, w_2 = 0.01$

It is a learnable function.

10) soft plus.

$$f(x) = \ln(1+e^x).$$



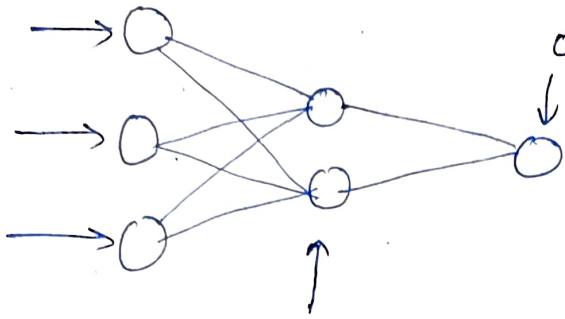
→ differentiable throughout.

→ similar to ReLU, but relatively smooth.

→ 0 to $+\infty$ } Accepted range

Technique which activation function we should use?

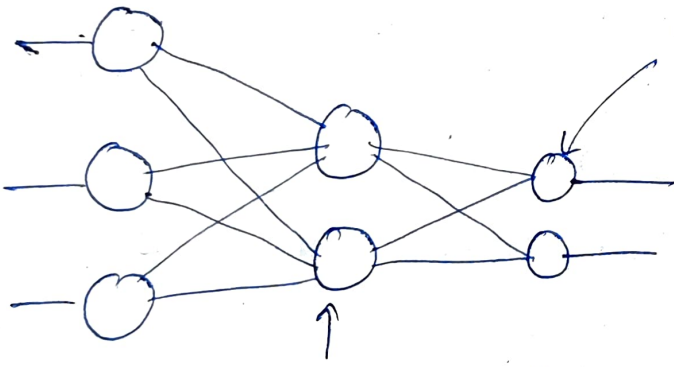
Say Binary classification



o/p layer (use sigmoid).

Hidden Layer always use ReLU } if convergence is not happening use PReLU, ELU.

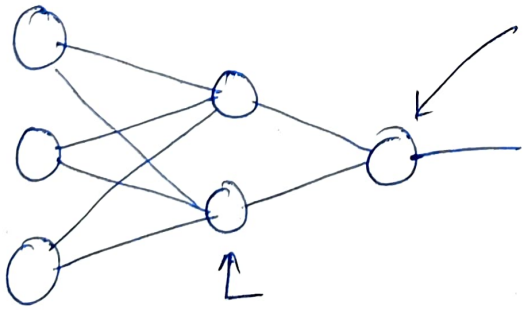
multiclass classification



use softmax activation fn.

use ReLU } if convergence not happening use PReLU, ELU.

Regression



use linear activation fn.
↳ Also there is a seperate loss function.

use ReLU or any variation of ReLU

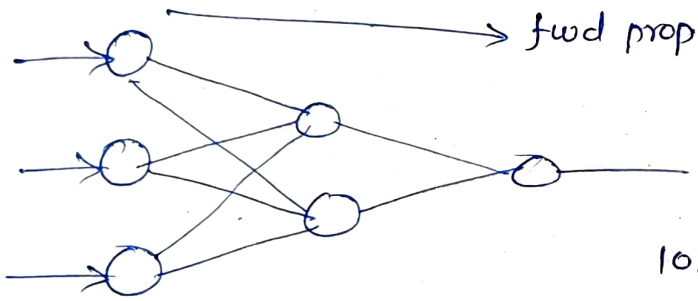
* Loss functions

DL (ANN)

- Regression
  1) MSE
  2) MAE
  3) Huber loss
- classification
  1) cross entropy
     - Binary CE (Binary classifn)
     - categorical CE (multiclass classifn)

* Loss V/S cost function

* 100 recs in dataset.



$$\text{loss} = \frac{1}{2}(y-\hat{y})^2 \leftarrow \text{single value.}$$

$$\Downarrow$$

cost function for batch input

$$\text{cost fn} = \frac{1}{2}\sum_{i=1}^{n}(y-\hat{y})^2.$$

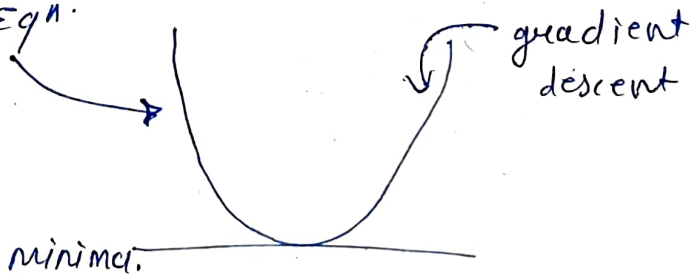
(A) Regression

(1) MEAN squared error

$$\text{Loss fn} = \frac{1}{2}(y-\hat{y})^2$$

↑ Quadratic eqn.

$$\boxed{\text{cost fn} = \frac{1}{2n}\sum_{i=1}^{n}(y-\hat{y})^2}$$



Adv

1) It is differentiable
2) It has only one local/global minima.
3) It converges faster.

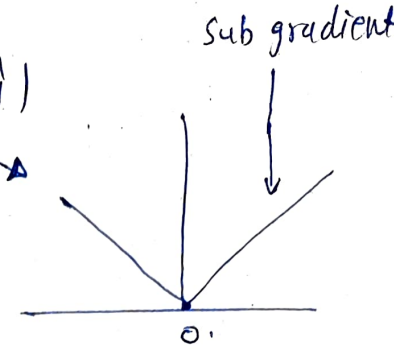
Disadv

1) not robust to outliers. (we are squaring the error) ∴ error penalising.

② Mean Absolute Error

$$LF = \frac{1}{2}|y-\hat{y}|$$

$$CF = \frac{1}{2n}\sum_{i=1}^{n}|y-\hat{y}|$$



Adv:

1) Robust to outliers

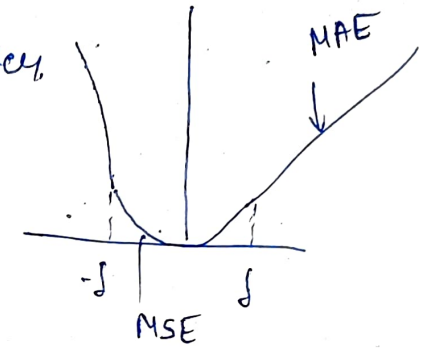
disadv

1) not diff at $x=0$
2) Time consuming due to sub gradient

③ Huber loss (MSE + MAE)

$$LF = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & |y-\hat{y}| \le \delta \\ \delta|y-\hat{y}| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

Hyper parameter



④ classification

① Binary cross entropy

Log req.

log loss.

$$\boxed{loss = -y \times \log(\hat{y}) - (1-y) \times \log(1-\hat{y})}$$

$$loss = \begin{cases} -\log(1-\hat{y}) & \text{if } y=0 \\ -\log(\hat{y}) & \text{if } y=1 \end{cases}$$

} we use sigmoid in Last layer for $\hat{y}$ calculation.

(2) Categorical cross entropy

	f_1	f_2	f_3	o/p
$i=1$	2	3	4	good
$i=2$	5	6	7	bad
$i=3$	8	9	10	neutral.

↓ OHE

	$j=1$	$j=2$	$j=3$
	Good	Bad	Neutral
	1	0	0
	0	1	0
	0	0	1

$$L(x_i, y_i) = -\sum_{j=1}^{c} y_{ij} \times \ln(\hat{y}_{ij})$$

c → no. of categories.
i = Row
j = column of o/p

$$y_i = [y_{i1}, y_{i2}, y_{i3}, \ldots y_{ic}]$$

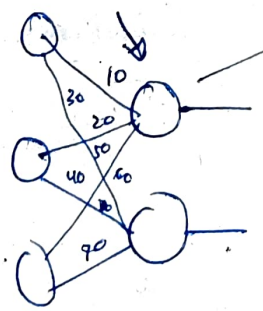
$$y_{ij} = \begin{cases} 1, & \text{if element in class.} \\ 0, & \text{otherwise} \end{cases}$$

$\hat{y}_{ij}$ ⇒ use softmax Activation function. — o/p layer

$$\sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

HL

o/p



$$\frac{e^{10}}{e^{10+20+\cdots+70}} = 0.4$$

Prob

+ always one

→ 0.6

conclusion

	HL	O/P	Loss
multiclass →	ReLU, softmax		cat C.E
Binary →	ReLU, sigmoid		Binary CE
Regression →	ReLU, Linear Activation		MSE, MAE, huberloss.

* Optimizers

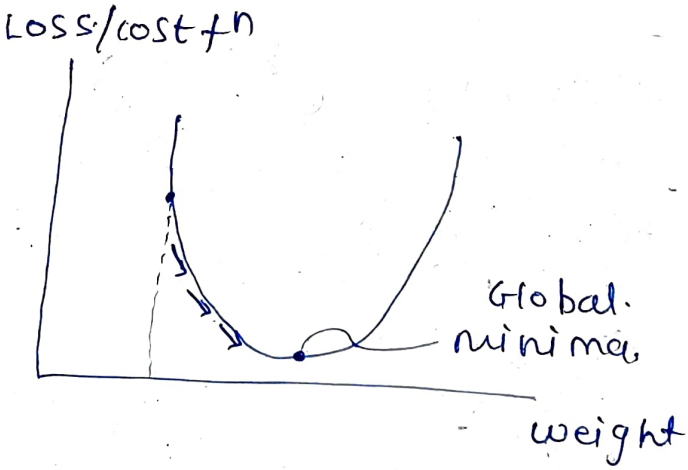
1) Gradient descent
2) SGD (stochastic GD)
3) Mini Batch SGD.
4) SGD with momentum
5) ADgrad
6) RMS PROP
7) Adam optimizer.

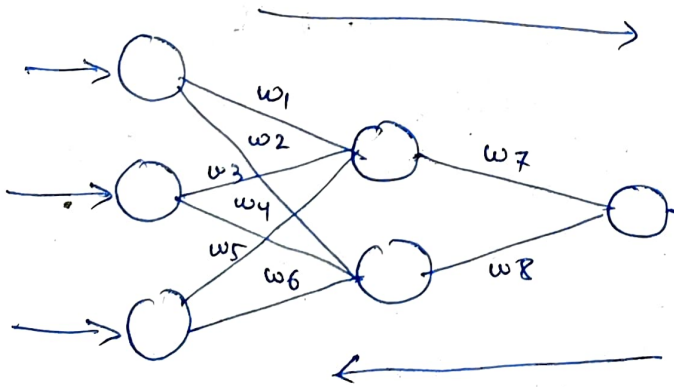
① Gradient Descent

wt updation formula

$$w_{new} = w_{old} - \eta \frac{\delta L}{\delta w_{old}}$$

$\eta \equiv$ learning rate





O/p $\hat{y}$

MSE

$$\text{Cost fun} = \frac{1}{2n} \sum_{i=1}^{n} (y - \hat{y})^2$$

we use optimizers in back propogation to update wt so that we can minimise loss/error.

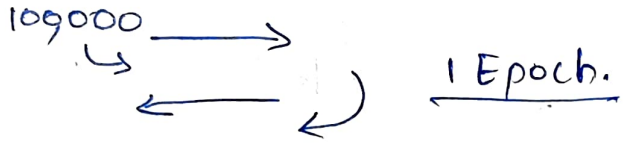
Disadv

1) It is resourse extensive technique (more ram required)

Epoch

It means traing neural network with full dataset for one cycle (Fwd and bwd propogation)

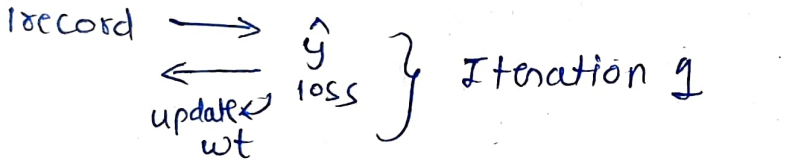
Ex: 100,000 data points

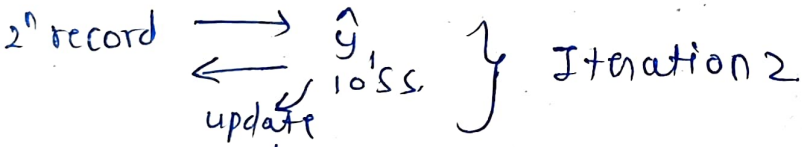


② Stochastic Gradient Descent (SGD)

1 million records in dataset

Epoch 1





1 million Iteration.

Say we need 10 Epochs to train model

=> 10 Epoch x 1 million Iterations => Total 10 million runs

Disadv

1) converge is very slow.
2) Time complexity high.

Adv:

1) low memory required.

3) Mini - Batch SGD

Batch size = 1000 recs

1 million records => No. of Iterations = $\frac{1 \text{ million}}{1000}$

= 1000 Iterations

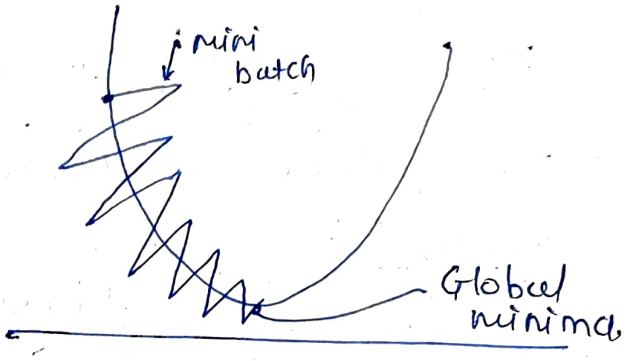
Epoch 1

1000 rec ⟶ ŷ ⟵ cost fn } Iteration 1

1000 records ⟶ ŷ ⟵ } Iteration 2

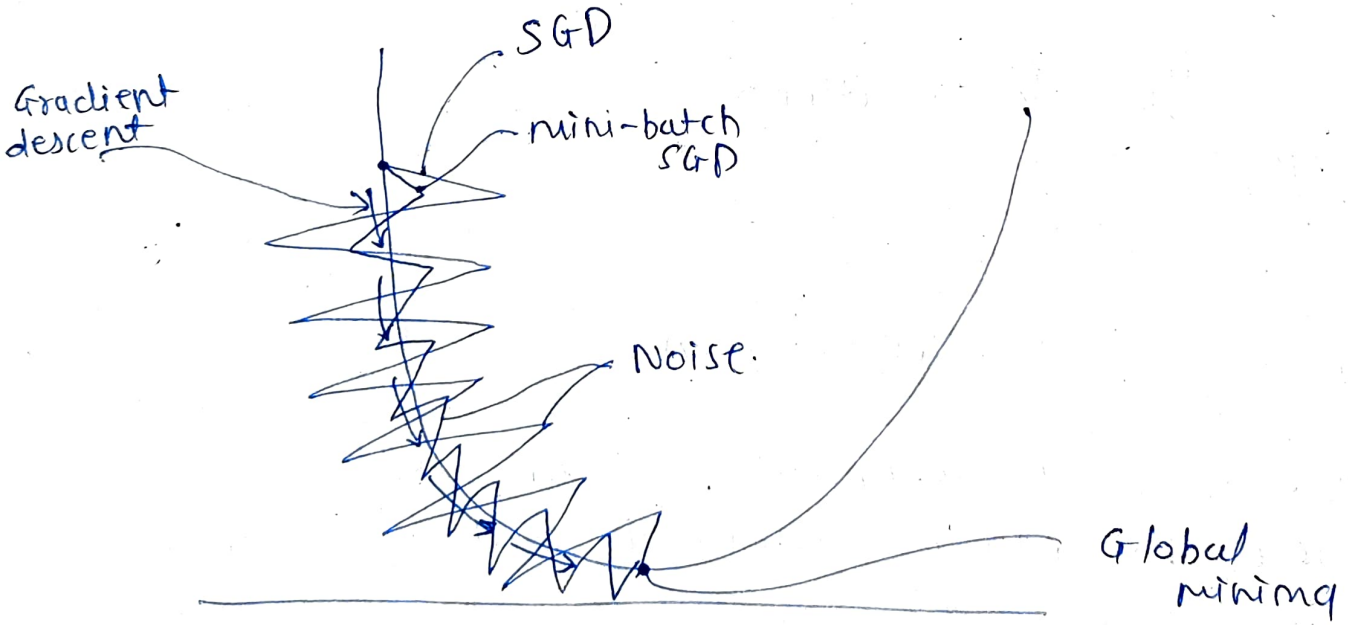
⋮

1000 Iteration



Adv

1) Less Resource intensive
2) Better convergence
3) Better / improved Time complexity.



noise => SGD > mini-batch SGD.

↓ more noise

↓ less noise.

To remove noise we use momentum.

4) SGD with momentum
↑ mini batch.

momentum is used to eliminate noise using moving avgs. exponential.

Exponential weighted Average?
↳ This is also used in Time series

$$w_{new} = w_{old} - \eta \frac{\delta L}{\delta w_{old}} \qquad b_{new} = b_{old} - \eta \frac{\delta L}{\delta b_{old}}$$

$$\boxed{w_t = w_{t-1} - \eta \frac{\delta L}{\delta w_{t-1}}}$$ $t \equiv$ time

Exponential weighted average → for smoothening the noise.

Time	t_1	t_2	t_3	....	t_n
value(v).	a_1	a_2	a_3	....	a_n

value at t_1, $V_{t_1} = a_1$

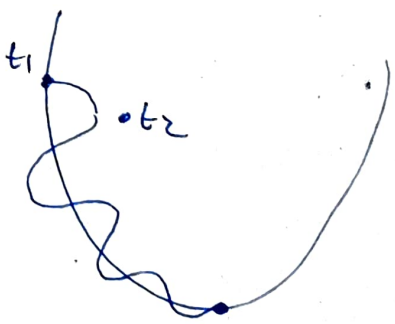
$\beta \equiv$ hyper parameter
↳ 0 to 1

$$V_{t_2} = \beta \times V_{t_1} + (1-\beta) \times a_2$$

$$V_{t_3} = \beta V_{t_2} + (1-\beta) a_3$$

⋮

$$\boxed{V_{t_n} = \beta V_{t_{n-1}} + (1-\beta) a_n}$$

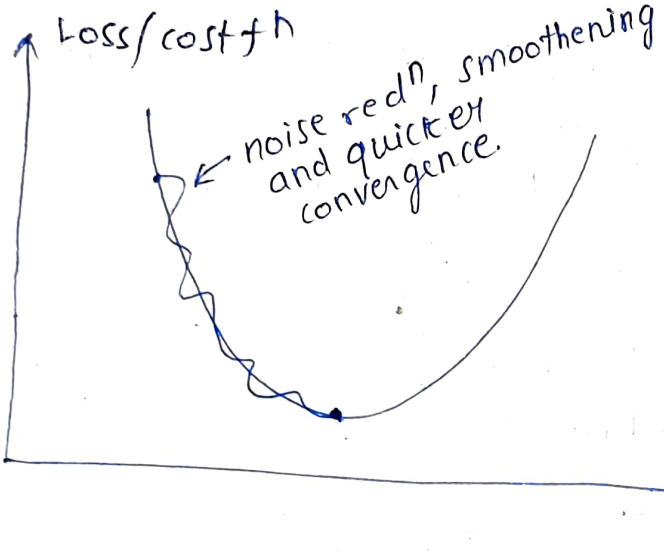


↑ This is applied in $\frac{\delta L}{\delta w}$.

Now

$$w_t = w_{t-1} - \eta V_{dw_t}$$

where $$V_{dw_t} = \beta \times V_{dw_{t-1}} + (1-\beta) \times \frac{\delta L}{\delta w_{t-1}}$$



Adv
→ smoothening of noise
→ Reducing the noise
→. Quicker convergence

⑤ Adagrad ⇒ Adaptive Gradient descent

$$w_t = w_{t-1} - \eta \frac{\delta L}{\delta w_{t-1}}$$

η = fixed in all above optimizers.

Replace with η'

$$w_t = w_{t-1} - \eta' \frac{\delta L}{\delta w_{t-1}}$$

$$\eta' = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

ϵ → small no. to avoid zero division error

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\delta L}{\delta w_t}\right)^2$$

increases with time ie with every epoch α_t increases.

Requirement:

Initially large learning rate will make quicker convergence initially, but as we approach global minima learning rate value should reduce so that we do not miss global minima.

as the time increase $\Sigma \frac{\delta L}{\delta w_t}$ will increase

∴ α_t will increase

∴ η' will decrease as we will reach near to global minima.

∴ we have made learning rate adaptive.

Disadv

for deep neural network

α_t value will become large

∴ η' will almost become negligible

∴ our model will be very slow or stop converging to global minima. ie $w_t \approx w_{t-1}$

6) Ada delta and RMSprop

$$\eta' = \frac{\eta}{\sqrt{sdw + \epsilon}}$$

initially $sdw = 0$

$$sdw_t = \beta\, sdw_{t-1} + (1-\beta)\left(\frac{\delta L}{\delta w_{t-1}}\right)^2$$

↑ exponential wt average

∴ learning rate will decrease slowly. by implementing sdw. ie exponential weighted average.

→ Here smoothening is missing ie Vdw.

⑦ Adam optimizer (momentum + RMS prop intution)
Vdw ↖ Adaptive learning rate
Sdw.

Initially $Vdw, Vd_b, Sdw, Sd_b = 0$.

$$w_t = w_{t-1} - \eta' Vdw$$

$$b_t = b_{t-1} - \eta' Vd_b$$

$$\eta' = \frac{\eta}{\sqrt{Sdw + \epsilon}}$$

$$Vdw_t = \beta \times Vdw_{t-1} + (1-\beta)\frac{\delta L}{\delta w_{t-1}}$$

$$Sdw_t = \beta\, Sdw_{t-1} + (1-\beta)\left(\frac{\delta L}{\delta w_{t-1}}\right)^2$$

Adv

1) Smoothening the Gradient descent/ noise
2) Learning rate is adaptive
3) Quicker convergence
4) Less Resource intensive

* For which algorithms feature scaling is req?

Req	Not Req
ANN	Dt
LR	RF
Linear Reg	XGBoost
KNN	Adaboost
k-means	

Distance based, optimizers involved } for quicker convergence