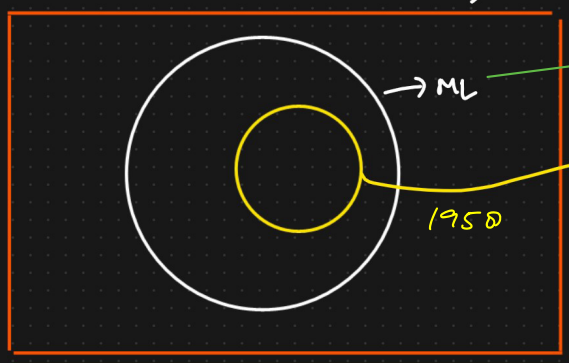
Deep Learning → recognizes patterns in the data and then makes predictions based on Multilayered Neural Network

Imp !! (mimics human brain)



Multilayered ⇒ Neural Networks

⇓

Mimic the Human Brain

ML - CPU
DL - GPU

Deep Learning → huge amounts of Data

1. ANN → Artificial Neural N/W → Classification, Regression
2. CNN → Convolutional Neural N/W → I/P : Images, Video frames → RCNN, MASKED RCNN, Detection, YOLO V5, V6, V7 (Computer Vision, Objection Detection ⇑)
3. RNN → Recurrent Neural N/W → NLP → NLP, Time Series

RNN → Sequential Data → I/P: Text, Time Series

⇓

{ Word Embedding, LSTM RNN, GRU RNN, Bidirectional LSTM RNN, Encoder Decoder, Transformers, BERT }

FRAMEWORK

TENSORFLOW KERAS

End to End Project

LSTM = Long Short Term Memory → precise & slow

GRU = Gated Recurrent Unit → Simple & Fast

Tensorflow → Deep learning library by Google

↳ Keras → API on top of Tensorflow

PyTorch → Deep learning library by Meta

② Why Deep Learning Is Becoming popular ?

2005 ⟶ Facebook, Youtube, Whatsapp, Linked, Twitter } ⇒ Social Media platform

2011-2012

⇓

DATA WAS GENERATED Exponentially

Image, Text, Documents, Videos

↳ Big DATA ⟶ HADOOP → unstructured DATA

⇓

2011 ⇒ Cloudera, Hortonworks

Netflix → movie streaming platform

⇓

Recommendation System

⇓

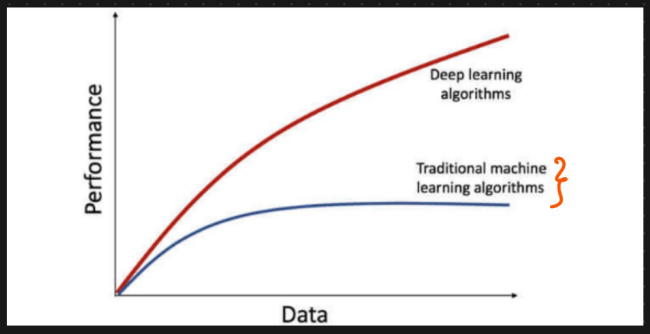
Increase Revenue

2011-2012 :

① Hardware Requirement ⟶ GPU's cost↓↓↓ → NVidia GPU Priced↓↓

TRAIN MODELS

② Huge amount of data is getting generated → Deep Learning Models Perform well



③ Deep learning is been used in Many Domains

1. Medical
2. ECommerce
3. Retails
4. Marketing . . . . .

④ Frameworks Open source

Community size ↑↑

Tensorflow → Google

Pytorch → Facebook

More Research

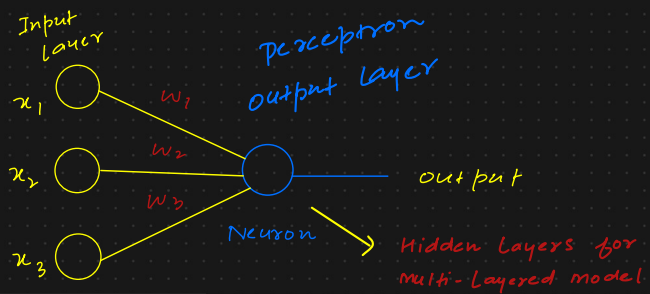
PERCEPTRON; Simplest Model that Mimics Brain

Simplest form of Neural Network Unit.

🧠 **How It Works:**

1. **Inputs** → Like features (e.g., height, weight).
2. **Weights** → Importance of each input.
3. **Summation** → Adds the weighted inputs.
4. **Activation Function** → Decides the output (e.g., 0 or 1).

🧮 Output = Activation($(\text{Input}_1 \times \text{Weight}_1) + (\text{Input}_2 \times \text{Weight}_2) + \ldots + \text{Bias}$)



Imp.

🧠 **Quick Recap:**

Type	Hidden Layer?	Example
Single-layer NN	❌ No	Perceptron
Multi-layer NN (MLP)	✅ Yes	Deep Learning models

Imp

Steps

Input → weight → Summation → Activation functions → output

V.V. Imp.

- Neurons don't "think" — **weights decide how much each input matters**
- Weights are adjusted through **training** (using backpropagation)
- This is how deep learning models **learn patterns**

Summation $(\text{Input}_1 \times w_1 + I_2 \times w_2 + \text{Bias})$

So, the entire process is called Forward Propagation

Actual Equation

$y = \sum x_i w_i + \text{bias} \;\leftarrow\; mx + c$ ← Think of it as bias.

↓

Important so we start from some default value and not just zero

③ Perceptron [Artificial Neuron or Neural Network Unit]

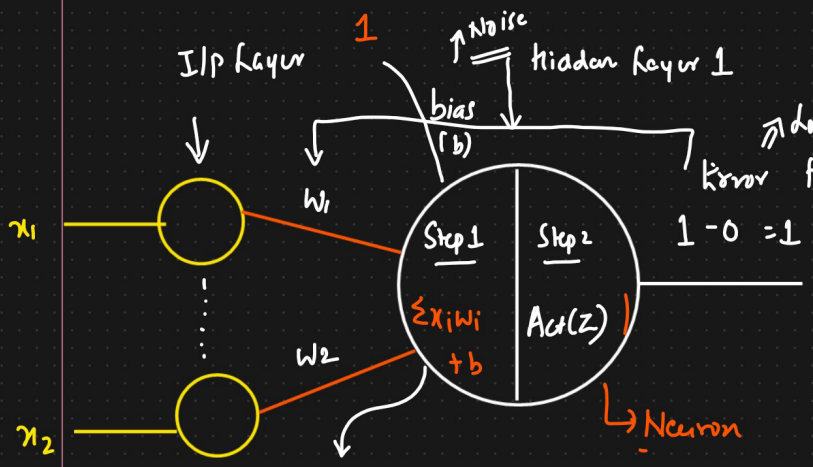
① Input layer ✓
→ No hidden layer in perceptron

② Hidden layer ✓

③ Weights ✓

④ Activation function ✓

[Single layered NN]
⇓
Binary classifier



DATASET

x_1 IQ	x_2 No. of study hours	O/p Pass/Fail
→ 95	3	0
→ 110	4	1
→ 100	5	1

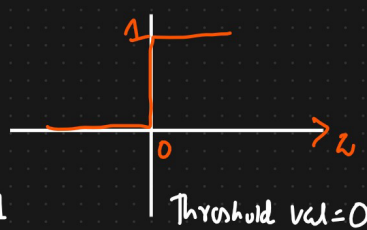
Imp ! $Z = w_1x_1 + w_2x_2 + b \rightarrow \beta_0x_1 + \beta_1x_1$ similar

$$Z = \sum_{i=1}^{n} x_i w_i + b$$

Activation function

Transform the output between 0 to 1 / -1 to 1

Step Function

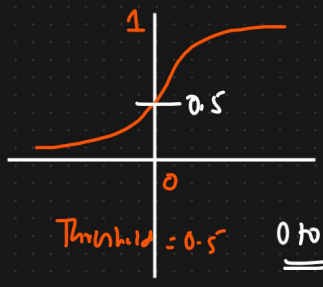


0 to 1

Threshold val = 0

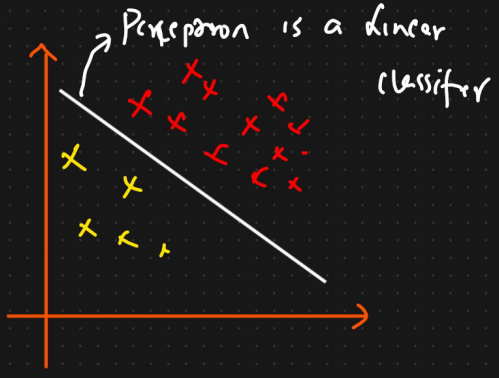
$$\begin{cases} 0 & Z \le 0 \\ 1 & Z > 0 \end{cases}$$

Sigmoid Function



Threshold = 0.5 — 0 to 1

$$\begin{cases} 1 & Z > 0.5 \\ 0 & Z \le 0.5 \end{cases}$$



Step 1

$$Z = \sum_{i=1}^{n} w_i x_i + b \rightarrow \text{Bias}$$

$$Z = b + w_1x_1 + w_2x_2 + w_3x_3 + \dots + w_nx_n$$

$$y = mx + c$$

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 \; - - - + \beta_n x_n$$

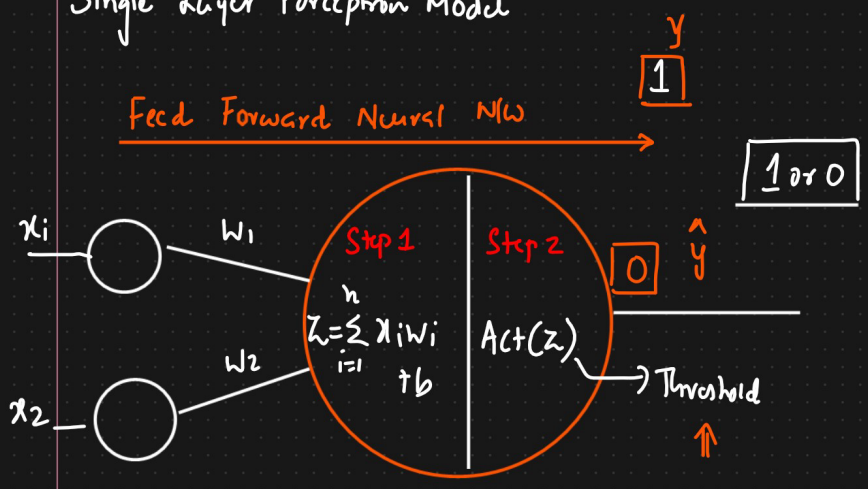
linear problem Statement

Perceptron Models

- Single Layer Perceptron Model
- Multi Layered Perceptron Model [ANN]

Single Layer Perceptron Model

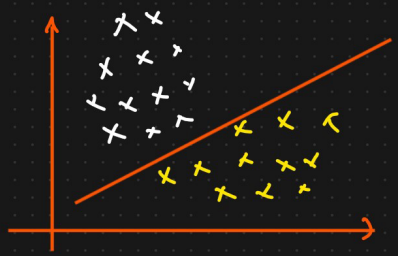
Feed Forward Neural NW



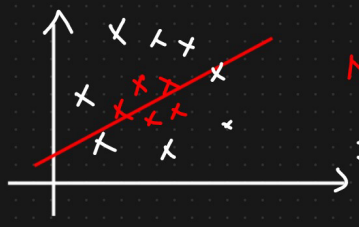
Multi Layered Perceptron Model

1. Forward Propagation
2. Backward "
3. Loss functions
4. Activation functions
5. Optimizers

Linear Separable problem



Binary classification ⇓ linearly separable



Non linear separable ⇒ Multi Layered NN

Activation function

$$\frac{1}{e^{-y}} = \frac{1}{e^{-(x_i w_i + b)}}$$ → gives output in 0 to 1.

$\geq 0.5 = 1$

$< 0.5 = 0$

Summary

Deep learning → mimic human brain with the help of neural networks

working ;

Input Data → Calculate weights → Add weights + Bias (Sum)

$\sum x_i w_i + b$

Activation functions to get probability → Output

pros / cons →

- More Accurate
- solves complex problems
- work on huge Data
- Requires GPU's
- More Time
- Blackbox (Hidden Layers)

Neural Networks

Linear classification
Perceptron

Single layered Neural Network

- Single layer
- No hidden layers
- only linear Tasks
- And / OR Gate
- Layer = 1

LSTM

Multi layered Neural Network

- Multiple layers
- Hidden Layers
- Complex & Non Linears
- Image Classifier / NLP
- Layers ≥ 2

Q1. What is non-linearity concept ? usually Activation functions.

ability of a model to learn complex patterns; like curves, shapes etc. not just simple line relationships.

imp

In short:

> **Non-linearity** gives neural networks the **power to learn complex, real-world relationships.**

Without it, even a deep network would just behave like a **linear regression** model.

Q11. what do you understand by activation functions ?

decides whether neuron should be activated.

- introduces probability & non linearity
- helps model to learn beyond linear data

✅ Activation Function Comparison Cheat Sheet

Activation	Used For	Non-Linearity	Probability Output	Common Use Case
ReLU	Hidden Layers (default)	✅ Yes	❌ No	Enables deep networks to learn patterns
Leaky ReLU	Hidden Layers (ReLU alternative)	✅ Yes	❌ No	Solves "dead ReLU" problem
Sigmoid	Output Layer (Binary Classification)	✅ Yes	✅ Yes	Converts output to probability (0–1)
Softmax	Output Layer (Multi-class Classification)	✅ Yes	✅ Yes	Distributes output as class probabilities
Tanh	Sometimes in Hidden Layers	✅ Yes	❌ No	Like Sigmoid but zero-centered

Forward & Backward propagation

🔵 Forward Propagation (Prediction Phase)

🔹 **Purpose**	**To compute the output (prediction) of the model.**
🔹 Direction	From **input → hidden layers → output**

🔹 What happens?

- Input data is passed through the network.
- At each layer:
  - Weighted sum is calculated: `Z = W·X + b`
  - Activation function is applied: `A = activation(Z)`
- Final output (ŷ) is produced.

| 🔹 Example | Predicting whether an email is spam or not. |

🔴 Backward Propagation (Learning Phase)

🔹 **Purpose**	**To update the weights to reduce error/loss.**
🔹 Direction	From **output → hidden layers → input** (reverse)

🔹 What happens?

- Compute the **loss** between predicted output and actual output.
- Apply **chain rule** of calculus (using gradients):
  - Compute how much each weight contributed to the error.
- Calculate gradients (`dW`, `db`) and update weights using **gradient descent**.

| 🔹 Example | Adjusting model so it better predicts spam in future.|

i/p → weights & zixi+b → o/p → o/p → loss fn → Activators → min. loss

Activation

Gradient Descent update

→ What is ReLU Rectified Linear Unit

✅ ReLU Meaning (In Simple Terms)

- **If the input is positive**, keep it as it is.
- **If the input is negative**, change it to zero.

👉 In short:

> **ReLU(x) = x if x > 0, else 0**

⚠️ Limitation: → Important

- **"Dead Neurons"**: If a neuron always receives negative inputs, it outputs 0 forever and stops learning.
  - To solve this, we use **Leaky ReLU** sometimes.

Summary:

> **ReLU = Let the positive values pass, block the negative ones.**
> It's simple, fast, and the default choice for hidden layers in deep learning.

Q1 What is Leaky ReLU ? Leaks negative values

Keeps neurons alive

🔍 What is Leaky ReLU?

Leaky ReLU is a modified version of **ReLU** that fixes its main problem:

> It **allows a small, non-zero output** when the input is negative.

⚠️ Why do we need it?

Problem with ReLU:

- ReLU outputs **0 for all negative inputs**.
- Some neurons may get stuck and **never activate or learn** — called the **"dead ReLU" problem**.

✅ What Leaky ReLU does:

- For **positive inputs**: same as ReLU → output = input.
- For **negative inputs**: output a **small negative value** (like 0.01 × input) instead of 0.

🧠 Intuition:

Input (x)	ReLU(x)	Leaky ReLU(x)
-3	0	-0.03
0	0	0
2	2	2

✅ Benefits of Leaky ReLU:

Feature
Keeps neurons alive
Allows small gradients
Works better for deep networks

Q1 What is Tanh hyperbolic Tangent

🔍 What is tanh in Deep Learning?

tanh, short for **hyperbolic tangent**, is an **activation function** used in neural networks — especially in hidden layers.

-1 to 1

unlike Softmax

0 to 1

✅ Key Characteristics:

Property	Description
Output Range	-1 to 1
Shape	S-shaped (like sigmoid, but zero-centered)
Non-linearity	✅ Yes – allows the network to learn complex patterns
Used In	Hidden layers (especially in RNNs or older networks)

🧠 Why use tanh?

- Unlike sigmoid (which outputs between 0 and 1), **tanh is centered around 0**.
- This makes optimization **easier and faster**, especially for data with negative values.

Conclusion;

1. I/p layers
2. Weight
3. $\Sigma x_i w_i$ + Bias
4. Activation function

} Forword propagation

5. Loss function $(\hat{y} - y)$
6. Minimize Loss with Optimizers
7. Update weights

} Backword propagation

Imp.

Machine learning = White Box (except Random Forest).

Deep learning = Black Box Models

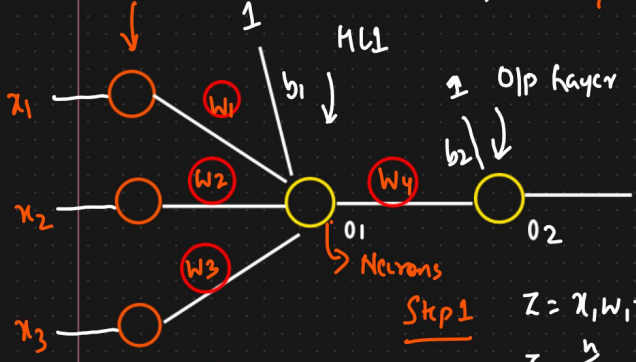
5 Multi layered Perceptron Model [Artificial Neural N/w]

① Forward Propogation
② Backward Propogation → Geoffrey Hinton →

③ Loss function
④ Optimizers ✓ Gradient Descent.
⑤ Activation function.

Multi layered NN

I/P layer — Forward Propogation → 2 layered Neural N/w



Backward Propogation.

Loss function = (1 − 0.51129) ≈ 0.49

x_1	x_2	x_3	Pass/Fail
IQ	Study hours	Play hours	O/p
95	4	4	1
100	5	2	1 ←
95	2	7	0

Error

Step 1 $z = x_1w_1 + x_2w_2 + x_3w + b_1$

$z = \sum_{i=1}^{n} w_i^T x_i + b$

① Forward Propogation 2: Activation(z)

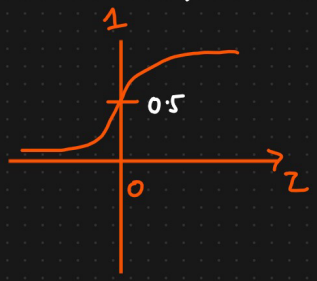
W_1 W_2 W_3 [0.01, 0.02, 0.03] [0.001]

Hidden Layer 1

Step 1: $z = 95 \times 0.01 + 4 \times 0.02 + 4 \times 0.03 + 1 \times 0.01$

$= 1.151$

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



Step 2: Activation (z)

$f(z) = \frac{1}{1+e^{-1.151}} = 0.759$

$O_1 = 0.759$

Hidden Layer 2

$W_4 = 0.02$

$b_2 = 0.03$

Step 1: $z = O_1 * W_4 + b_2$

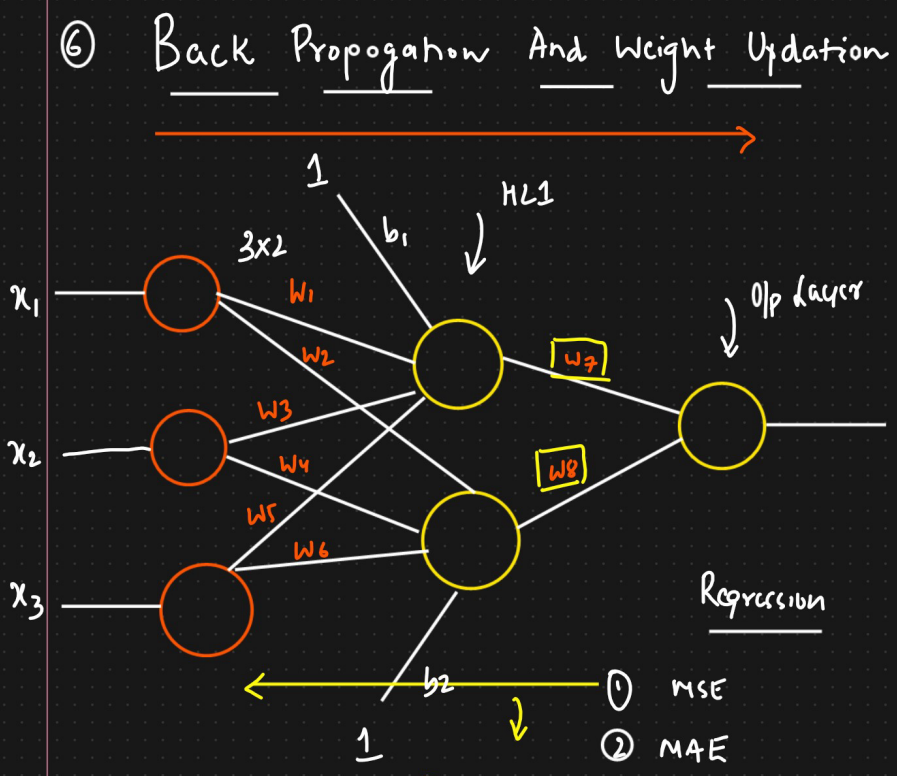
$= 0.759 * 0.02 + 0.03$

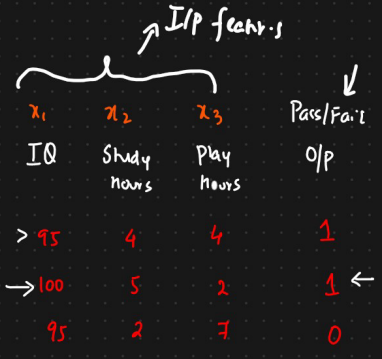
$= 0.045180$

Step 2 : Activation(z) $\frac{1}{1+e^{-(0.04518)}} = 0.511129$

$O_2 = 0.511129 \Rightarrow \hat{y}$

⑥ Back Propogation And Weight Updation Formula





Loss function $(y-\hat{y})^2$

Regression
① MSE
② MAE
③ Huber Loss

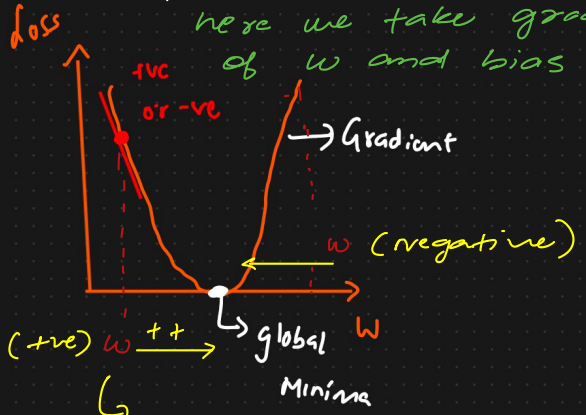
Classification
① Binary Cross Entropy
② Categorical CROSS Entropy

Weight updation Formula

similar to ML but here we take gradient of w and bias also

$$W_{7_{new}} = W_{7_{old}} - \eta \left[\frac{\partial L}{\partial W_{7old}} \right]$$ ← slope

$$W_{8new} = W_{8old} - \eta \left[\frac{\partial L}{\partial W_{8old}} \right]$$ ← Learning Rate



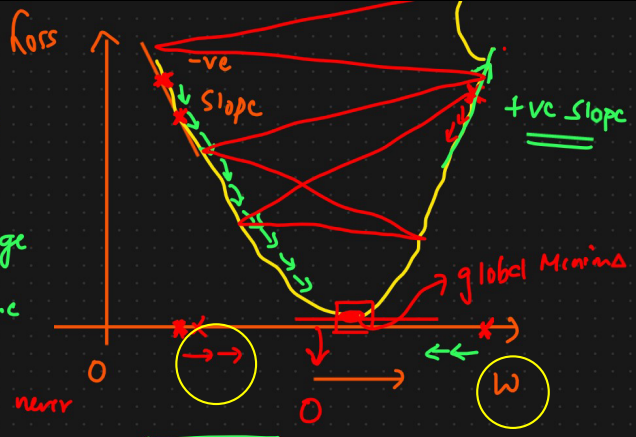
My aim is to increase weight to reach Global minima.

v.v.imp

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$ ⇒ Weight Updation Formula.

Gradient Descent ↗ Optimizers

Optimizers : To reduce the loss value



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

$= W_{old} + \eta\,(+ve)$

$W_{new} >> W_{old}$

η = large value ⇓ It may never converge

↳ learning Rate

0.001 → Small value.

↳ learning Rate

0.001 → standard.

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

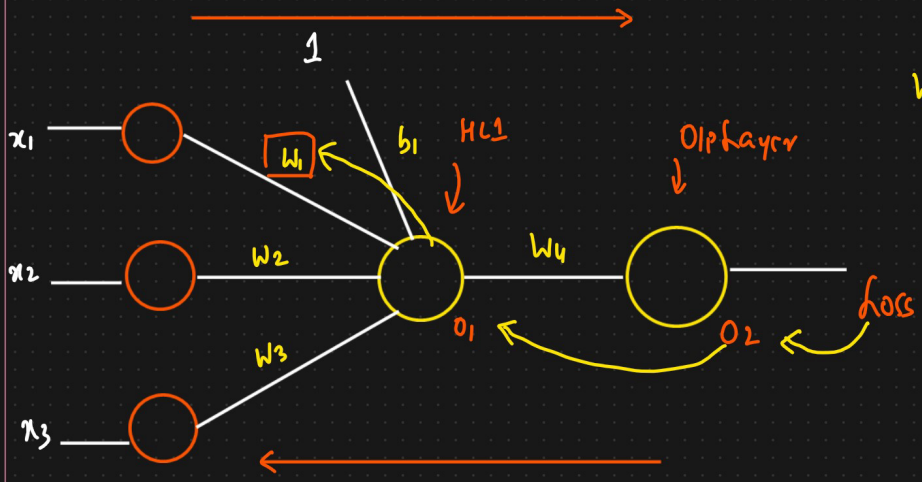
$= W_{old} - \eta\,(+ve)$

$W_{new} << W_{old}$

When W reaches global Minima

$W_{new} = W_{old}$

① Chain Rule of Derivative → used to find derivatives of composite functions (function inside a function).



$W_{new} = W_{old} - \eta \frac{\partial h}{\partial W_{old}}$

$W_{4_{new}} = W_{4_{old}} - \eta \frac{\partial h}{\partial W_{4_{old}}}$

$$\frac{\partial h}{\partial w_{4old}} = \frac{\partial h}{\partial o_2} * \frac{\partial o_2}{\partial w_{4old}} \Rightarrow \text{Chain Rule of Derivation}$$

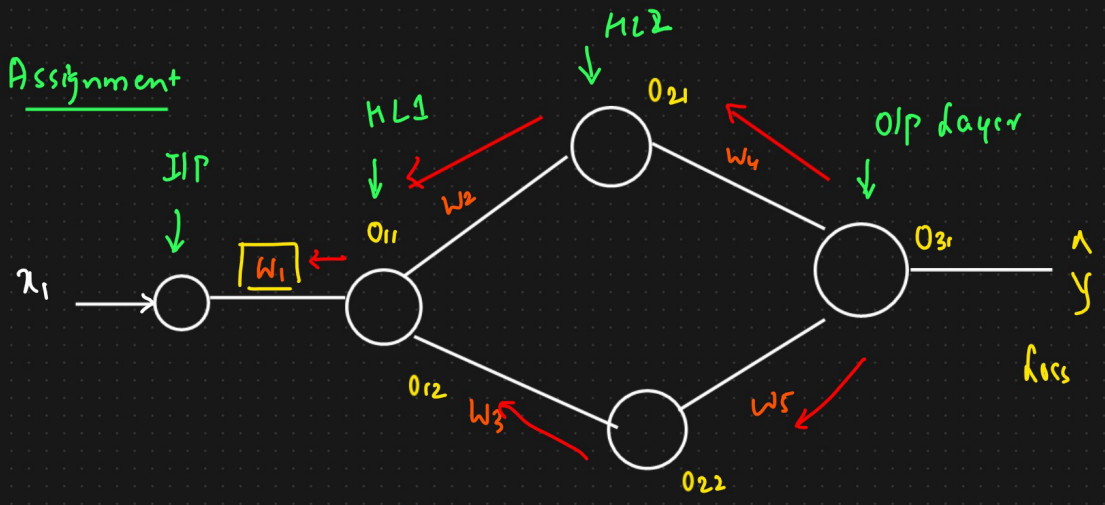
$$w_{1new} = w_{1old} - \eta \boxed{\frac{\partial h}{\partial w_{1old}}}$$

$$\boxed{\frac{\partial h}{\partial w_{1old}} = \frac{\partial h}{\partial o_2} * \frac{\partial o_2}{\partial o_1} * \frac{\partial o_1}{\partial w_{1old}}}$$

w_{2new}

w_{3new}.

Assignment



$$w_{1new} = w_{1old} - \eta \boxed{\frac{\partial h}{\partial w_{1old}}}$$

$$\Downarrow$$

$$\frac{\partial h}{\partial w_{1old}} = \left[\frac{\partial h}{\partial o_{31}} * \frac{\partial o_{31}}{\partial o_{21}} * \frac{\partial o_{21}}{\partial o_{11}} * \frac{\partial o_{11}}{\partial w_{1old}}\right]$$

$$+$$

$$\left[\frac{\partial h}{\partial o_{31}} * \frac{\partial o_{31}}{\partial o_{22}} * \frac{\partial o_{22}}{\partial o_{12}} * \frac{\partial o_{12}}{\partial w_{1old}}\right]$$

$\partial h \rightarrow \partial w_{old}$ (Chain Rule)

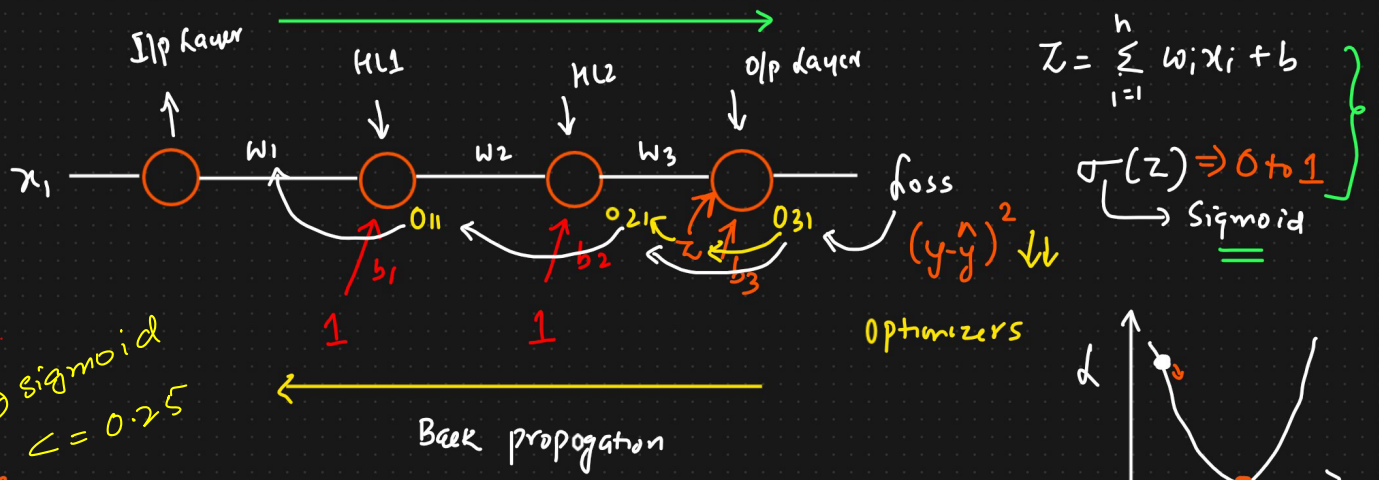
vvimp

Backpropagation = Chain Rule Repeated Backward

- Output Layer: Compute how loss changes with output (easy)
- Then apply **chain rule** to pass gradient back through each layer
- Eventually, you get the gradient of loss **with respect to every weight**

(Solution = Change Activation Function)

⑧ Vanishing Gradient Problem And Activation functions $\phi(z)$



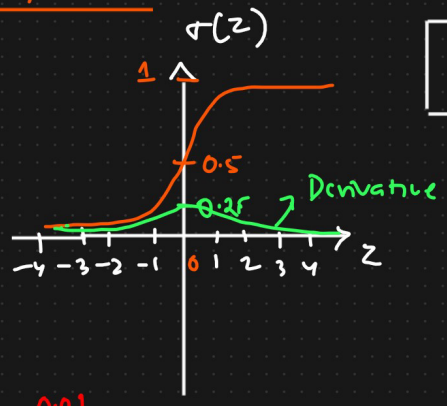
$$z = \sum_{i=1}^{n} w_i x_i + b$$

$\sigma(z) \Rightarrow 0$ to 1 → Sigmoid

Imp.
∂ sigmoid $< = 0.25$

⑨ Sigmoid Activation function

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



$$0 \le \sigma(z) \le 1$$

Vanishing Gradient Problem.

Derivative $(\sigma(z))$

$$0 \le \sigma(z) \le 0.25$$

0.01

$$W_{1\,new} = W_{1\,old} - \eta \left[\frac{\partial L}{\partial W_{1\,old}}\right] \Rightarrow \text{Small value}$$

$\Rightarrow$ Small Value

Equal

$$\Rightarrow W_{1\,new} \approx W_{1\,old}$$

$$\frac{\partial L}{\partial W_{1\,old}} = \frac{\partial L}{\partial O_{31}} * \left[\frac{\partial O_{31}}{\partial O_{21}}\right] * \frac{\partial O_{21}}{\partial O_{11}} * \frac{\partial O_{11}}{\partial w_{11}}$$

$0-0.25 * 0-0.25 * 0-0.25 \Rightarrow$ Smaller value $\Rightarrow 0.0000001$

$$O_{31} = \sigma(\underbrace{w_3 * O_{21} + b_3}_{z}) \qquad z = w_3 * O_{21} + b_3$$

$$O_{31} = \sigma(z)$$

$$\frac{\partial O_{31}}{\partial O_{21}} = \frac{\partial(\sigma(z))}{\partial(z)} * \frac{\partial z}{\partial O_{21}}$$

(Sigmoid; Chain Rule)

$0 \leq \sigma(z) \leq 0.25$ * $\frac{\partial((w_3 * o_{21}) + b_3)}{\partial(o_{21})}$

⇓

Derivative of Sigmoid

$$\frac{\partial o_{31}}{\partial o_{21}} = 0 \leq \sigma(z) \leq 0.25 * w_{3_{old}}$$

→ This causes loss of Gradient which means, we cant have updated values

* To fix this problem Researchers started exploring other Activation function

In this case, we will use tanh instead of Sigmoid function.

① Tanh ② Relu ③ Prelu ④ Swiss

Conclusion;

- Perceptron → No chain Rule
- Multi Layered Neural Network; we do use chain Rule of derivatives to update all the weights (composite fxs).

Steps;

1. Inputs
2. weights
3. $\sum w_i \times x_i$ + Bias
4. Hidden Layers
5. Activation functions
6. Loss functions $(y - \hat{y})$
7. Calculate derivatives for w/B
8. Update $w_{new} = w - \eta \, \partial h / \partial w_{old}$

   $B_{new} = B - \eta \, \partial h / \partial B_{old}$

} Chain Rule of Derivatives

Imp. Sigmoid value derivative ≤ 0.25 always

Derivative

ELU → more time / complex

Activation	Derivative Range	Max Derivative
ReLU	0 or 1	✅ 1
Leaky ReLU	Small positive to 1	✅ 1
Sigmoid	0 to 0.25	✅ 0.25
Tanh	0 to 1	✅ 1
Softmax	0 to 0.25 (per class)	✅ 0.25

Q. Difference b/w Sigmoid and Tanh

Tanh = 0 Centered, −1 to 1, hyperbolic

Sigmoid = not 0 centered, 0 to 1, Sigmoid

V.V.Imp Derivative value for different Gradients.

Activation	Derivative Range	Max Derivative
ReLU	0 or 1	✅ 1
Leaky ReLU	Small positive to 1	✅ 1
Sigmoid	0 to 0.25	✅ 0.25
Tanh	0 to 1	✅ 1
Softmax	0 to 0.25 (per class)	✅ 0.25

→ Gradient Vanishing problem since low numbers.

Remember → Gradient vanishing can still occur in Tanh

EVOLUTION OF ACTIVATION FUNCTIONS

1. Sigmoid / Softmax
2. Tanh
3. ReLU → One problem
   ↳ Dead Neuron when Negative

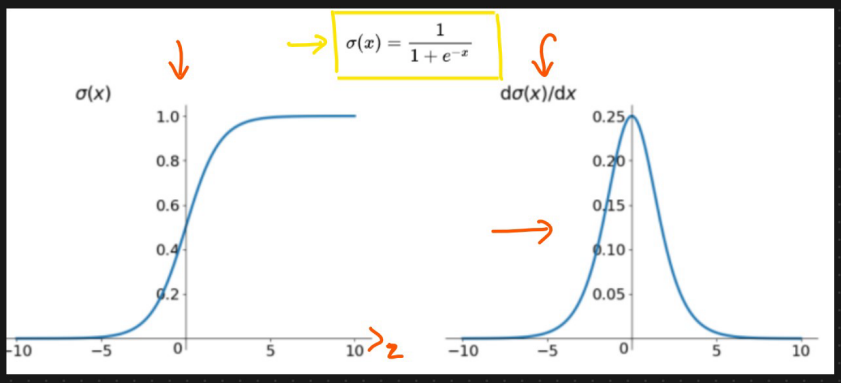
Best

4. Leaky Relu → solves problem of Dead Neurons, by leaking Negative values.

Activation Functions

① Sigmoid Activation function [0 to 1]

$$z = \sum_{i=1}^{n} w^T x + b$$

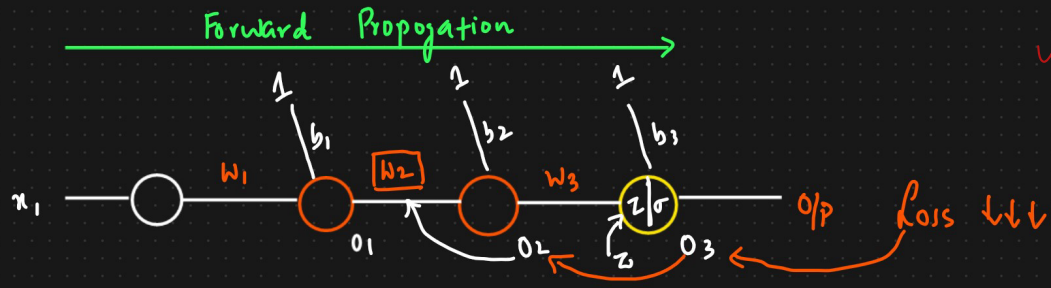


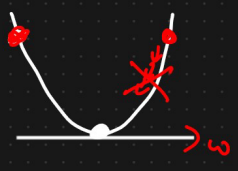
$\sigma(z) \Rightarrow$ 0 to 1

$\phi(z) \Rightarrow$

$\frac{\partial \sigma(z)}{\partial z}$ = 0 to 0.25

problem for vanishing Gradient.





$$W_{2new} = W_{2old} - \eta \left[\frac{\partial h}{\partial W_{2old}} \right]$$

0.001

0.00001

$\Rightarrow W_{2new} \approx W_{2old} \Rightarrow$

$$\frac{\partial h}{\partial W_{2old}} = \frac{\partial h}{\partial O_3} * \left[\frac{\partial O_3}{\partial O_2} \right] * \frac{\partial O_2}{\partial W_2}$$

0.20 * 0.01 ×

Let $z = (O_2 * W_3) + b_3$

$$\frac{\partial O_3}{\partial O_2} = \frac{\partial(\sigma(z))}{\partial z} * \frac{\partial z}{\partial O_2}$$

[0 to 1]

$$= [0 - 0.25] * \frac{\partial [(O_2 * W_3) + b_3]}{\partial O_2}$$

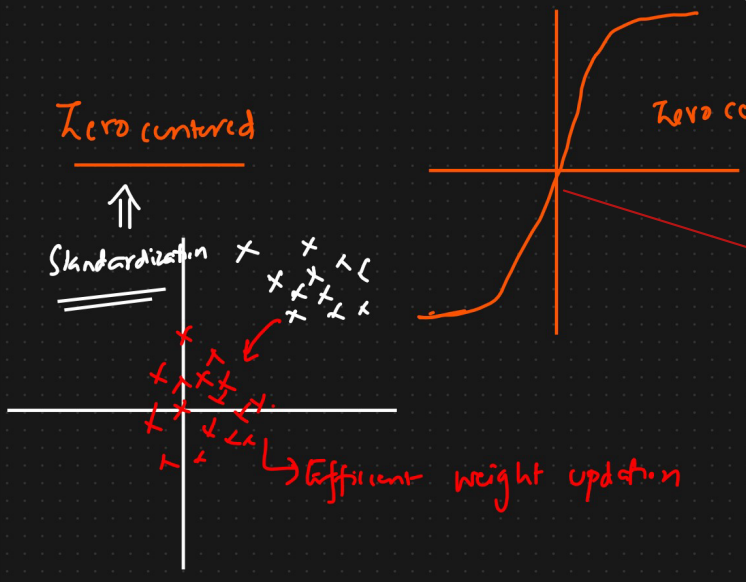
$$= [0 - 0.25] * W_3 \Rightarrow \text{Small value} \Rightarrow$$

Advantages

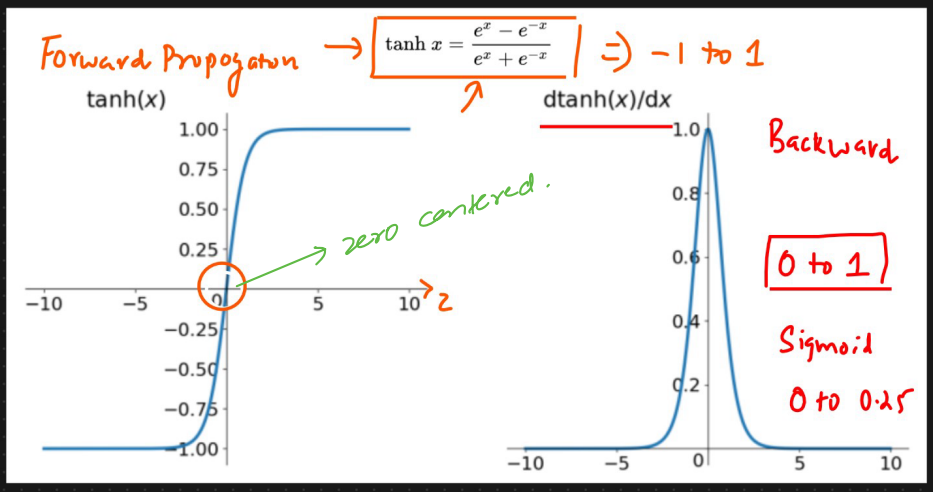
1. Binary classification Suitable.
2. Clear prediction i.e very close 1 or 0

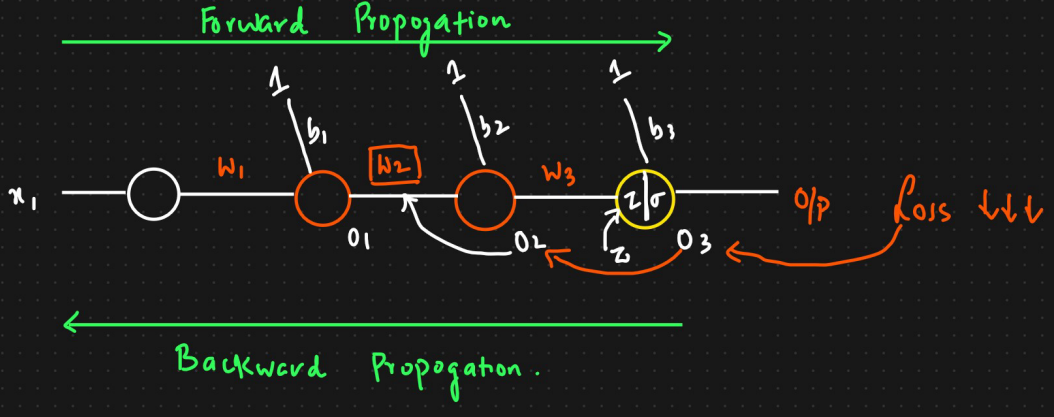
Disadvantages

1. Prone to vanishing gradient Problem.
2. Function output is not Zero centred ⇒ Efficient weight updation
3. Mathematical operation are relatively time Consuming



② Tanh Activation Function





$$\frac{\partial h}{\partial w_{2old}} = \frac{\partial h}{\partial o_3} * \frac{\partial o_3}{\partial o_2} * \frac{\partial o_2}{\partial w_2}$$

$0.20_{11} * 0.01 \times$

Let $z = (o_2 * w_3) + b_3$

$$\frac{\partial o_3}{\partial o_2} = \frac{\partial(\tanh(z))}{\partial z} * \frac{\partial z}{\partial o_2}$$

[0 to 1]

$$= [0-1] * \frac{\partial[(o_2 * w_3) + b_3]}{\partial o_2}$$

$$= [0-1] * w_3 \Rightarrow \text{Small value} \Rightarrow$$

Advantages

① Zero Centric ⇒ Weight updation is Efficient

Disadvantages

① Prone to Vanishing Gradient Problem

② Time complexity

③ Relu Activation Function

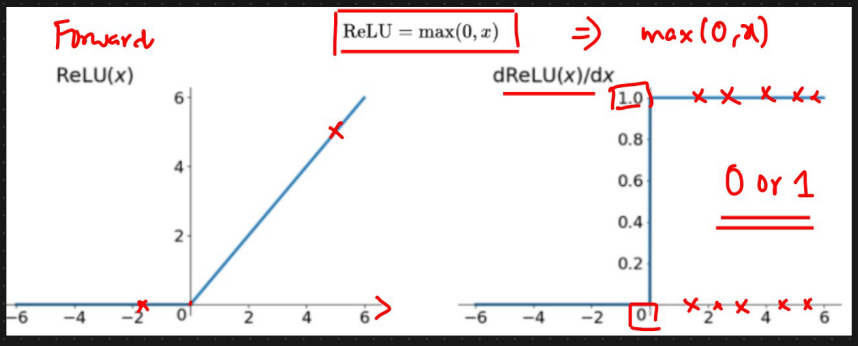
[Rectified Linear Unit].

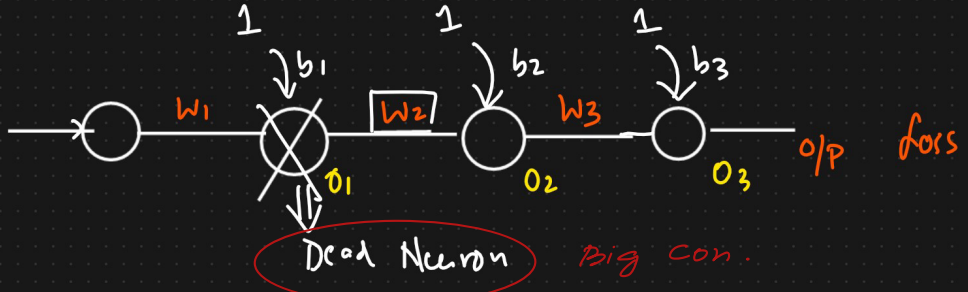
Tanh ⇒ 0 to 1

Sigmoid ⇒ 0 to 0.25

Forward

$\text{ReLU} = \max(0, x)$ ⇒ $\max(0, x)$





$$\frac{\partial h}{\partial w_{2old}} = \frac{\partial h}{\partial o_3} * \boxed{\frac{\partial o_3}{\partial o_2}} * \frac{\partial o_2}{\partial w_2}$$

0.20_{11} * 0.01 ×

Let $z = (o_2 * w_3) + b_3$

$$\frac{\partial o_3}{\partial o_2} = \boxed{\frac{\partial (\text{Relu}(z))}{\partial z}} * \frac{\partial z}{\partial o_2} \qquad [0 \text{ to } 1]$$

$$= [0 \text{ or } 1] * \frac{\partial [(o_2 * w_3) + b_3]}{\partial o_2}$$

$$= [\overset{-ve}{0} \text{ or } \overset{+ve}{1}] * w_3 \Rightarrow \text{Small value} \Rightarrow$$

Derivative of

If Relu output is $\boxed{1}$ $\Rightarrow$ Weight updation will happen

If Relu output is $\boxed{0}$ $\Rightarrow$ Dead Neuron

$$w_{2new} = w_{2old} - \eta \boxed{\frac{\partial h}{\partial w_{2old}}} \Rightarrow 0$$

If Derivative of Relu(z) is 0

$$\boxed{w_{2new} \approx w_{2old}} \Rightarrow \text{Dead Neuron}$$

If $z = +ve$ $\qquad \frac{\partial \text{Relu}(z)}{\partial z} = 1$

If $z = -ve$ $\qquad \frac{\partial \text{Relu}(z)}{\partial z} = 0$

Advantages

① Solving Vanishing Gradient Problem

② Max(0, x) → Calculation is superfast. The Relu function has a linear Relationship

③ It is much faster than Sigmoid or Tanh.

Disadvantages

① Dead Neuron

② Relu function O/P

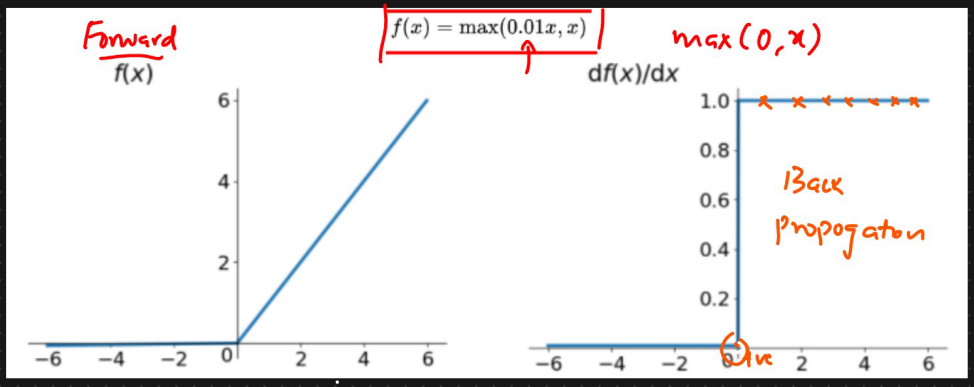
(0, x) ⇒ 0 or +ve number

⇓

It is not zero centric

④ Leaky Relu And Parametric Relu

max(αx, x) → hyperparameter α = alpha = 0.01, 0.02, 0.03



Relu → Dead Neuron → Dead Relu Problem

Advantages

① Leaky Relu has all the advantage of Relu

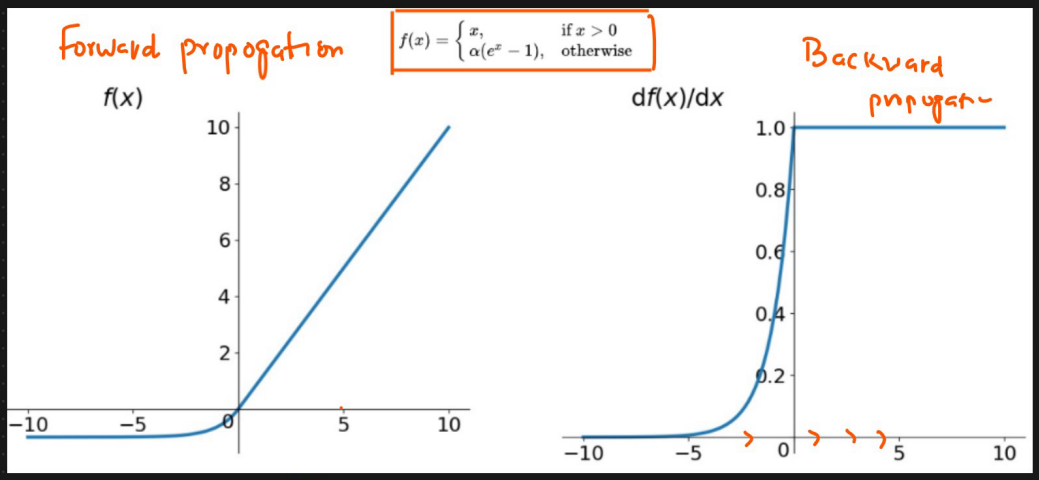
② It removes the Dead Relu Problem

Disadvantage

① It is not zero centric

⑤ ELU (Exponential Linear Units)

$\alpha(e^x - 1)$.



[It is used to solve Relu problems]

Advantages

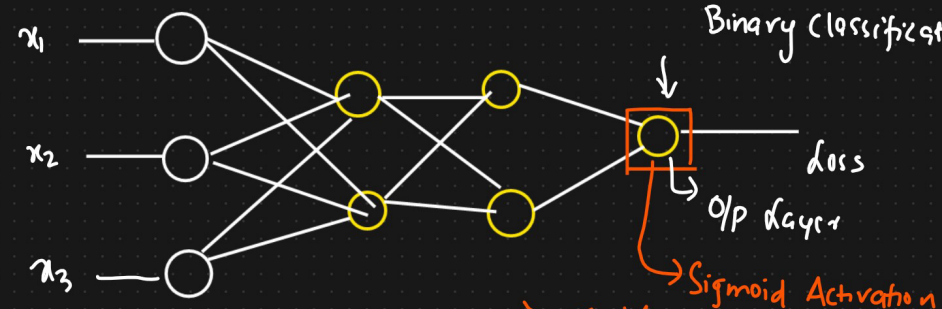
① No Dead Relu Issues

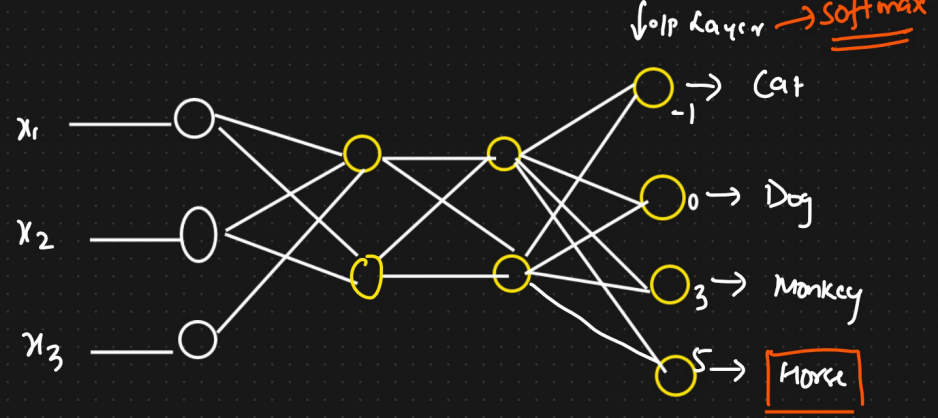
② Zero centered

Disadvantage

1) Slightly more computationally Intensive.

⑥ Softmax Activation function [Multiclass classification Problem]





Softmax Activation

$$\text{Softmax} = \frac{e^{y_i}}{\sum_{K=0}^{n} e^{y_K}}$$

$y_i = 0 * w + b$

Softmax ⇒ Cat $= \frac{e^{-1}}{e^{-1+0+3+5}} = 0.00033$

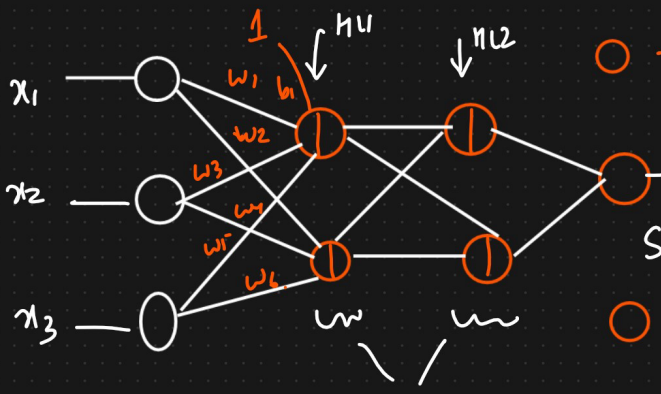
Dog $= \frac{e^{0}}{e^{-1+0+3+5}} = 0.0024$

Monkey $= \frac{e^{3}}{e^{-1+0+3+5}} = 0.0183$

Horse $= \frac{e^{5}}{e^{-1+0+3+5}} = 0.1353$

$Pr(\text{Horse}) = \frac{0.1353}{0.00033 + 0.0024 + 0.0183 + 0.1353} \approx 86\%$

⑦ Which Activation Function To Use When ?



↓ O/P layer

→ Softmax ⇒ Multiclass Classification

Sigmoid → Binary classification

Hidden layers → Sigmoid or Tanh ⇓ Vanishing Gradient Problem

Relu or its variant

Prelu, Leaky Relu,

Elu

In short :

Hidden layer — ReLU

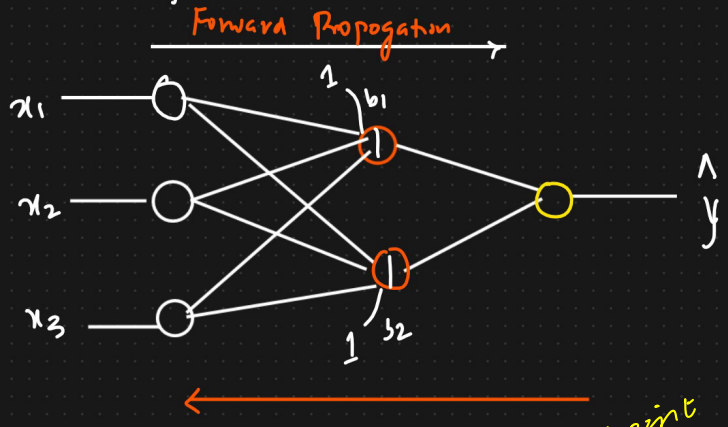
Output layer (classification) — Sigmoid Softmax

Output layer (Regression) — Regression Activator

✅ Activation Function Decision Chart (with Linear)

Situation / Layer Type	Best Activation Function
✅ Hidden Layers (default)	**ReLU**
Hidden Layers (risk of dead neurons)	**Leaky ReLU**
Hidden Layers (RNNs / time series)	**tanh**
Output – Binary Classification	**Sigmoid**
Output – Multi-class Classification	**Softmax**
Output – Regression (real number prediction)	**Linear**
Advanced models (deep / experimental)	**Swish / ELU**

Loss function And Cost Function



$$\text{Cost-fn} = \frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2 \quad \text{(Error)}$$

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

Optimizers

Gradient Descent

x_1	x_2	x_3	O/P ($\hat{y}$)
-	-	-	0
-	-	-	1
-	-	-	0
-	-	-	1

Loss function → Single Datapoint Error

Costfunction → Entire Data Error. → Loss function for Batches of Data Points

$$MSE = (y-\hat{y})^2$$

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

ANN → Cost functions

- Classification
- Regression
  1. Mean Squared Error (MSE)
  2. Mean Absolute Error (MAE)
  3. Huber Loss
  4. RMSE

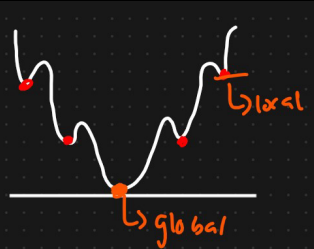
1. Mean Squared Error (MSE)

$$\text{loss function} = (y-\hat{y})^2$$

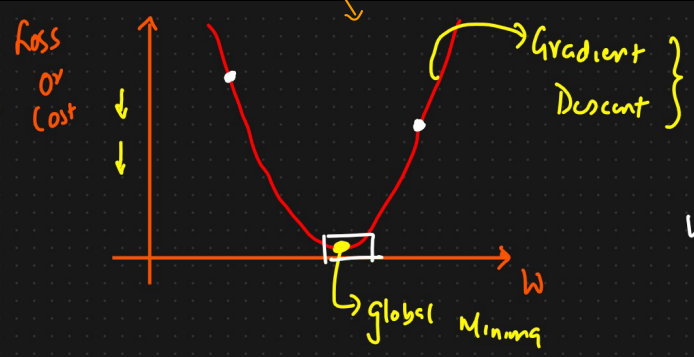
$$\text{Cost fn} = \frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2$$

↳ Quadratic Equations

If we plot loss function we get Gradient Descent



Weight updation



$$W_{new} = W_{old} - \eta \left[\frac{\partial h}{\partial W_{old}} \right]$$

Slope

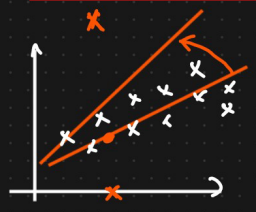
Advantages

1. MSE is Differentiable
2. It has 1 local or global Minima
3. It converges faster

Disadvantages

Imp

1. Not Robust to outliers



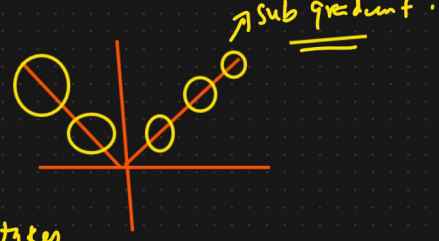
penalizing the Error

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

2. Mean Absolute Error (MAE)

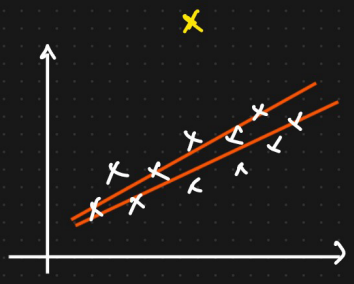
Loss fn : $|y - \hat{y}|$

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



Advantages

1. Robust to outliers



Disadvantages

1. Convergence usually takes time in MAE

3. Huber loss

Combination of MSE and MAE where we use MSE for No outliers and vice versa.

1. MSE
2. MAE

MSE ⇑ No outlier ⇑ Hyperparameter ⇑

Cost fn = $\begin{cases} \frac{1}{n} \sum_{i=1}^{n} (y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \boxed{\delta} \end{cases}$

$$\left[\delta|y-\hat{y}| - \frac{1}{2}\delta^2, \text{otherwise}\right.$$

⇓

MAE

④ RMSE (Root Mean Squared Error)

=

$$\text{Cost function} = \sqrt{\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}{N}}$$

Advantages

Disadvantages

② Loss or Cost function For Classification Problems

Important

Classification ⟶ Cross Entropy (Cost)

- Binary Cross Entropy (Binary class) Loss
- Categorical Cross Entropy (Multiclass) Loss

① Binary Cross Entropy

Log Loss

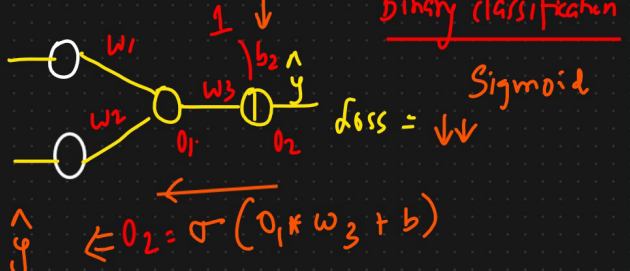
⇓

$$\text{loss} = -y * \log(\hat{y}) - (1-y) * \log(1-\hat{y})$$

y = Actual value

$\hat{y}$ = Predicted value

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



$$\hat{y} = \frac{1}{1+e^{-z}}$$ ⇒ Sigmoid Activation function

② Categorical CROSS Entropy (Multiclass classification) OHE is used

OHE → One Hot Encoding

	f_1	f_2	f_3	O/P	Good (j=1)	Bad (j=2)	Neutral (j=3)
→	2	3	4	Good	1	0	0
→	5	6	7	Bad	0	1	0
→	8	9	10	Neutral	0	0	1

C = No of categories

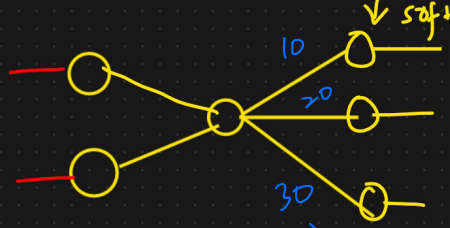
i = 1 to n

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

Actual ⇐ $y_{ij} = \begin{bmatrix} y_{11} & y_{12} & y_{13} & \cdots & y_{1C} \\ y_{21} & y_{22} & y_{23} & \cdots & y_{2C} \\ \vdots & & & & \end{bmatrix}$

$$y_{ij} = \begin{cases} 1 & \text{if the element is in the class} \\ 0 & \text{Otherwise} \end{cases}$$

Prediction ⇐ $\hat{y}_{ij}$ ⇒ Softmax Activation $= Sof(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$



O/p $\hat{y}_{ij}$ = Probabilities

Categorical Cross Entropy $\Rightarrow [0.2, 0.3, 0.5]$

[This also gives the probability of other categories]

$[0.1, 0.2, 0.3, 0.2, 0.2] \leftarrow 1$

$= \frac{e^{z_i}}{e^{z_j}}$

$= \frac{e^{10}}{e^{10+20+30}}$

③ Sparse Categorical Cross Entropy

0 1st 2nd → Categories

$[0.2, 0.3, 0.5]$

$\Downarrow$

2nd Index $\Rightarrow$ O/p

Disadvantage

① Loosing info about the probability of other category.

0 1 2nd 3rd 4th

$[0.2, 0.3, 0.1, 0.2, 0.2] \Rightarrow 1^{th}$ Index

$\Downarrow$

Category $\Rightarrow$ O/p

④ Right Combination

Activation applied

$\Downarrow$

	Hidden Layers	O/p Layer	Problem Statement	Loss function
①	Relu or its Variants	Sigmoid	Binary classification	Binary Cross Entropy
②	Relu or its Variants	Softmax	Multi class	Categorical or Sparse CE
③	Relu or its Variants	linear	Regression	MSE, MAE, Huber loss, RMSE

Conclusion

✅ Types of Neural Networks & Their Use Cases

Neural Network Type	Best Suited For	Data Type
ANN (Artificial Neural Network)	Regression & classification on **tabular data**	Numeric/tabular data
CNN (Convolutional Neural Network)	Image recognition, video analysis, object detection	Images, video, spatial data
RNN (Recurrent Neural Network)	Sequence modeling, text, time-series forecasting	Text, speech, time-series

Loss function → Error for single data point

Cost function → Error for entire data (epoch or Batch)

ANN → Regression, classification

Regression

Loss → MSE → Outlier Sensitive
MAE → outlier insensitive
Huber loss → Comb MSE / MAE
RMSE

VVimp Why do we need Regression Activator fn. in LR? Unlike softmax, it is used to output Raw unbounded values, not non linearity.

🚫 "No Activation" = Linear Activation

That means:

> The output layer just returns the **raw value** from the final neuron, with **no transformation** applied.

✅ Summary:

Term	Meaning
No Activation	Return raw output (linear function)
Used in	Regression (real-valued outputs)
Why	Keeps values unbounded (−∞ to +∞)

Classification

📊 Classification Cheat Sheet

Classification Type	Loss Function	Activation Function
Binary Classification	**Binary Cross Entropy (Log Loss)** 📌 $L = -[y \cdot \log(p) + (1-y) \cdot \log(1-p)]$	**Sigmoid** 📌 $\sigma(x) = 1 / (1 + e^{(-x)})$
Multi-Class Classification	**Categorical Cross Entropy** 📌 $L = -\sum y_i \cdot \log(p_i)$ (for i = 1 to C)	**Softmax** 📌 $p_i = e^{(z_i)} / \sum e^{(z_j)}$ (for j = 1 to C)

Important

✅ Activation Functions by Problem Type

Problem Type	Hidden Layer Activation	Output Layer Activation
Binary Classification	ReLU	Sigmoid
Multi-Class Classification	ReLU	Softmax
Regression (Single Output)	ReLU	Linear

$IM Question ?

> "Why should I use **Artificial Neural Networks (ANNs)** for regression or classification when I can just use **Linear Regression** or **Logistic Regression**?"

✅ Short Answer:

You **don't always need ANN**. But you might **want** to use ANN when **traditional ML models aren't enough**.

🔍 Deeper Comparison:

Aspect	Linear/Logistic Regression	ANN (Deep Learning)
Complexity of patterns	Handles **linear** relationships only	Captures **non-linear** relationships
Feature engineering	Often requires **manual** feature crafting	Learns features **automatically**
Scalability	Fast and interpretable	Scales better with **large data**
Overfitting risk	Low on small data	High unless regularized/trained well
Interpretability	Very high (coefficients are clear)	Low (black box)
Speed	Very fast	Slower, needs more compute

📈 Example Scenario:

❌ **Logistic Regression Fails:**

You have a classification problem where the classes are **not linearly separable** (e.g., spiral or concentric circles).

👉 Logistic regression draws a straight line. It fails.

✅ **ANN Wins:**

A neural network can learn **non-linear decision boundaries** and separate the classes better.

TL;DR:

> Use **Logistic/Linear Regression** when:
>
> - "Data is small"
> - "Relationships are linear"
> - "You want speed & interpretability"

> Use **ANN** when:
>
> - "Data is large or complex"
> - "You suspect **non-linear interactions**"
> - "You're hitting performance limits with traditional models"

💡 **Pro tip**: Always **start simple** (like Logistic Regression).
Only add complexity (ANN) when it's needed.

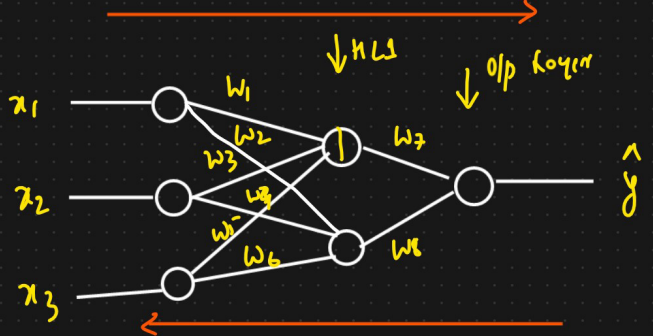
Optimizers ⟶ used for reducing cost function or reach Global minima.

1. Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini batch SGD
4. SGD With Momentum
5. Adagrad and RMSPROP
6. Adam Optimizers

Epoch → completed 1 iteration i.e.: 1 Forward P. 1 Backward P.

Best of both worlds but bit noisy

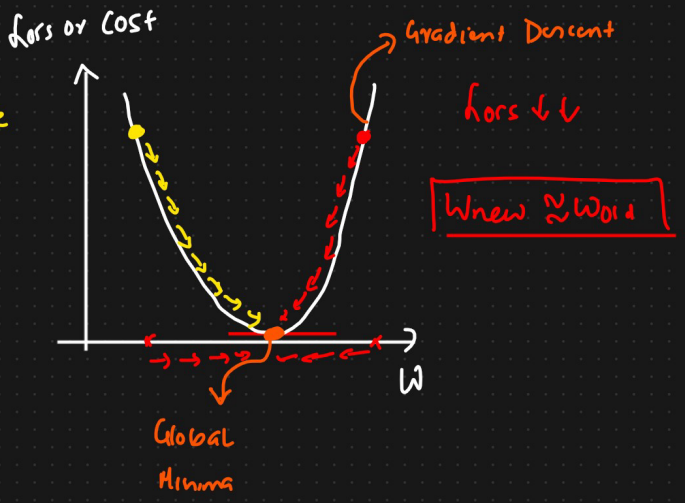
① Gradient Descent Optimizer



Backward Propogation

Loss = [ ] ↓↓↓

Optimizers ↑



Learning Rate

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

MSE

$$\text{Loss fn} = (y - \hat{y})^2 \qquad \text{Cost fn} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Dataset = 1000 Data points

Epochs, Iteration

1 Epoch = 1 Iteration

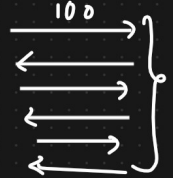
10 Iteration

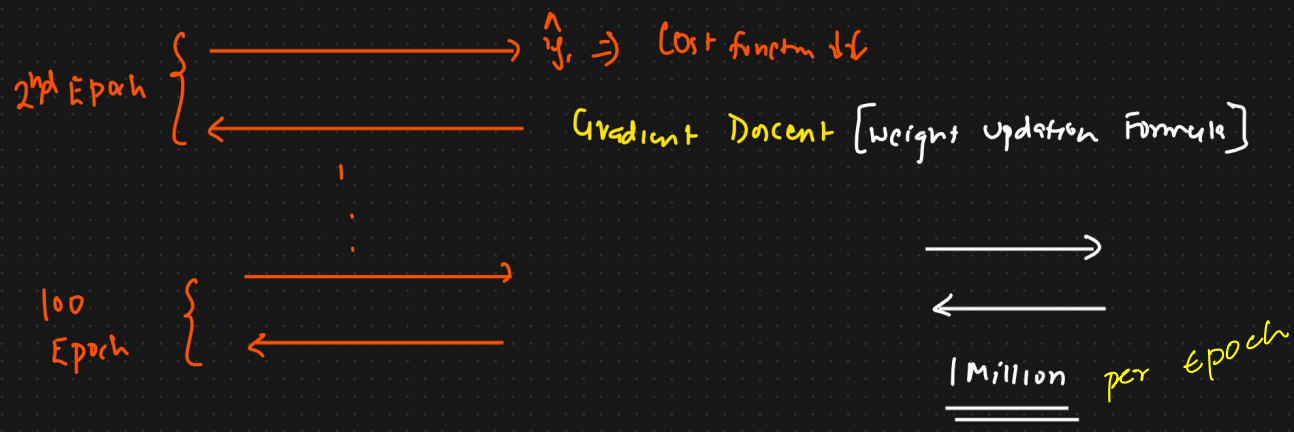
100/10 = 100

1 Epoch { 1000 datapoint ⟶ $\hat{y}_i$ ⇒ Cost function ↓↓

Weights will get updated ⟵ }

Gradient Descent optimizer [weight updation]





Advantages

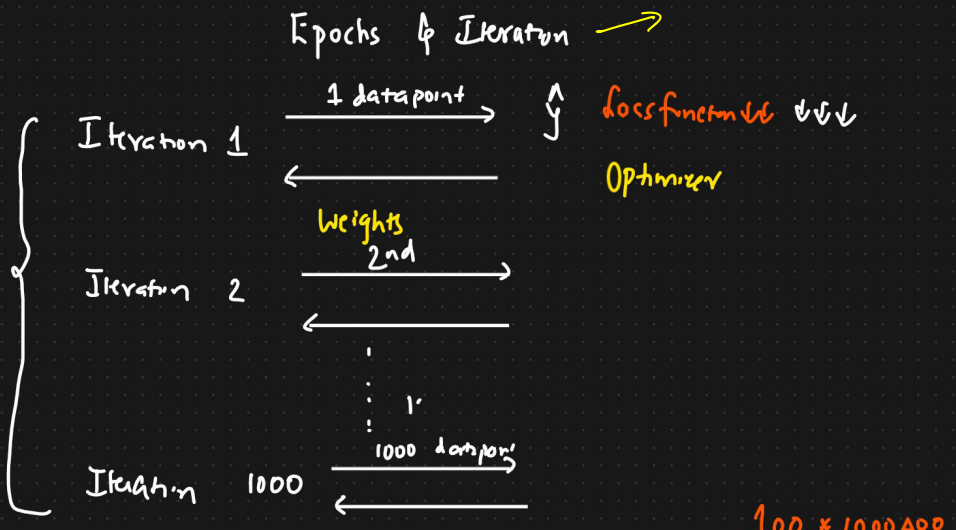
① Convergance will happen.
- Accurate Gradients
- Good for small Datasets

Disadvantage

① Huge Resource RAM, GPU
⇓
Resource Intensive.

② Stochastic Gradient Descent (SGD)

1000 datapoint



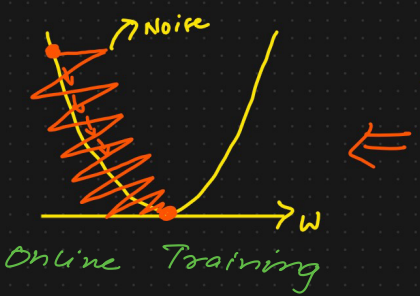
100 × 1000 = 100000

Epochs

1000000 datapoint

Advantage

Ⓐ Solve Resource Issue



Disadvantage

Ⓑ Time Complexity ↑↑

Ⓒ Convergence will also take more time.

Ⓓ Noise gets Introduced

→ may skip Global minima.

③ Mini Batch SGD

Batch sizes

No. of iterations = $\frac{100000}{1000}$ = 100 iterations

Epoch, Iteration, Batch_size

Datapoints = 100000

batch_size = 1000

MSE

Cost fn = $\sum_{i=1}^{1000} (y_i - \hat{y}_i)^2$ ↓↓

Epoch 1

Iteration 1 → 1000 →

← Optimizer ⇒ Mini Batch SGD

Change the weights

Iteration 2 → 1000 →

←

Iteration 3 →

←

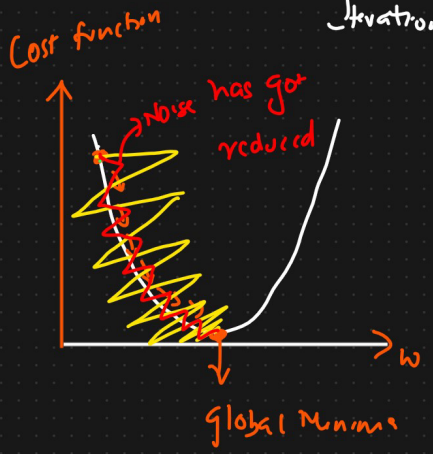
⋮

Iteration 100 →

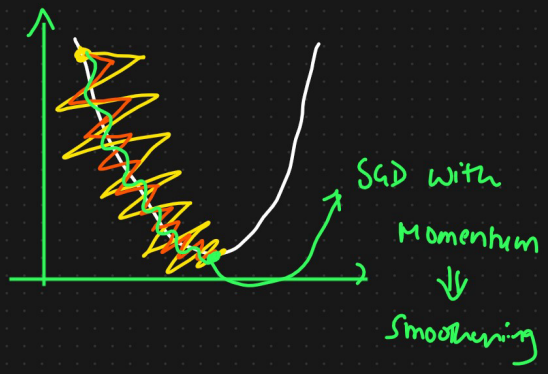
←

↓↓

8gb	1
16gb	5000



Batch Size



Advantages

① Convergence speed will increase =

② Noise will be less when compared to SGD

③ Efficient Resource Usage (RAM)

Disadvantage

① Noise still exists

In every epoch 1000 iterations of 1000 DP

Conclusion

Optimizers = Reduce cost function

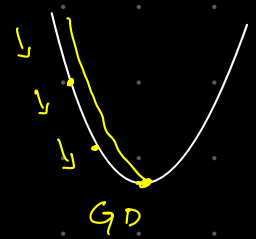
Working

🔪 1. Batch Gradient Descent (GD)

You taste **all 100 cakes at once**, take notes, average out all your feedback, and then **make one big adjustment** to the recipe.

- 🎯 One big decision after thorough review
- ⏳ Takes time (you must finish all cakes)
- 📉 Very precise, but slow

📌 **1 update per round (epoch)**
📌 Uses all 100 cakes each time



☕ 2. Stochastic Gradient Descent (SGD)

You're in a hurry — you **taste one cake**, make a quick adjustment. Then you taste the next cake, make another tweak.

- 🎲 Fast but very chaotic
- 👎 Some tweaks may help, others might hurt
- 🎯 But over time, you may stumble upon the perfect recipe

📌 **100 updates per round (epoch)**
📌 Uses 1 cake at a time, very noisy

🍽️ 3. Mini-Batch Gradient Descent (batch size = 10)

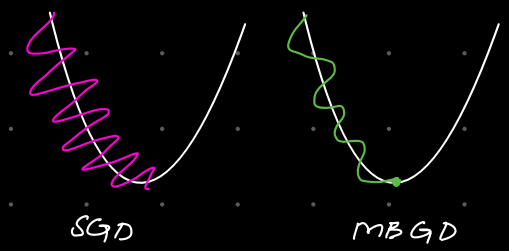
You group the 100 cakes into **10 plates** (batches) of 10 cakes each. You taste a plate, make a small adjustment, and repeat.

- ⏱️ Balanced speed
- ✅ Not too noisy, not too slow
- 🎯 Gets better with practice

📌 **10 updates per round (epoch)**
📌 Each update uses 10 cakes

🔁 Summary (Cake Edition):

Method	How You Taste	How You Adjust	Feeling
GD	Taste all 100	One big tweak	Slow but sure
SGD	Taste 1 cake	Constant tweaking	Fast but shaky
Mini-Batch	Taste 10 at a time	Small steady tweaks	Best of both worlds



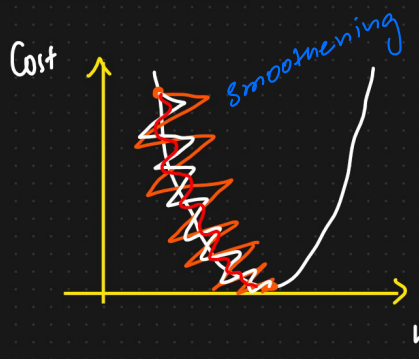
Explanation:

✅ Gradient Descent Variants Explained

Type	How it Works	Speed	Stability	Used When	Pros	Cons
Batch Gradient Descent	Uses the **entire dataset** to compute gradient before updating weights.	❌ Slow (per update)	✅ Very stable	Small datasets	+ Accurate gradients + Converges smoothly	– Very slow on large data – High memory usage
Stochastic GD (SGD)	Uses **1 data point** at a time to update weights.	✅ Fast (per update)	❌ Very noisy	Large datasets, online learning	+ Fast updates + Good for huge/streaming data + Can escape local minima	– Noisy convergence – May overshoot or bounce around
Mini-Batch SGD	Uses a **small subset** of data (e.g., 32, 64) for each update.	⚖️ Balanced	✅ Smooth + efficient	Most common in deep learning	+ Fast and stable + Efficient on GPUs + Generalization-friendly	– Needs tuning of batch size – Still a bit noisy

we need Momentum based SGD to smoothen noise (exponential weighted average)

④ SGD With Momentum



$$W_{new} = W_{old} - \eta \frac{\partial h}{\partial W_{old}}$$ (learning rate)

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

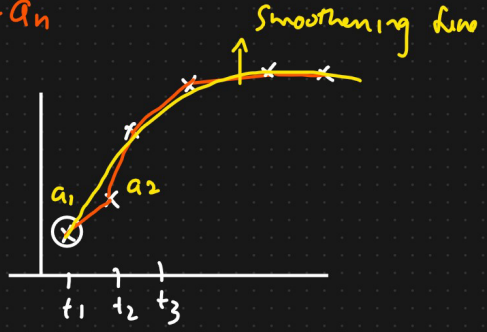
$$W_t = W_{t-1} - \eta \frac{\partial h}{\partial W_{t-1}}$$

Weight Updation formula

Exponential Weight Average {Smoothening} ⇒ ARIMA, SARIMAX

Time =	t_1	t_2	t_3	t_4	- - -	t_n
Values =	a_1	a_2	a_3	a_4	- - - -	a_n

Time Series



$$V_{t_1} = a_1$$

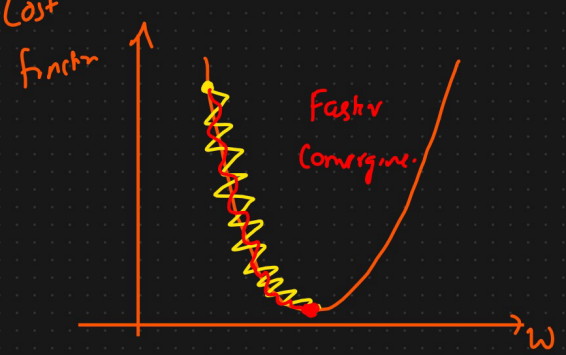
$$V_{t_2} = \boxed{\beta} * V_{t_1} + (1-\beta) * a_2$$

$\beta = 0$ to 1

$$= 0.95 * a_1 + (0.05)\, a_2$$

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

$$= 0.95 \left[0.95 * a_1 + (0.05)\, a_2 \right] + (0.05) * a_3$$



Advantage

① Reduces the noise

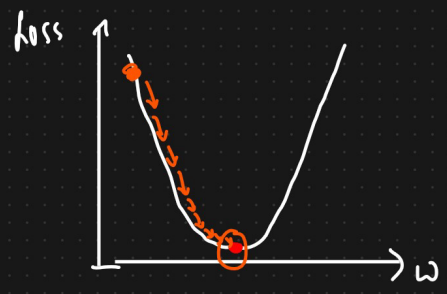
② Quick Convergence

⑤ Adagrad : Adaptive Gradient Descent

η = fixed ⇒ Dynamic learning Rate

$$w_t = w_{t-1} - \eta \frac{\partial h}{\partial w_{t-1}}$$

→ Learning Rate = 0.001



As the convergence happen the learning rate should change

$0.00001 \approx 0$

$$w_t = w_{t-1} - \eta' \frac{\partial h}{\partial w_{t-1}}$$

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

$\eta' \downarrow\downarrow$ $\alpha_t \uparrow\uparrow$

Epsilon ≈ small value

$\alpha_t \uparrow\uparrow$

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

$w_t \approx w_{t-1}$

$t=1 \rightarrow t=2 \rightarrow t=3$

$\eta = 0.01$ $\eta = 0.005$ $\eta = 0.003$

Disadvantage

① η' → Possibility to become a very small value ≈ 0

⑥ Adadelta And RMSPROP

Exponential Weight Average

$\beta = 0.95$ $\quad S_{dw_t} = 0$

$$S_{dw_t} = \beta * S_{dw_{t-1}} + (1-\beta)\left(\frac{\partial h}{\partial w_{t-1}}\right)^2$$

$$\eta' = \frac{\eta}{\sqrt{S_{dw_t} + \epsilon}}$$

[Dynamic LR + Smoothening EWA]

$$w_t = w_{t-1} - \eta' \frac{\partial h}{\partial w_{t-1}}$$

⑦ Adam Optimizer

SGD with Momentum + RMSPROP [Dynamic LR + Smoothening]

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

Weight Updation

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

Bias Updation

$$\eta' = \frac{\eta}{\sqrt{S_{dw_t} + \epsilon}}$$

$S_{dw_t} = 0$

EWA

$$S_{dw_t} = \beta * S_{dw_{t-1}} + (1-\beta)\left(\frac{\partial h}{\partial w_{t-1}}\right)^2$$

[EWA]

$$V_{dw_t} = \beta * V_{dw_{t-1}} + (1-\beta)\frac{\partial h}{\partial w_{t-1}}$$

$$V_{db_t} = \beta * V_{db_{t-1}} + (1-\beta)\frac{\partial h}{\partial b_{t-1}}$$

⇒ Momentum
↳ Smoothening

Final Conclusion

✅ Gradient Descent Variants & Optimizers — Full Comparison

Type	How it Works	Speed	Stability	Used When	Pros	Cons
Batch Gradient Descent	Uses the **entire dataset** to compute gradient before each weight update.	❌ Slow (per update)	✅ Very stable	Small datasets	+ Accurate gradients + Smooth convergence	– Very slow on large data – High memory usage
Stochastic GD (SGD)	Uses **1 data point** at a time to compute and apply updates.	✅ Fast (per update)	❌ Very noisy	Large datasets, online learning	+ Very fast updates + Good for real-time/streaming + Can escape local minima	– Noisy updates – May bounce or overshoot
Mini-Batch SGD	Uses a **small batch** (e.g., 32–128 samples) per update.	⚖️ Balanced	✅ Smooth + efficient	Deep learning, GPU training	+ Combines speed & stability + Efficient on hardware + Improves generalization	– Needs batch size tuning – Some noise still persists
SGD with Momentum	Like Mini-Batch SGD, but adds **momentum**: a fraction of past updates to current updates — helps **smooth curves** and accelerate convergence.	✅ Faster than SGD	✅ More stable	Deep learning, non-convex optimization	+ Speeds up convergence + Reduces oscillation + Helps escape flat regions (plateaus)	– Needs momentum tuning – Risk of overshooting
AdaGrad	Adapts learning rate **per parameter** based on past squared gradients — good for sparse data.	⚡ Fast initially	⚖️ Decent	Sparse features, NLP tasks	+ No LR tuning needed + Great for sparse data like text (NLP)	– Learning rate decays too fast – Stops learning early
Adam Optimizer	Combines **momentum + adaptive learning rate** per parameter using moving averages of gradients.	✅ Fast & efficient	✅ Very stable	Deep learning (default choice)	+ Fast convergence + Works well out-of-the-box + Handles sparse & noisy gradients	– May overfit – Sensitive to hyperparameters – May not generalize best

🧠 TL;DR:

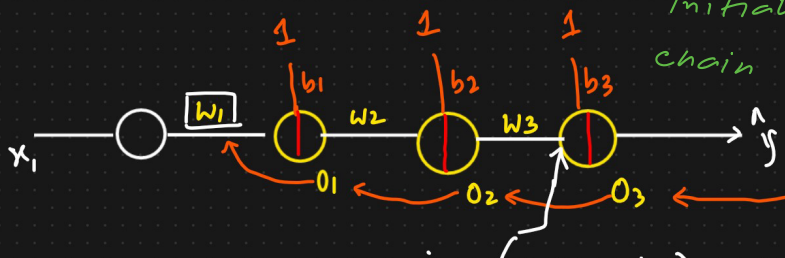
Situation	Best Choice
Small, simple dataset	**Batch GD**
Real-time or massive streaming data	**SGD**
General deep learning tasks	**Mini-Batch SGD / Adam**
Complex loss surfaces, faster training	**SGD with Momentum**
NLP or sparse data	**AdaGrad / Adam**

Online Train

Less Noise

Imp. Adam optimizer = Best optimizer (Moving avg of gradients).
Combines Adaptive Learning Rate + Momentum

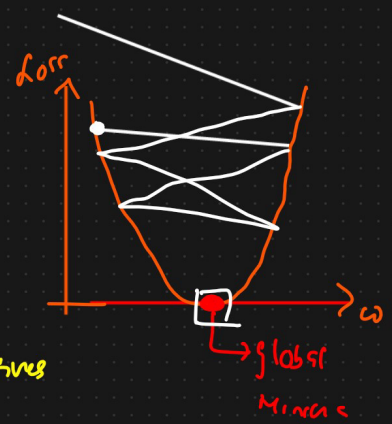
Exploding Gradient Problem ⇒ Weight Initialization Technique



Initializing with high value means chain derivative will give high Wnew

$\rightarrow z = (O_2 * w_3 + b_3) \rightarrow \sigma(z)$

Optimizers



$$W_{1new} = W_{1old} - \left[\eta \frac{\partial h}{\partial W_{1old}}\right]$$

$\{ W_{new} >>>> W_{1old} \quad W_{new} << W_{1old} \}$

Chain Rule of Derivatives

$$\frac{\partial h}{\partial W_{1old}} = \frac{\partial h}{\partial O_3} * \frac{\partial O_3}{\partial O_2} * \frac{\partial O_2}{\partial O_1} * \frac{\partial O_1}{\partial W_{1old}}$$

big * big * big * big ⇒ big value

Weight initialization ⇓ Very high value

$$\frac{\partial O_3}{\partial O_2} = \frac{\partial \sigma(z)}{\partial z} * \frac{\partial z}{\partial O_2}$$

$$= [0 - 0.25] * \frac{\partial (O_2 * w_3 + b_3)}{\partial O_2} = [0 - 0.25] * w_3 \Rightarrow 500 - 1000$$

✅ **Solutions:**

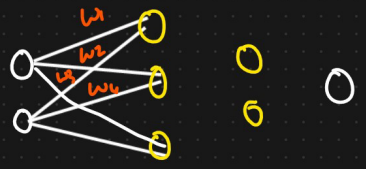
Problem	Solution
Gradients getting too large	Gradient Clipping
Poor initialization	Use He/Xavier Initialization
Very deep network	Use architectures like ResNet (skip connections)
High learning rate	Lower learning rate

Exploding gradients happen when gradients grow too large during backpropagation, often due to **deep networks or bad initialization.** They can break the training process. Fix it with good initialization, gradient clipping, and careful architecture design.

Gradient explode =
- Bad initialization
- Deep networks

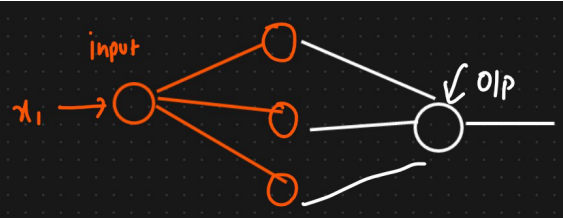
Weight Initializing Techniques

1. Uniform Distribution ✓
2. Xavier / Glorot Initialization ✓
3. Kaiming He Initialization ✓



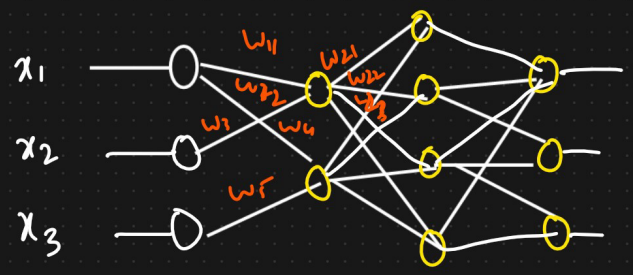
Key Points

1. Weights should be small ✓
2. Weights should not be same ✓
3. Weights should have good variance



input = 1

Output = 1



input = 3

O/p = 3

① Uniform Distribution

$W_{ij} \sim$ Uniform Distribution $\left[\frac{-1}{\sqrt{input}}, \frac{1}{\sqrt{input}}\right]$

$\left[\frac{-1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right]$

② Xavier/Glorot Initialization

Researcher $\longrightarrow$ Xavier Glorot

① Xavier Normal Init

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

$$\sigma = \sqrt{\frac{2}{(input + output)}}$$

② Xavier Uniform

$W_{ij} \sim$ Uniform Distribution $\left[\frac{-\sqrt{6}}{\sqrt{input + Output}}, \frac{\sqrt{6}}{\sqrt{i/p + o/p}}\right]$

③ Kaiming HE Initialization

① He Normal

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

$$\sigma = \sqrt{\frac{2}{\text{input}}}$$

② He uniform

$$W_{ij} \sim \text{Uniform Distribution} \left[-\sqrt{\frac{6}{\text{input}}}, \sqrt{\frac{6}{\text{input}}} \right]$$

CONCLUSION;

Q. why do we need weight Initialization Techniques?

when training a neural network, how we initialize the weights matters a lot - it can make or break our model's ability to learn.

Imp

✅ Key Reasons:

Reason	Explanation
Breaks Symmetry	If all weights are the same (e.g., zeros), neurons learn the same features — no diversity.
Avoids Vanishing/Exploding Gradients	Poor initialization can cause gradients to become too small or too large during backprop.
Speeds Up Convergence	Good initialization helps the model learn faster and more reliably.

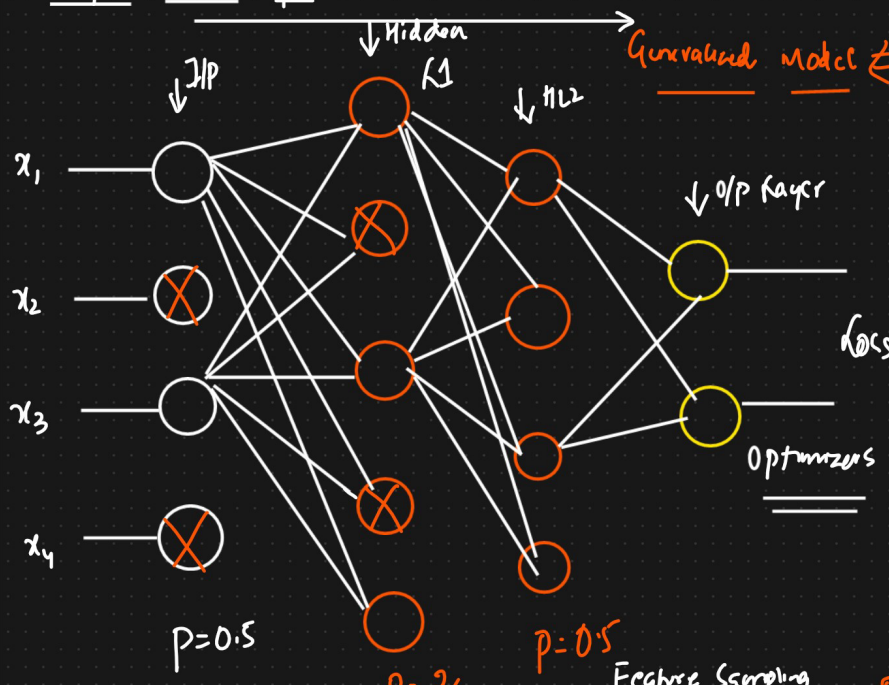
✅ Essential Weight Initialization Techniques

Technique	Use When	Activation Function
Xavier (Glorot)	For balanced input/output layers	`tanh`, `sigmoid`
He Initialization	For deep networks with ReLU-based activations	`ReLU`, `Leaky ReLU`
Random Uniform	Simple or shallow networks	Any (but not recommended for deep networks)

🔁 Quick Recap:

- 🎯 **Xavier** → good for smooth activations
- ⚡ **He** → best for ReLU and its variants
- ⚠️ **Avoid** zeros or constant initializations

Drop Out Layer



Generalized Model ∈ ⇒ Overfitting

Model → Training Acc ⇒ 90%
Test Acc ⇒ 60%

Focus ↓↓

Optimizers

Machine Learning

DT

⇒ Overfitting

⇓

Generalized Model

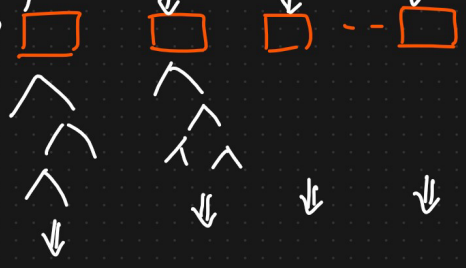
$0 \leq P \leq 1$

⇓

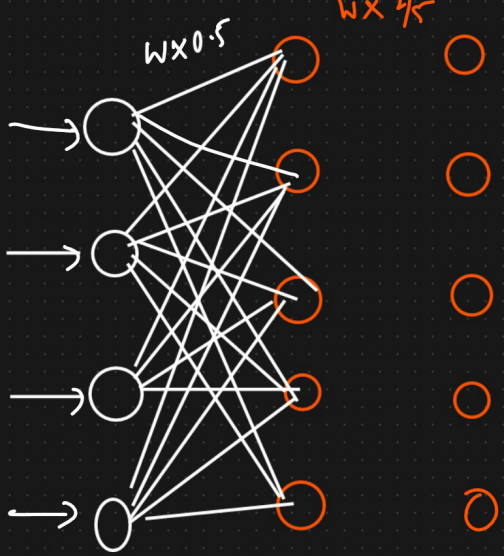
Hyperparameter

Feature Sampling
Row Sampling

Random Forest



Test data



🧠 What is a Dropout Layer (in Deep Learning)?

> A **Dropout layer** is a regularization technique used in neural networks to **prevent overfitting**.

🧪 How it works:

During **training**, the Dropout layer:

- **Randomly turns off (drops)** a percentage of neurons in a layer.
- This means those neurons don't **contribute to forward or backward passes** for that batch.

For example, if you set:

```python
Dropout(rate=0.5)
```

➡️ 50% of the neurons in that layer are randomly "shut off" during each training step.

very important

📌 Why use Dropout?

- It forces the network to **not rely too heavily** on specific neurons.
- Encourages the network to **learn redundant representations**, making it more **robust**.
- Great for **reducing overfitting**, especially in large networks.

🧰 In Code (Keras Example):

```python
from tensorflow.keras.layers import Dense, Dropout

model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))  # Drop 50% of neurons
model.add(Dense(10, activation='softmax'))
```

ANN PRACTICAL

FOR CLASSIFICATION

Tensorflow → library for Deep learning by Google

keras → API on top of Tensorflow

Important; we can use either Tensorflow / Pytorch.

STEP I; Setting up Environment

Bash → conda create -p venv python == 3.13
conda activate venv
pip install -r requirements.txt
install ipykernell

PREPROCESSING

1. Remove unnecessary columns
2. use Label Encoder → Binary Categories
3. use OHE for → Multi Categories

Imp- If we use label Encoder on Multi class, it will assign a=1, b=2, c=0; which will mean for DL b > a & c, so OHE is preffered.

4. Save Encoders in pickle file.
5. Split the Data into train_test
6. Scale the Data (Standard Scaler)
   → Save this also as pickle

VVImp In Tensorflow, we call ANN as sequential Model

Useful parameters; steps

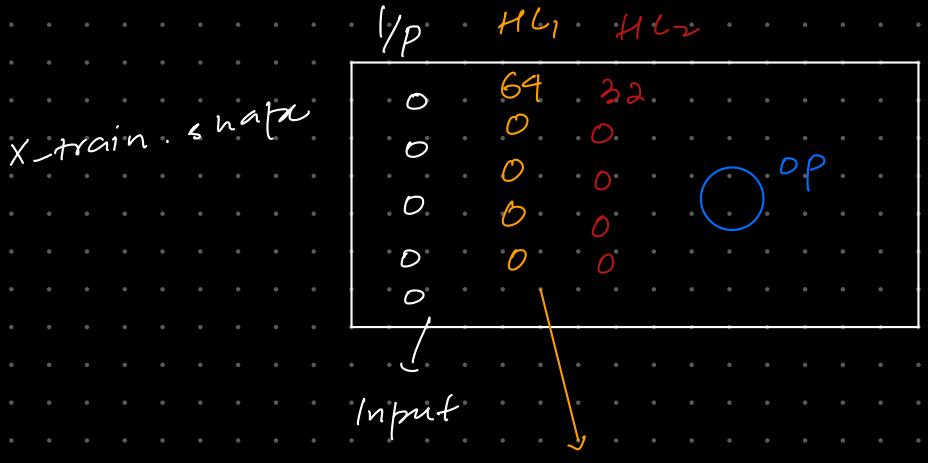
1. sequential
2. Dense ——→ 64 means 64 Neuron
3. Activation fn → Sigmoid, softmax → O/p
   Relu, Tanh → Hidden
4. Optimizers →

5. Loss → GD, SGD

6. Evaluation Metrics

7. Training → Logs → folders
↓
Tensorboard

Libraries → tensorflow.keras.models import Sequential ← ANN
tensorflow.keras.layers import Dense → Nodes
tensorflow.keras.callbacks import EarlyStopping
TensorBoard
↳ logs



Nodes I can setup with Dense

Build ANN Model

```
# Build ANN Model
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],))) ## 1st Input layer connected to 64 neurons - HL1
model.add(Dense(32, activation='relu')) # no shaoe needed here as it is a hidden layer - Hidden layer with 32 neurons - HL2
model.add(Dense(1, activation='sigmoid')) # Output layer with 1 neuron for binary classification
```

→ Setup Optimizer, Loss function, metrics

```
# compile the model with optimizer, loss function and metrics
from tensorflow.keras.optimizers import Adam
model.compile(optimizer=Adam(learning_rate=0.01), loss='binary_crossentropy', metrics=['accuracy'])
```

→ setup Tensorboard & Earlystopping

- ✅ **TensorBoard**: For **real-time monitoring** of training (loss, accuracy, etc.)
- ✅ **EarlyStopping**: To **automatically stop training** when performance stops improving (prevents overfitting and saves time) → stops when model reaches convergence

→ mention epochs

```
# Set up the tensorboard callback
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

# Set up early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
```

Ruko zara sabr karo till 5 epochs

→ Train the Model file

```
history = model.fit(X_train_scaled, y_train,
                    validation_data=(X_test_scaled, y_test),
                    epochs=100,
                    callbacks=[tensorboard_callback, early_stopping])
```

✓ 16.5s

```
Epoch 1/100
250/250 [==============================] - 2s 7ms/step - loss: 0.5535 - accuracy: 0.7806 - val_loss: 0.5473 - val_accuracy: 0.8030
Epoch 2/100
250/250 [==============================] - 1s 6ms/step - loss: 1.2733 - accuracy: 0.7465 - val_loss: 1.9880 - val_accuracy: 0.6815
Epoch 3/100
250/250 [==============================] - 1s 6ms/step - loss: 2.8616 - accuracy: 0.7368 - val_loss: 3.8312 - val_accuracy: 0.7565
Epoch 4/100
250/250 [==============================] - 1s 6ms/step - loss: 3.3271 - accuracy: 0.7303 - val_loss: 17.8678 - val_accuracy: 0.6560
Epoch 5/100
250/250 [==============================] - 1s 6ms/step - loss: 6.7133 - accuracy: 0.7289 - val_loss: 3.3616 - val_accuracy: 0.8045
Epoch 6/100
250/250 [==============================] - 1s 6ms/step - loss: 8.6875 - accuracy: 0.7306 - val_loss: 9.0012 - val_accuracy: 0.7670
Epoch 7/100
250/250 [==============================] - 1s 6ms/step - loss: 13.4958 - accuracy: 0.7394 - val_loss: 7.7190 - val_accuracy: 0.7380
Epoch 8/100
250/250 [==============================] - 1s 6ms/step - loss: 24.6641 - accuracy: 0.7144 - val_loss: 17.9428 - val_accuracy: 0.7920
Epoch 9/100
250/250 [==============================] - 1s 6ms/step - loss: 18.2670 - accuracy: 0.7521 - val_loss: 6.6817 - val_accuracy: 0.8015
Epoch 10/100
250/250 [==============================] - 1s 6ms/step - loss: 19.5793 - accuracy: 0.7377 - val_loss: 43.4841 - val_accuracy: 0.6100
Epoch 11/100
250/250 [==============================] - 1s 6ms/step - loss: 25.3717 - accuracy: 0.7340 - val_loss: 17.2696 - val_accuracy: 0.7635
```

→ Save model as .h5

```
# Save model file   as .h5 is compatible with Keras
model.save('customer_churn_model.h5')
```

→ Monitor logs in Tensorboard.

```
# Load TensorBoard Extension
%load_ext tensorboard

# Launch TensorBoard
%tensorboard --logdir logs/fit
```

Finally Get Predictions

→ Import pkl files & .h5 file

→ Get prediction → Ready for production.

→ Create app.py which runs all the trained models

For Regression

```python
### Load the trained model, scaler pickle,onehot
model=load_model('customer_churn_model.h5')

## load the encoder and scaler
with open('ohe_geo.pkl','rb') as file:
    label_encoder_geo=pickle.load(file)

with open('label_encoder_gender.pkl', 'rb') as file:
    label_encoder_gender = pickle.load(file)

with open('scaler.pkl', 'rb') as file:
    scaler = pickle.load(file)
```

In [65]:
```python
## PRedict churn
prediction=model.predict(input_scaled)
prediction
```

```
WARNING:tensorflow:5 out of the last 5 calls to <function Model.make_predict_function.<locals>.predict_function at 0x33227f560> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more details.
WARNING:tensorflow:5 out of the last 5 calls to <function Model.make_predict_function.<locals>.predict_function at 0x33227f560> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more details.
1/1 [==============================] - 0s 37ms/step
```

Out[65]:
```
array([[0.01600381]], dtype=float32)
```

In [66]:
```python
prediction_proba = prediction[0][0]
```

In [67]:
```python
prediction_proba
```

Out[67]:
```
0.016003806
```

In [68]:
```python
if prediction_proba > 0.5:
    print('The customer is likely to churn.')
else:
    print('The customer is not likely to churn.')
```

```
The customer is not likely to churn.
```

CNN

Convolutional Neural Network is the type of Deep learning model that is especially powerful for working with images, videos etc.

🧩 Why "Convolutional"?

?

Because it uses a mathematical operation called **convolution** to extract **features** (like edges, textures, shapes) from input data—especially images.

```
Input Image Patch:
[1, 2, 0]
[4, 5, 6]
[7, 8, 9]
```

And a **3x3 filter** (also called kernel):

```csharp
Filter:
[1, 0, -1]
[1, 0, -1]
[1, 0, -1]
```

kernel

➡️ Convolution means:
Multiply each value from the image and the filter **element-wise**, then **sum them up**.

```markdown
Result:
= 1*1 + 2*0 + 0*(-1)
+ 4*1 + 5*0 + 6*(-1)
+ 7*1 + 8*0 + 9*(-1)

= 1 + 0 + 0 + 4 + 0 -6 + 7 + 0 -9 = -3
```

That **-3** becomes a single pixel in the **output feature map**.

You move the filter over the whole image to generate more values.

Purpose of Convolution =
Detect edges, corners, texture

🔍 Key Components of a CNN:

1. **Input Layer** – Raw image data (e.g., 224×224 pixels).
2. **Convolutional Layer** – Applies filters to detect patterns (like edges or corners). → (Kernel)
3. **Activation Function** – Usually **ReLU**, adds non-linearity.
4. **Pooling Layer** – Reduces dimensionality (e.g., max pooling).
5. **Fully Connected Layer** – Final layer(s) to make predictions. → Flattening
6. **Output Layer** – Gives final result (e.g., classification label like "cat" or "dog").

📸 What CNNs Are Used For:

- Image classification (e.g., recognizing objects in photos)
- Face recognition
- Self-driving cars (lane detection, obstacle recognition)

🧠 Simple Analogy:

Think of CNNs like a brain that first **detects edges**, then **shapes**, then **objects** — like how humans recognize patterns step by step.

Main Concept

is to mimim how human brain analyzes images

Cerebral cortex → visual cortex (Layers).

CNN Terminology

Q. What is an image?

Image is a grid of pixel values

→ For B & W image → each pixel = 0-255

0 = Black, 255 = White

Single channel (one value for each pixel)

e.g., 5×5 image = 5×5 vector

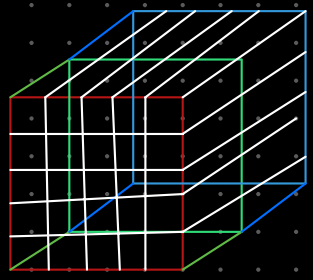
→ For a colored image (RGB)

Each pixel has 3 values (3 channel)

R G B

e.g., 5×5 image = 5×5×3 tensor

Imp.



5×5×3 → 3 channel

🔷 For Grayscale Images:

- **Pixel value = 0 → black**
- **Pixel value = 255 → white**
- Values in between represent **shades of gray**
  - 127 ≈ medium gray

🔷 For RGB Images:

Each pixel has **three channels**:

- Red (R), Green (G), Blue (B)
- Each ranges from **0 (no intensity)** to **255 (full intensity)**

Example:

- **(0, 0, 0)** → Black (no light at all)
- **(255, 255, 255)** → White (full intensity of all colors)
- **(255, 0, 0)** → Red
- **(0, 255, 0)** → Green
- **(0, 0, 255)** → Blue

Q. What is Tensor?

A tensor is a container that can hold data in n Dimensions (any Dimensions)

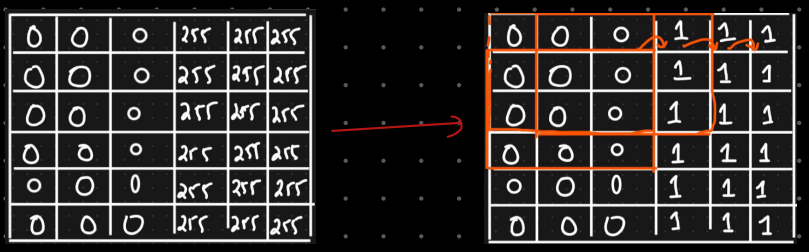
🔢 Examples:

- **0D Tensor → Scalar →** `7`
- **1D Tensor → Vector →** `[3, 5, 7]`
- **2D Tensor → Matrix →** `[[1, 2], [3, 4]]`
- **3D Tensor → Stack of matrices → Like a color image** `[Height x Width x Channels]`

Steps

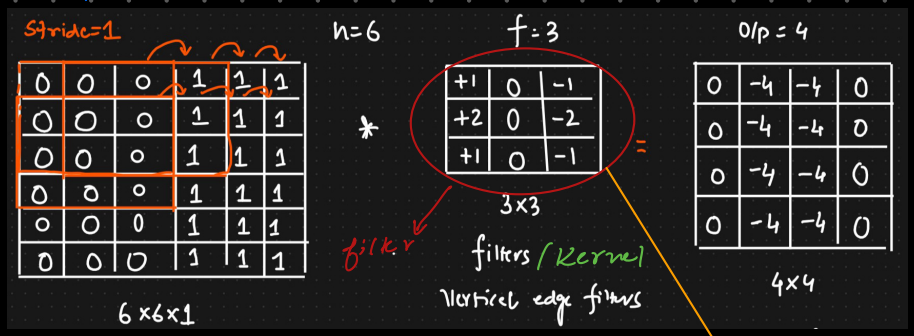
→ **Step I**; Input image pixel info

Step II; Normalize ⟶ Divide by 255



Step III; Convolution operation in CNN; to detect edges, corners and texture.

Stride = 1 means Right by 1 step.



This one is Vertical edge filter, we can similarly many other filter for texture corner etc.

Imp

Q// how come we got O/p as 4x4 ?

So we have formula ;

↳ $n - f + 1$ (n: input, f: filter)

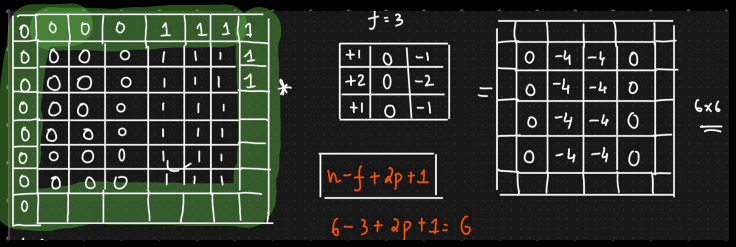
$\Rightarrow 6 - 3 + 1 = 4$

Filters : are used to get info out of images

Q// What is padding ; → As seen above 6x6 → 4x4

padding is used to prevent the loss of image information (pixels) during convolution operations.

✅ **In short**: Padding adds extra pixels (usually zeros) around the image to preserve size and information during convolution.



New formula

$= n + 2p - f + 1$

$= 6 + 2(1) - 3 + 1$

$= 8 - 3 + 1 = 6$

$n = 6$

$p = 1$

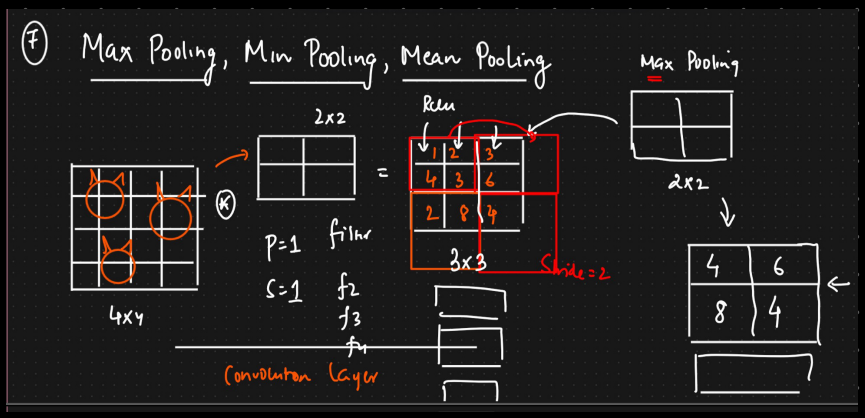
$f = 3$

Types

Zero Padding	Pads with zeros around the image. Most common.
Reflect Padding	Pads with reflected values of the border.

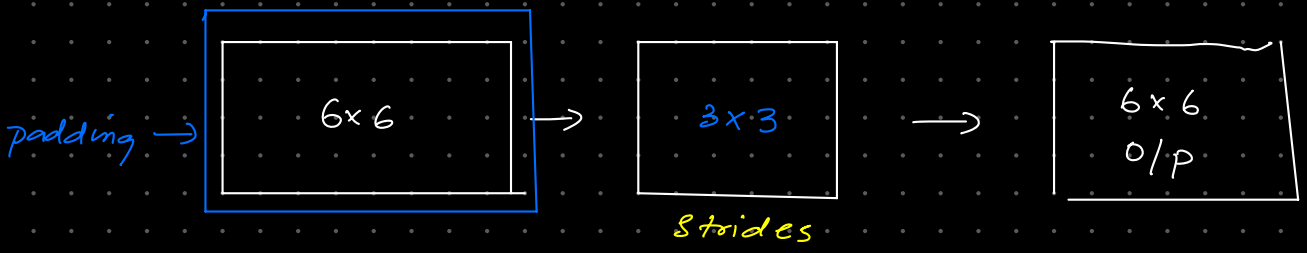
POOLING = is a technique in CNN to reduce dimensionality

Types =



CONCLUSION

1. Input Image
2. Normalization (0-1) where 1 = 255
3. Convolution method (apply filter / kernel)



4. Activation function (Relu)
5. pooling → to reduce dimensionality
   Max, Min or Mean pooling
6. Flattening → elongating pooling output

2	5
1	6

= 2, 5, 1, 6

7. Finally → ANN classification (Dense layers).

Conclusion

✅ Step-by-Step: Your CNN Understanding

✅ 1. Input Image

> Let's say a grayscale image: `6x6` matrix of pixel values (0–255)

✓ Correct.

✅ 2. Normalization

> Normalize pixel values: `255 → 1`, `0 → 0`

✓ Correct. Sometimes values are scaled between `0–1` or `-1 to 1`.

✅ 3. Apply Convolution (Filter/Kernel 3x3)

> To detect edges, textures, corners, etc.

✓ Correct.

```
Input Image (6x6) → Normalize →
Convolution (3x3 filter) →
ReLU →
Pooling (e.g., Max Pooling) →
Flatten →
Fully Connected Layers (ANN) →
Output with Activation (sigmoid/softmax)
```

🔧 4. You said: Output will be 6x6

Not always — usually the output size **shrinks** (unless you use padding).

Formula (without padding):

arduino — Copy — Edit

```
Output size = (Input size − Kernel size) + 1
            = (6 − 3) + 1 = 4
→ So, output = 4x4
```

✅ But if **padding = same**, then output stays 6x6.

❓ CONFUSION: Why ReLU after Convolution?

Let's clarify:

✅ **After each convolution, we apply ReLU (or another activation function).**

🔍 **Why?**

Because convolution may produce **negative values**, and **ReLU** keeps only the positive ones:

lua — Copy — Edit

```
ReLU(x) = max(0, x)
```

> This introduces **non-linearity** into the model. Without this, your entire CNN would just be one big linear operation, and it wouldn't be able to learn complex patterns.

So yes — we apply **ReLU right after convolution, before pooling.**

v.v.imp

✅ 5. Pooling Layer (e.g., Max Pooling)

> To reduce dimensions, keep important features

✓ Correct. Typically reduces size (e.g., 4x4 → 2x2)

✅ 6. Flatten the Output

> Convert the 2D/3D tensor to 1D vector for input into fully connected (ANN) layer

✓ Correct.

✅ 7. Apply Fully Connected (Dense) Layers

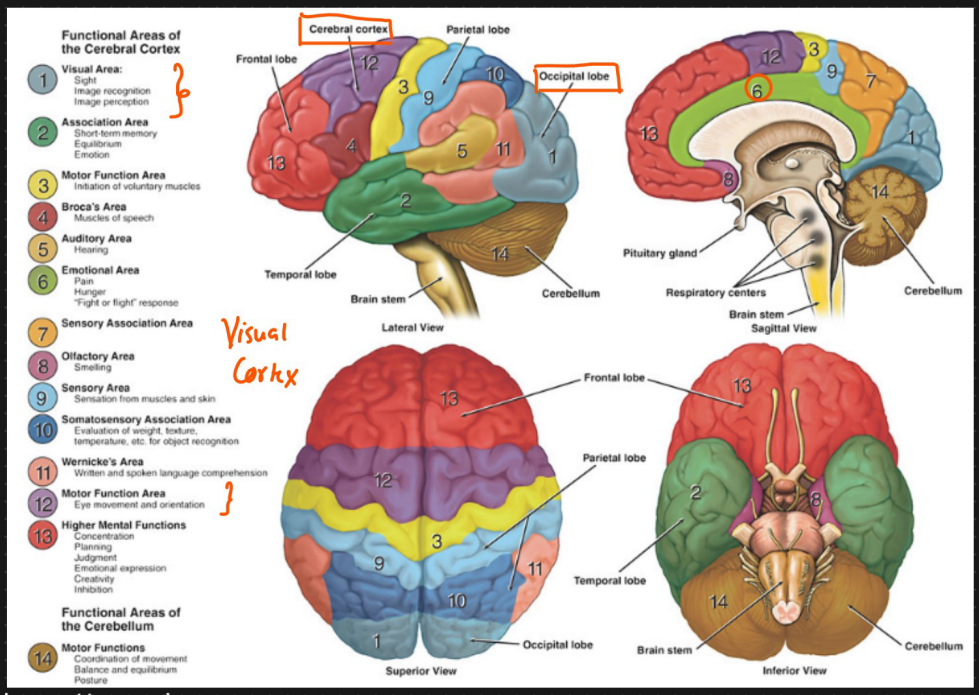
> Feed into ANN, apply activation functions (like **ReLU**, **softmax**, etc.)

✓ Correct.

✅ Final Layer Activation:

- For **binary classification:** `sigmoid`
- For **multi-class classification:** `softmax`

Convolutional Neural N/w



https://www.dana.org

① ANN → Supervised Learning → [Classification, Regression]

Dataset : I/p features O/p

② CNN : I/p => Images Eg: Image classification, Object Detection, Segmentation

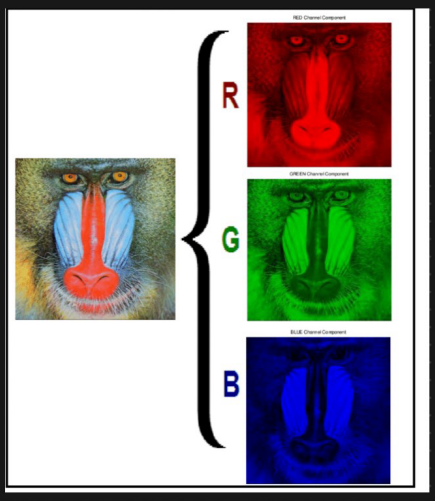
② Cerebral Cortex And Visual Cortex

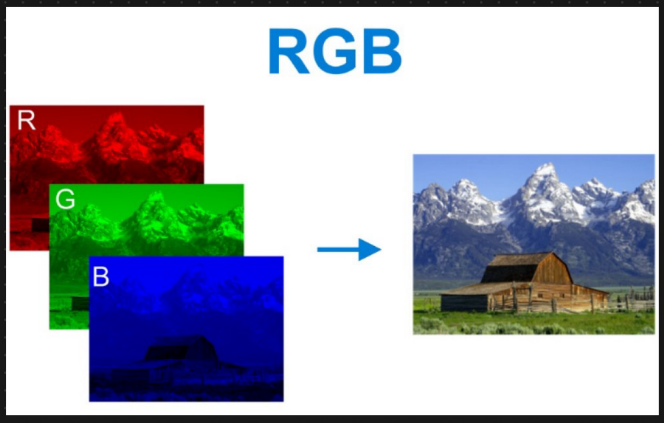
Visual Cortex (V1-V5) [Region of the brain that receives, integrates and processes visual information relayed from the retinas].

- V1 → Primary Visual Cortex [Orientation of edges And Lines]
- V2 [Differences in color, complex patterns, object orientation]
- V3, V4, V5 [Object Recognition]

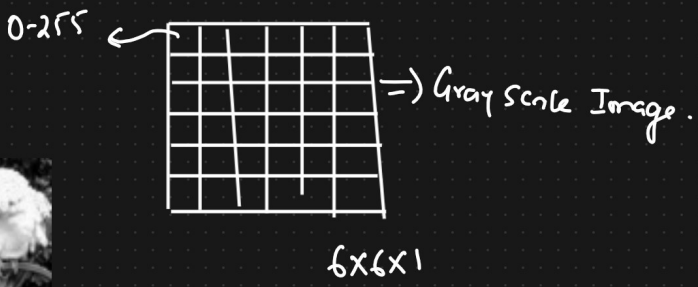
Visualize the Image

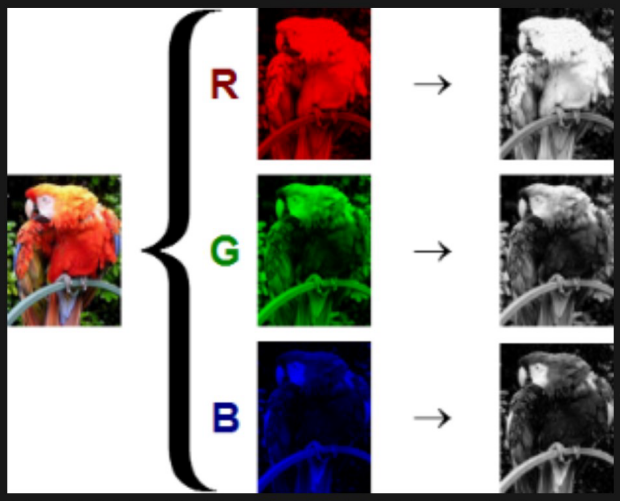
③ RGB Images And Gray Scale Images



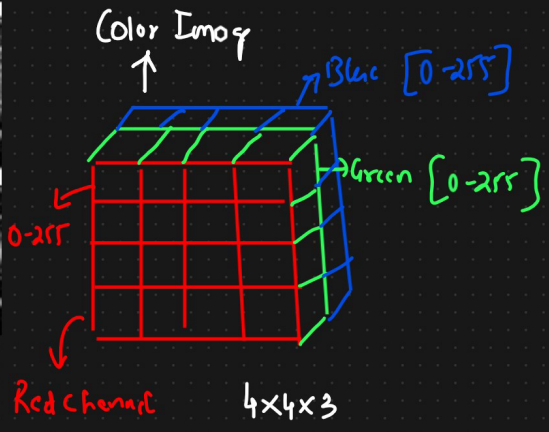


https://www.researchgate.net/





https://commons.wikimedia.org/

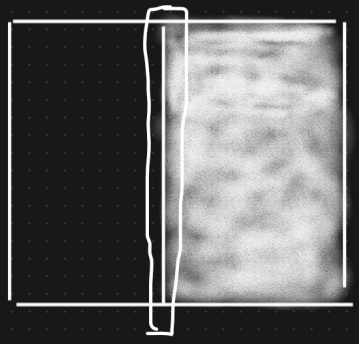


④ Convolution Operation In CNN

→ (0,1)

0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255

6X6X1



Imp

Step 1

(1) Normalize

Divide by 255

+1	+2	+1
0	0	0
-1	-2	-1

Imp Convolution operation

Stride=1

0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1

6 x 6 x 1

n = 6

*

f = 3

+1	0	-1
+2	0	-2
+1	0	-1

3 x 3

filters

Vertical edge filters

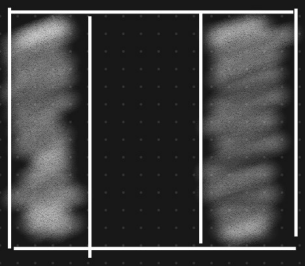
=

o/p = 4

0	-4	-4	0
0	-4	-4	0
0	-4	-4	0
0	-4	-4	0

4 x 4

$n - f + 1 = 6 - 3 + 1 = 4$



⇐

2rr	0	0	2rr
2rr	0	0	2rr
2rr	0	0	2rr
2rr	0	0	2rr

⇓

⑤ Padding In CNN

① Zero padding

② Neighbour padding

0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	
0	0	0	0	1	1	1	
0	0	0	0	1	1	1	
0	0	0	0	1	1	1	
0							

*

f = 3

+1	0	-1
+2	0	-2
+1	0	-1

=

	0	-4	-4	0	
	0	-4	-4	0	
	0	-4	-4	0	
	0	-4	-4	0	

6 x 6

6x6 n=6 8x8

$n - f + 2p + 1$

$6 - 3 + 2p + 1 = 6$

$3 + 2p + 1 = 6$

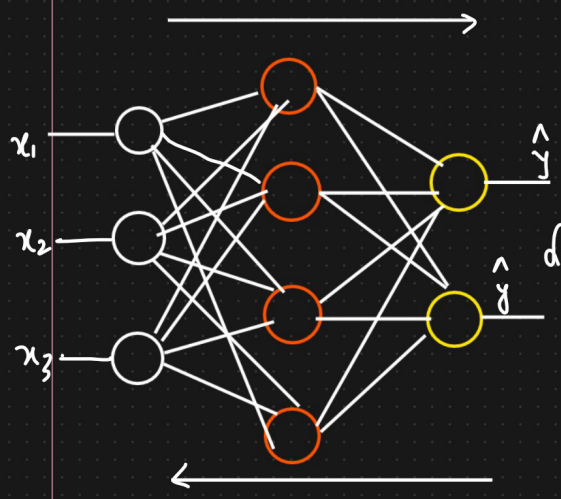
$2p = 6 - 4$

$p = 2/2 = 1$

7x7 3x3 7x7

How much padding you need to apply?

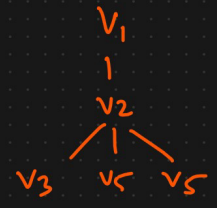
⑥ Operation of CNN Vs ANN



$$z = w^T x_i + b$$

$$relu(z)$$

Visual Cortex



Image

0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1

$*$

+1	0	-1
+2	0	-2
+1	0	-1

f_1

f_2

f_3

f_n

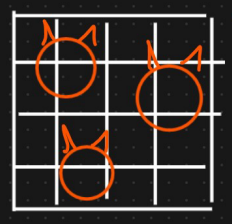
$=$

→ Relu operation $max(0, x)$

0	-4	-4	0
0	-4	-4	0
0	-4	-4	0
0	-4	-4	0

Convolution Layer

⑦ Max Pooling, Min Pooling, Mean Pooling



4x4

$\circledast$

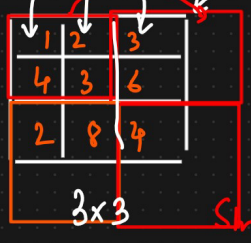
2x2 filter

P=1

S=1

$=$

Relu



3x3

Stride = 2

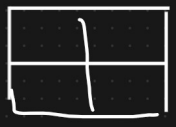
f_2

f_3

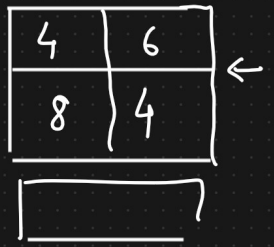
f_n

Convolution Layer

Max Pooling

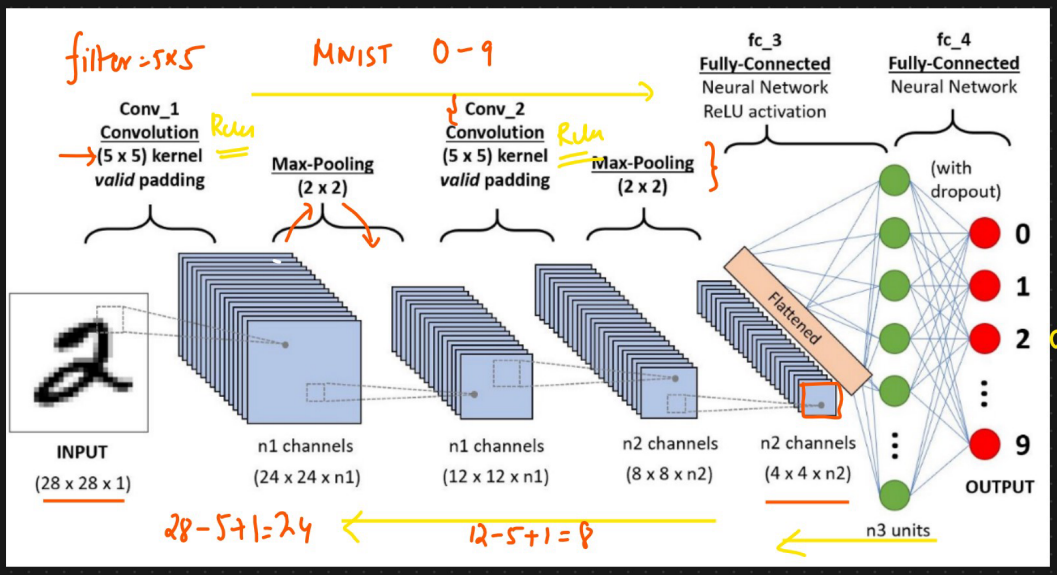


2x2



⑦ Location Invariant

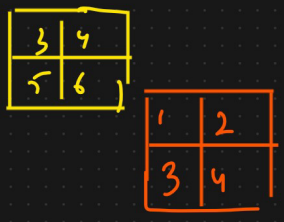
⑧ Fully Connected Layer In CNN [Flattened Layer]

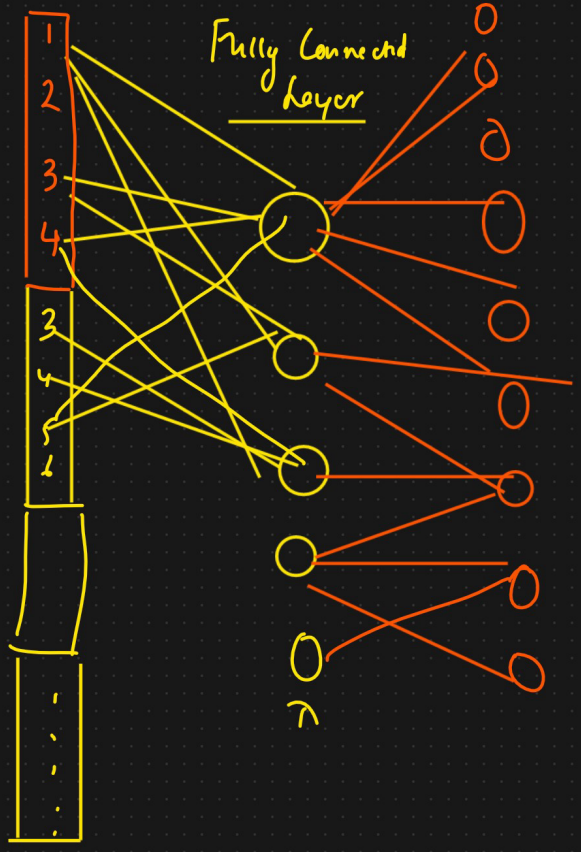


loss function ↓↓

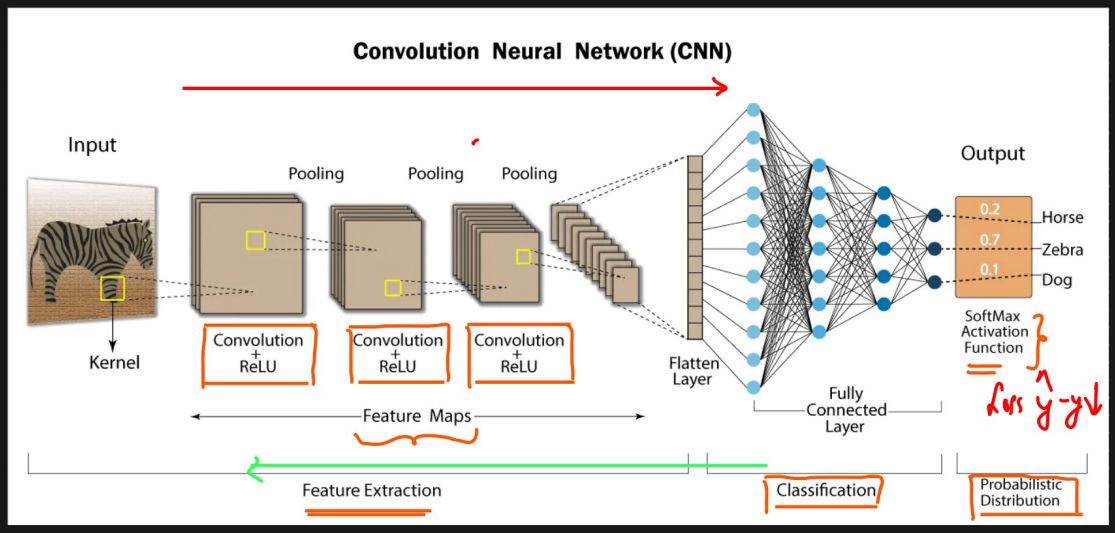
Optimizer

n-f+1

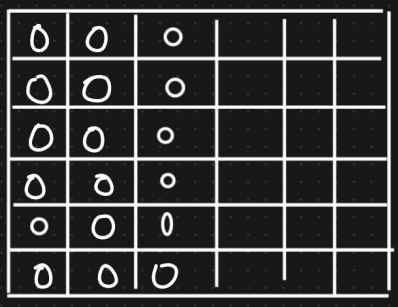




⑨ CNN Complete Example



https://developersbreach.com/convolution-neural-network-deep-learning/



*

+1	0	-1
+2	0	-2
+1	0	-1

0	-4	-4	0
0	-4	-4	0
0	-4	-4	0
0	-4	-4	0