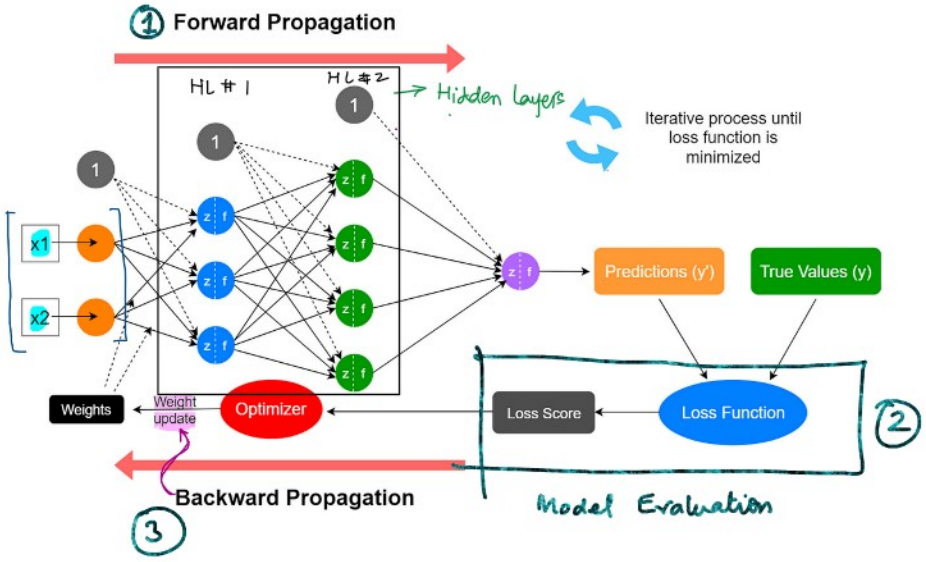
Multiple Layer Perceptron (MLP)

05 October 2025 09:56



Multiple Layer Perceptron

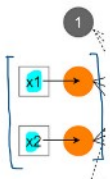
A multi-layer perceptron is class of ANN that consists of multiple layer (hidden layers) of neurons in a feed-forward network.

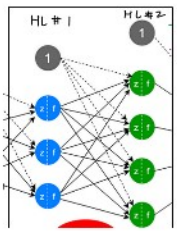
Architecture of MLP

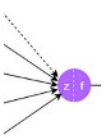
at least (minm) one hidden layer is present in MLP.

Input layer → Hidden layer(s) → output layer

- receives the input data (after flattening) and connect each of the input features (columns) to neurons in the input layer
- one or more hidden layer(s) between input and output layers where the actual learning happens.
- produces the final prediction
  - Regression
  - Classification
    - Binary
    - Multi-class.







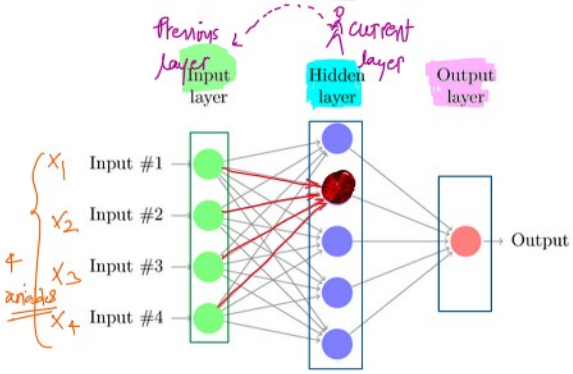
- No learning or calculation

Neural Network Terminologies

Fully Connected Network

- a layer where each and every neuron is connected to every neuron in the preceding/previous layer

Neural Network Terminologies



(self-read)

Single Layer Perceptron (SLP) vs Multi Layer Perceptron (MLP)		
Feature	**SLP (Single Layer Perceptron)**	**MLP (Multi Layer Perceptron)**
Layers	1 layer (input → output)	2 or more layers (input → hidden(s) → output)
Neurons	No hidden layer, just output neuron(s)	One or more hidden layers with multiple neurons
Functions	Can only solve **linearly separable** problems	Can solve **non-linear** and more complex problems
Learning	Simple weights & bias update (perceptron rule)	Uses backpropagation and more complex optimization
Representation	Linear decision boundaries	Can learn complex, non-linear boundaries
Use cases	Very basic classification tasks	Most modern neural network applications

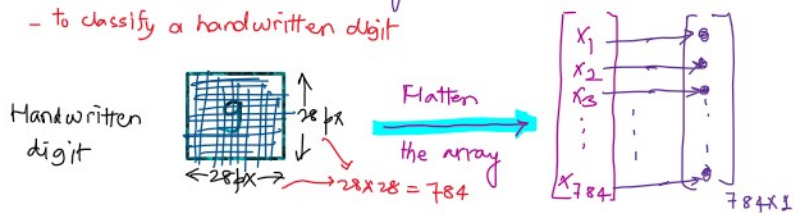
Computation: simple and fast (but just a proof of concept)

more computationally intensive than SLP (state of art NN model)

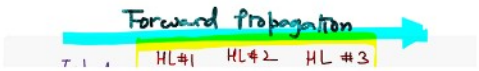
Working of MLP

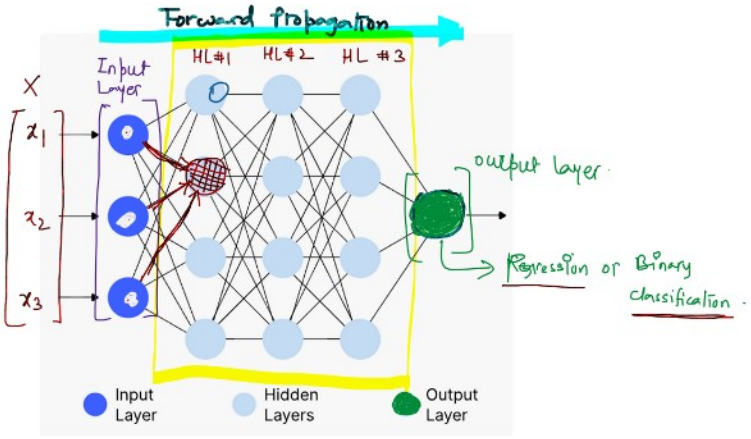
Step #1 Getting input data ready.

- to classify a handwritten digit



Step #2 Take (weighted sum of inputs + bias) → in hidden layer





HL#1 → 1st

Each neuron in the 1st HL receives a weighted sum of inputs or features (x_1, x_2, x_3) from the input layer. ⊕ (bias)

Computing the weighted sum of inputs along with bias:

$$Z_j^l = W_{ij}^l x_i + b_j^l$$

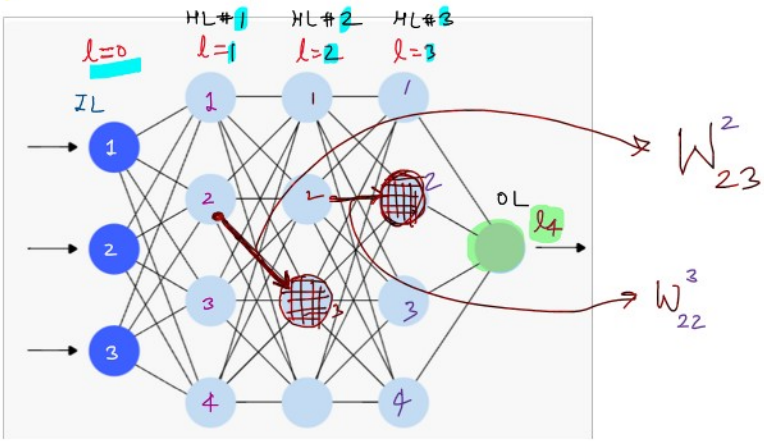
where

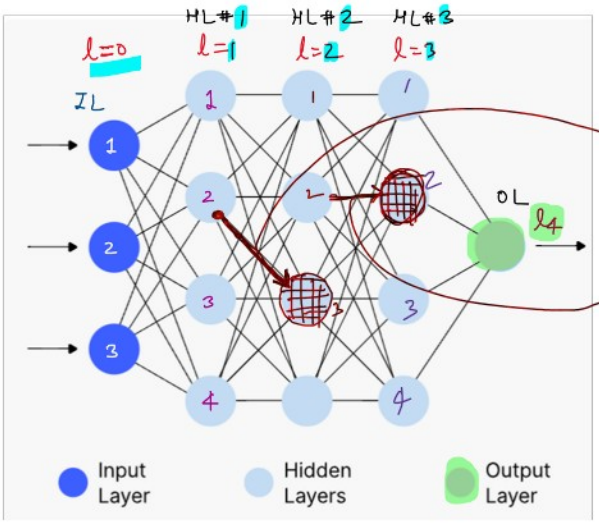
Z_j^l : is the weight sum for neurons j in the layer l

W_{ij}^l : is the weight between neuron i from the previous layer (l-1) and neuron j in the current layer (l)

x_i : is the input from neuron i in the previous layer

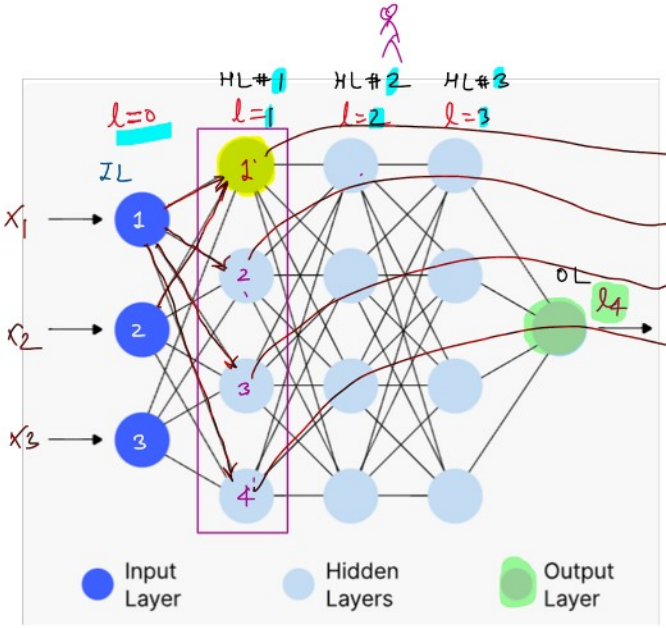
b_j^l : is the bias associated with neuron j in the layer l





W^{2}_{23}

W^{3}_{22}



$$Z_1^{(1)} = W_{11}^{(1)} X_1 + W_{21}^{(1)} X_2 + W_{31}^{(1)} X_3 + b_1^{(1)} \leftarrow 1^{st} \text{ neuron } | \ l=1 \quad (j=1)$$

$$Z_2^{(1)} = W_{12}^{(1)} X_1 + W_{22}^{(1)} X_2 + W_{32}^{(1)} X_3 + b_2^{(1)} \leftarrow 2^{nd} \text{ neuron } | \ l=1 \quad (j=2)$$

$$Z_3^{(1)} = W_{13}^{(1)} X_1 + W_{23}^{(1)} X_2 + W_{33}^{(1)} X_3 + b_3^{(1)} \leftarrow 3^{rd} \text{ neuron } | \ l=1$$

$$Z_4^{(1)} = W_{14}^{(1)} X_1 + W_{24}^{(1)} X_2 + W_{34}^{(1)} X_3 + b_4^{(1)} \leftarrow 4^{th} \text{ neuron } | \ l=1$$

$$Z_1^{(1)} = W_{11}^{(1)} X_1 + W_{21}^{(1)} X_2 + W_{31}^{(1)} X_3 + b_1^{(1)} \leftarrow 1^{st} \text{ neuron } | \ l=1 \quad (j=1)$$

$$Z_2^{(1)} = W_{12}^{(1)} X_1 + W_{22}^{(1)} X_2 + W_{32}^{(1)} X_3 + b_2^{(1)} \leftarrow 2^{nd} \text{ neuron } | \ l=1 \quad (j=2)$$

$$Z_3^{(1)} = W_{13}^{(1)} X_1 + W_{23}^{(1)} X_2 + W_{33}^{(1)} X_3 + b_3^{(1)} \leftarrow 3^{rd} \text{ neuron } | \ l=1$$

$$Z_4^{(1)} = W_{14}^{(1)} X_1 + W_{24}^{(1)} X_2 + W_{34}^{(1)} X_3 + b_4^{(1)} \leftarrow 4^{th} \text{ neuron } | \ l=1$$

For layer #1

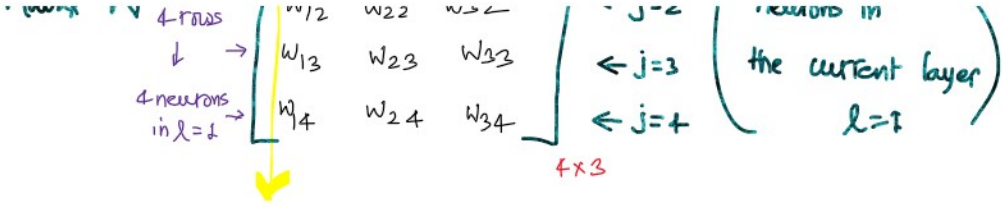
3 input variables (in $l=0$) ↑ 3 columns

$[a_{ij}]$ → j^{th} column, i^{th} row → RC: column ↑, Row ↓

Weight Matrix $W =$ (4 rows → 4 neurons)

$$\begin{bmatrix} W_{11} & W_{21} & W_{31} \\ W_{12} & W_{22} & W_{32} \\ W_{13} & W_{23} & W_{33} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

$\leftarrow j=1$, $\leftarrow j=2$, $\leftarrow j=3$ (neurons in the current layer)



Input vector : $X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{3\times 1}$ Bias vector $b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}_{4\times 1}$ layer # 1.

$$Z_1 = W * X + b = [WX] + b$$

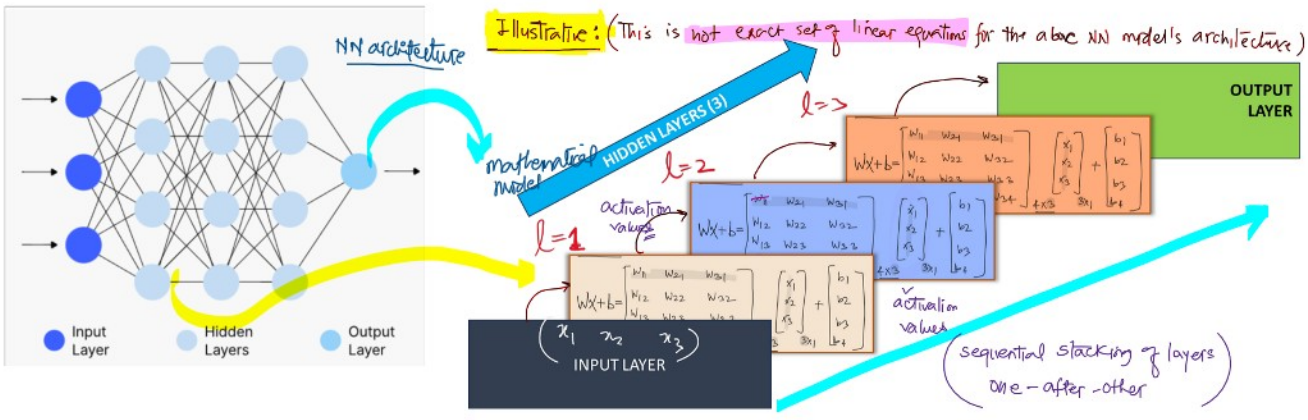
$4\times 3 * 3\times 1 \rightarrow 4\times 1$; $4\times 1 + 4\times 1$

addition is valid.

$$Z^{(1)} = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \\ w_{14} & w_{24} & w_{34} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$ for layer = 1.

$$z_1 = w_{11}x_1 + w_{21}x_2 + w_{31}x_3 + b_1$$

$$z_3 = w_{13}x_1 + w_{23}x_2 + w_{33}x_3 + b_3$$



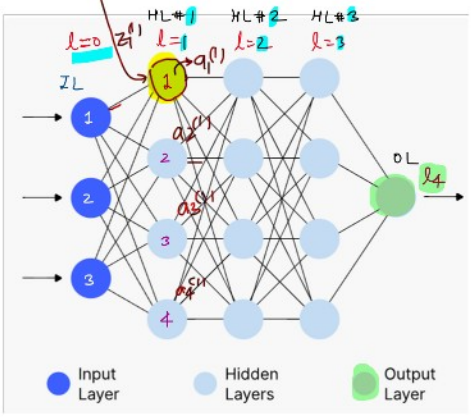
Task: Update the equations for l=2 and l=3.

— Manish Kaushik

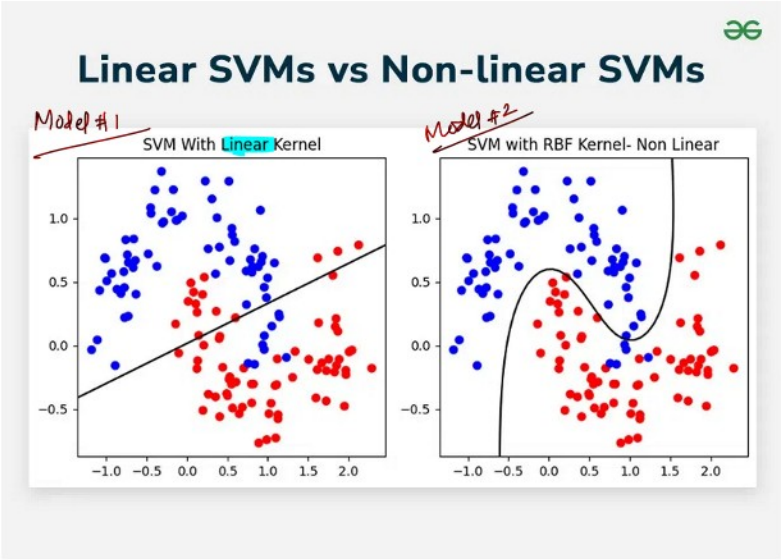
For layer = 1 and 1st neuron (j=1)

$$z_j^{(l)} = w_{11}^{(l)} x_1 + w_{21}^{(l)} x_2 + w_{31}^{(l)} x_3 + b_j^{(l)} \leftarrow j=1 \Rightarrow 1^{st} \text{ neuron } / l=1$$

$$z_1^{(l)} \rightarrow \boxed{\text{activation function}} \rightarrow a_1^{(l)}$$



Activation Functions



→ Model # 2 has better accuracy as kernel is polynomial → (non-linear)

Purpose: Activation function introduces non-linearity into the neural n/w so it can learn complex patterns (otherwise, the n/w would behave like a linear regression model regardless of its depth (many hidden layers)

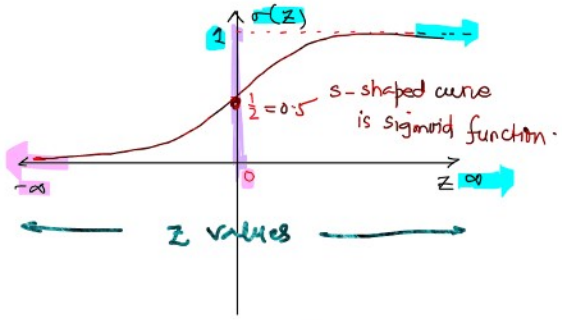
1. Sigmoid Activation Function

$$f(z) = \sigma(z) = \frac{1}{1+e^{-z}}$$: primarily used for binary classification model.

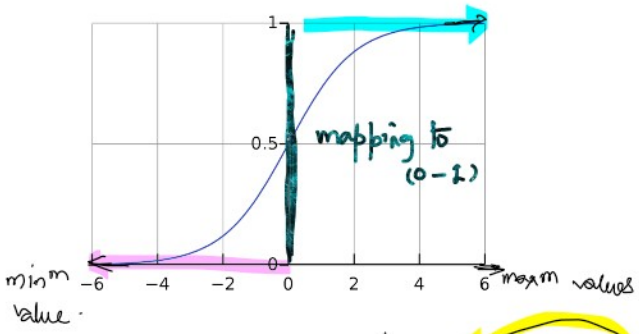
$z=0$; $\sigma(0) = \frac{1}{1+e^{0}} = \frac{1}{1+1} = \frac{1}{2} = 0.5$

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

$z \to -\infty$; $\sigma(-\infty) = \frac{1}{1+e^{-(-\infty)}} = \frac{1}{1+e^{\infty}} = \frac{1}{\infty} \to 0$



$$\begin{cases} z \geq 0.5 \rightarrow \text{class \#1} \\ z < 0.5 \rightarrow \text{class 0} \end{cases}$$



Note: Technically speaking, a sigmoid is any s-shaped curve that flattens out near its minimum and maximum values.

tanh: hyperbolic tangent

↳ RNN / LSTM → will be discussed here.

Range of sigmoid activation function: (0, 1)

Application: Binary classification → (output layer) → probabilistic outputs

Pros:

- a smooth gradient → sigmoid is differentiable function
- and range is very useful for probabilities (0,1)

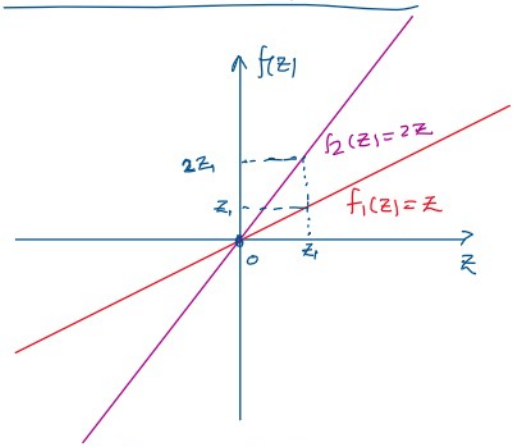
Cons: Pro-tip* → (RNN/LSTM)

- Vanishing Gradient: → is a problem for very large/small input values
  ↳ (will be discussed in RNN)
- Not zero-centered (all outputs are going to be positive)
  ↳ which eventually leads to slow convergence

} FYI

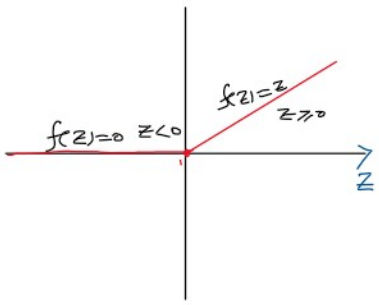
2. ReLU: Rectified Linear Unit

Linear Activation Function

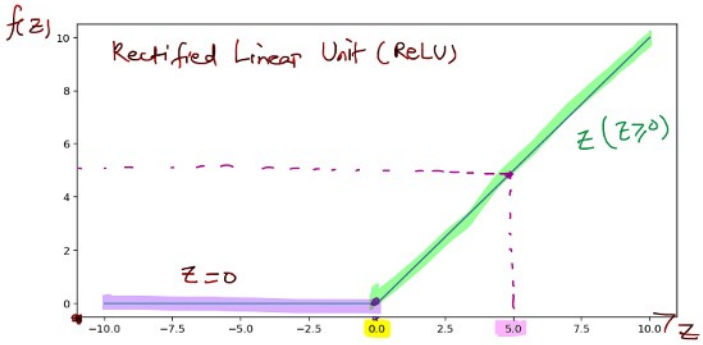


- output is proportional to the input
- lines pass through the origin → $y = mx$ form
- since, lines are passing through origin its usage is limited in deep learning given that it lacks non-linearity

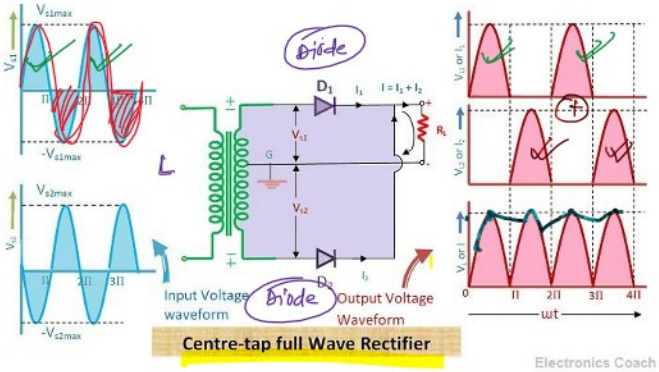
Rectified Linear Unit



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



* Pro-tip Why is it called rectified ??



It's called 'rectified' because of how it fixes or clips the -ve values to zero, just like a rectifier in electronics engg. → which only allows positive parts of a signal to pass.

Similarly, ReLU activation functions rectifies the input by

- keeping all positive values
- and replacing all negative values with zero.

$$ReLU(z) = \max(0, z)$$

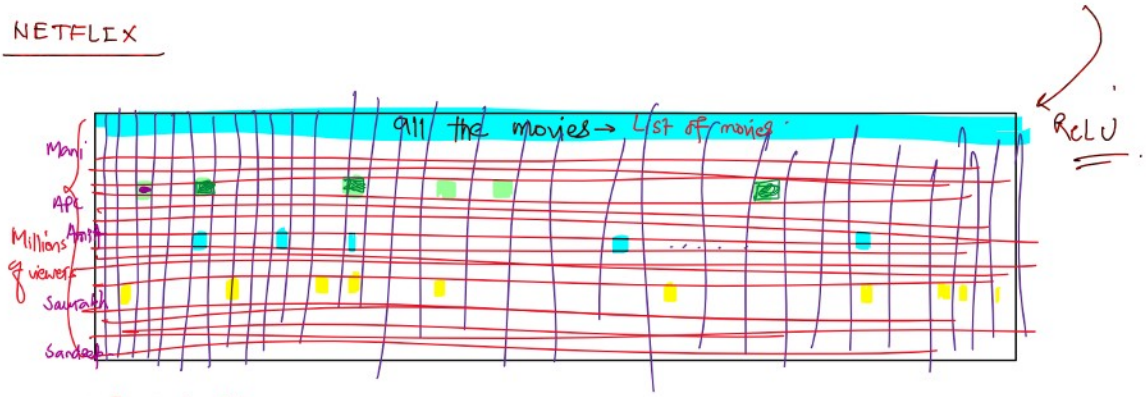
In some sense, ReLU is used as activation function to introduce non-linearity but at the same time it also adds sparsity * Pro-tip

What is sparse data ?

sparse data or sparsity is an important concept in both ML and DL specially when dealing with large datasets.

- Sparse data means most of the values in data are zero or empty.

NETFLIX



- In such datasets, most of the entries are going to be '0' → sparse data.

Note: Text, recommendations, some computer vision problems, NLP tasks naturally produce sparse data.

ReLU adds non-linearity by cutting-off the -ve values and output becomes sparse.

↓

many neurons output '0' values

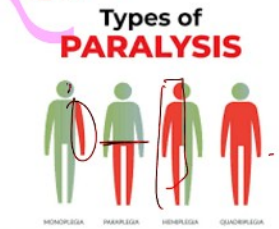
Given a neuron's (weighted sum of inputs + bias) becomes -ve (-ve) for all inputs → ReLU will map it to '0'.

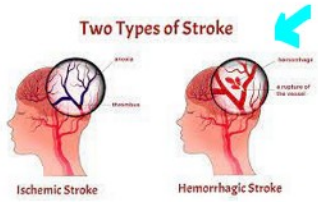
↓

Dying 'ReLU' problem

Analogy → A person getting brain stroke.

© afc 'Dying ReLU'





↳ (ray of hope → Physiotherapy)

Leaky ReLU

Dying ReLU Problem

- If a neuron's input ($Wx+b$) becomes negative, ReLU outputs 0
- During backpropagation, the gradient of ReLU is also 0 for negative inputs

$$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

(Neuron)

→ it stops updating as no gradient and it stays dead and always output '0' forever.

Example:

If a neuron's weights along with bias produce

$z = -3$ then

output: $f(z) = 0$

gradient: $f'(z) = 0$

→ No weight update during backpropagation

- The neuron is effectively dead — permanently inactive.

Pros of using ReLU

Even though neurons can be dead while using ReLU as AF and can lead to "dying ReLU problem" however it still the most popular activation functions in hidden layers because:

✓ # simplicity ⟶ Mathematical formulation: $\max(0, z)$ → very cheap to compute

simplicity → Mathematical formulation: max(0, z) → very cheap to compute → computational cost is less

Faster Convergence → ReLU doesn't saturate for large positive values → gradients stay/keep changing leading to faster convergence.

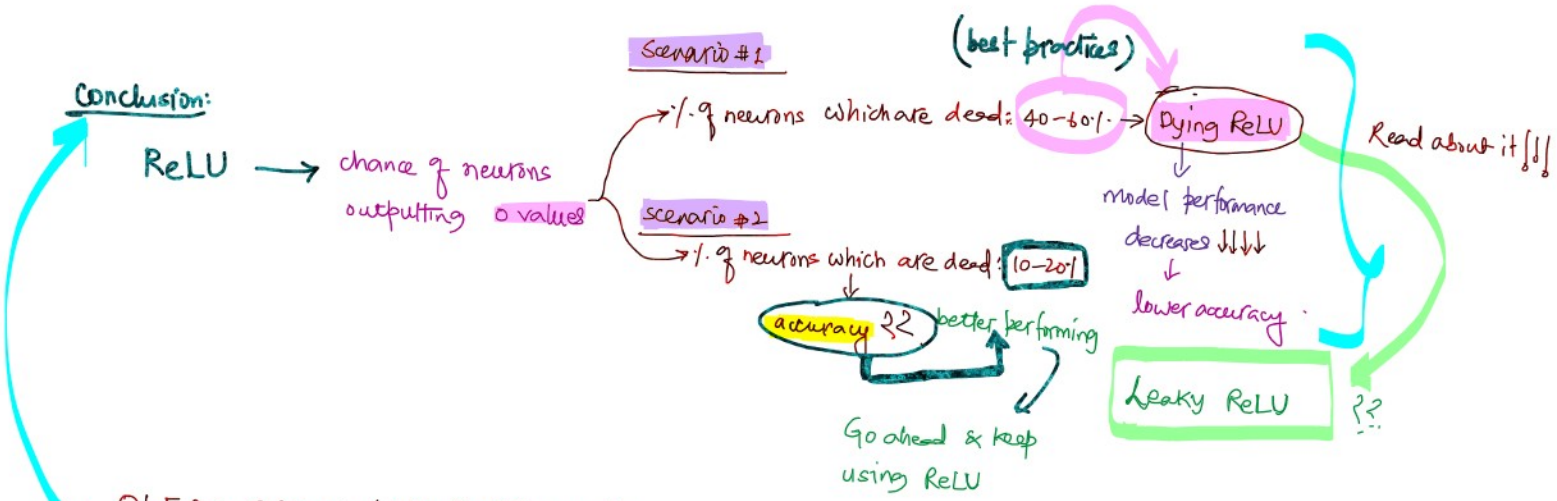
Sparse activations → ReLU makes some of the neurons dead or '0' → leads to efficient computations of the neural network model

Cons of using ReLU

Dying ReLU → it is a problem which refers to the scenario where a significant number of neurons in the neural n/w model always outputs zero after applying the ReLU AF.

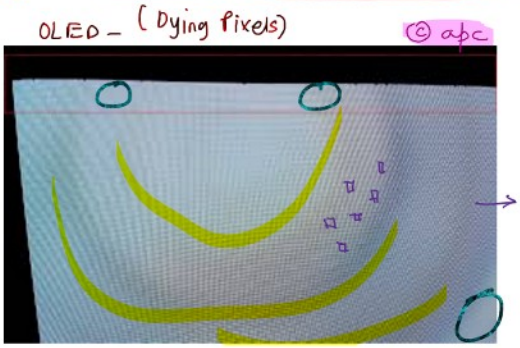
it is also non-zero centered just like sigmoid function.

Will do it later (module #3)
Vishwas: Although ReLU has drawbacks such as producing biased (non-zero-centered) outputs, these issues can be effectively taken care of by using good weight initialization (e.g., Kaiming) and Batch Normalization. Because of this ReLU is one of the most widely used activation functions for hidden layers in deep neural networks.



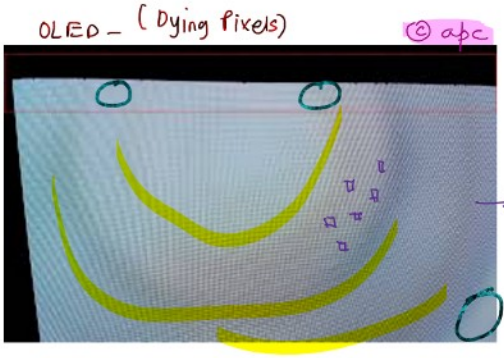
OLED: Organic Light Emitting Diode

OLED - (Dying Pixels)



→ GPUs are getting hotter

OLED – (Dying Pixels)



→ GPUs are getting hotter