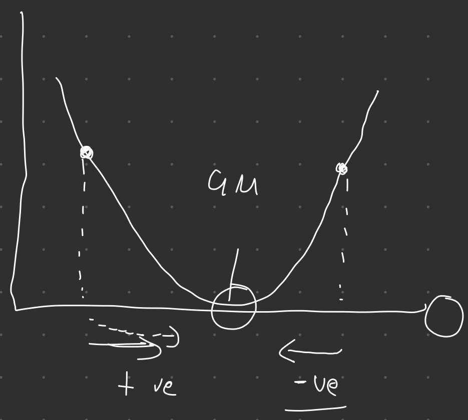
After this we use optimizers like Gradient Descent



1 epoch => 1 FP
1 BP

=> 62%

2 epoch => 69

3 " => 72

4 " => 84

5 " => 86

⋮

20th epoch => 92%

Activation function

1. Sigmoid AF => $\sigma = \frac{1}{1+e^{-(z)}}$

=> Vanishing Gradient Problem

What is the Vanishing Gradient Problem?

The vanishing gradient problem happens during backpropagation when gradients (the small updates we calculate to adjust weights) become so tiny that the earlier layers of a deep neural network stop learning.

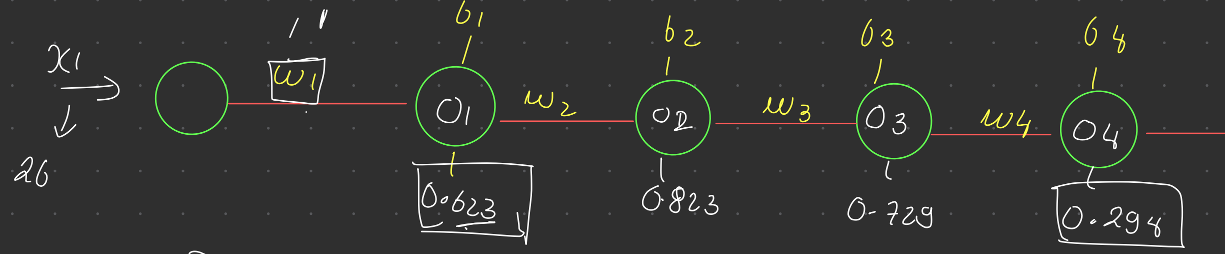
In other words:

When training deep networks, gradients are multiplied layer by layer.

If these gradients are very small (< 1), multiplying them across many layers makes them shrink toward zero.

This means the first few layers (closer to the input) never get updated properly, so the network fails to learn important low-level features.

Made with Goodnotes



$$\sigma\boxed{\left[(26 \times 1) + b\right]} \Rightarrow \sigma = \frac{1}{1 + e^{-((26 \times 1)+b)}}$$

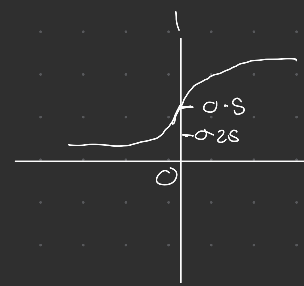
$$\Downarrow$$

0.623

100 x 2

100 x 0.5

= 50



$$w_{new} = w_{old} - \eta \boxed{\left(\frac{\partial Loss}{\partial w_{old}}\right)}$$

$$\frac{\partial Loss}{\partial w_1 old} = \frac{\partial loss}{\partial o_4} \times \frac{\partial o_4}{\partial o_3} \times \frac{\partial o_3}{\partial o_2} \times \frac{\partial o_2}{\partial o_1} \times \frac{\partial o_1}{\partial w_1}$$

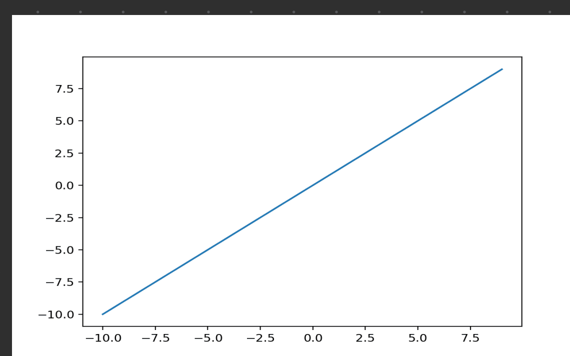
Type of Activation Functions

1. Linear Activation Function

A linear activation function is the simplest type of activation function.
It basically means:

f(x) = x

So, the output is the same as the input. The neuron doesn't transform the data it just passes it forward as it is.



Where do we use it?

In regression tasks (predicting continuous values like salary, house price, temperature, etc.).

Usually in the output layer of a neural network, because we don't want the output to be restricted between 0–1 (like sigmoid) or -1–1 (like tanh).

Example:
If you're predicting house price, you want outputs like ₹50,00,000 or ₹80,00,000.
A sigmoid function would squeeze everything between 0 and 1, which won't make sense here.

Limitations

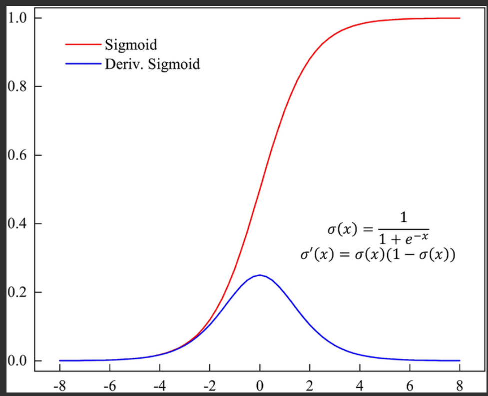
If you use linear activation in all layers, the whole network becomes just a linear model, no matter how many layers you add.

That means it cannot capture complex, non-linear patterns in data.
That's why hidden layers use non-linear activations (like ReLU, tanh, sigmoid), but the output layer for regression can be linear.

2. Sigmoid Activation Function

The sigmoid function is an S-shaped curve that squashes any real number into a range between 0 and 1.
The formula is:

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



So no matter how large or small the input, the output will always stay between 0 and 1.

Where do we use it?

In binary classification problems (e.g., predicting yes/no, disease/no disease, spam/not spam).

Usually in the output layer when you want a probability as the output.
Example:
If the sigmoid outputs 0.85, you can interpret it as 85% chance of having heart disease.

Limitations

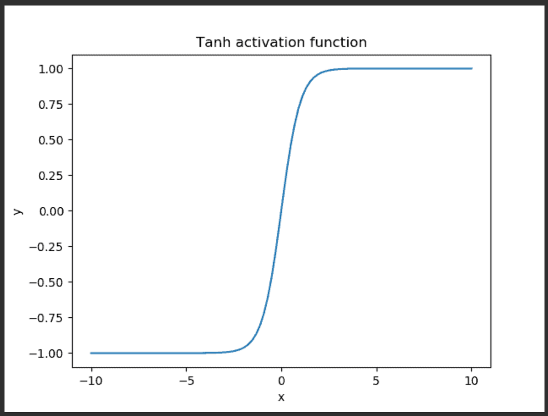
Vanishing gradient problem: for very large or very small inputs, the gradient becomes almost 0, which slows learning.

Not used in hidden layers much nowadays (ReLU is preferred).

3. Tanh Activation Function

The tanh function (short for hyperbolic tangent) is another squashing function like sigmoid, but instead of squeezing values into 0 to 1, it squeezes them into -1 to +1.
Formula :

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



Where do we use it?

- **Often used in hidden layers of nural network**

Useful when data has both positive and negative values because it centres the output around 0 (unlike sigmoid which is centred at 0.5)

Advantages

Outputs are zero-centered (good for optimization).

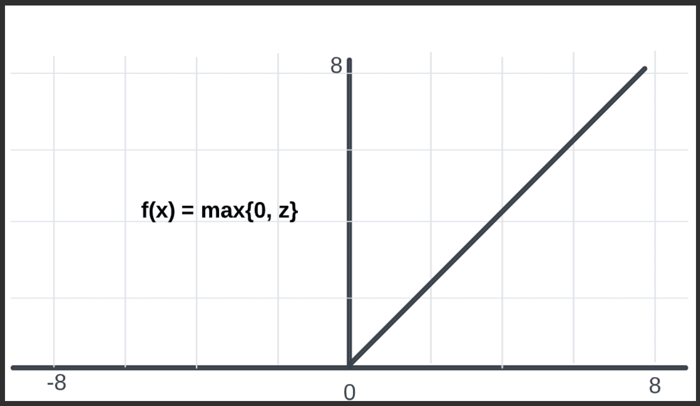
Stronger gradients than sigmoid in the range (−1,1), so learning can be faster.

Limitations

Still suffers from the vanishing gradient problem when inputs are very large (positive or negative).

That's why in modern deep learning, ReLU is more common in hidden layers.

4. ReLU Activation Function



ReLU stands for Rectified Linear Unit.

It's super simple:

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

That means:

IF input x < 0 -> output = 0

if input x > 0 -> output = x

So it either passes positive values as they are or blocks negative values by turning them into 0.

Where do we use it?

- Hidden layers of almost all modern deep neural networks.
- Works really well in CNNs (Convolutional Neural Networks), image recognition, NLP, and many more tasks.

Advantages

- Very fast and simple to compute.
- Helps avoid vanishing gradient problem (better than sigmoid/tanh).
- Makes training deep networks much faster.

Limitations

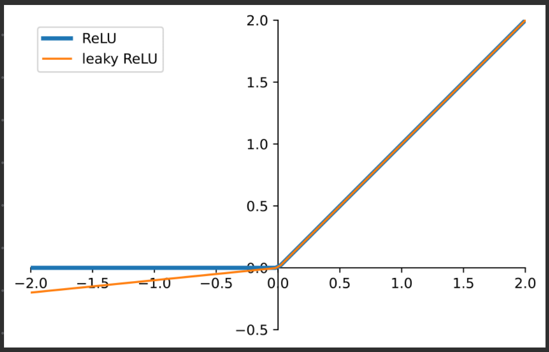
- Dying ReLU problem: sometimes neurons get stuck at 0 forever if weights update badly.
- Not smooth at 0 (not differentiable there, but still works fine in practice).

5. Leaky ReLU Activation Function

It's just like ReLU, but with a small twist.
In ReLU, whenever the input is negative, the output is 0.

Leaky ReLU's fix:

Instead of giving 0 for negative inputs, it gives a tiny negative value (like 0.01 × input).
This way, the neuron is never completely dead.



Formula:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{if } x <= 0 \end{cases}$$

Advantages of Leaky ReLU

Fixes "Dead Neuron" Problem

- **In normal ReLU, if inputs go negative, the output is always negative neuronsays 0, and sometimes the neuron stops learning permanently (dead neuron).**
- **Leaky ReLU solves this by allowing a small negative slope, so neurons still update weights.**

Computationally Simple

- **Just like ReLU, the function is very easy to compute (no heavy math like exponentials in Sigmoid/Tanh).**

Better Gradient Flow

- **Since even negative inputs have a small gradient (e.g., 0.01), the network can continue learning, reducing the vanishing gradient issue.**

Works Well in Deep Networks

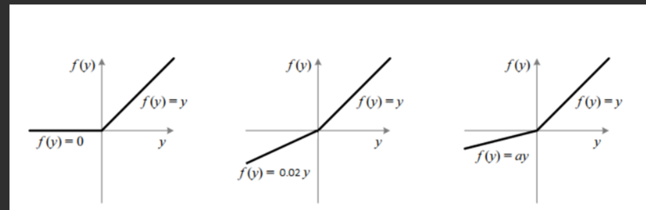
- **Especially useful in deep neural networks where ReLU may suffer from many dead neurons.**

Limitations: small negative slope may bias results, slope value needs tuning

6. PReLU Activation Function

PReLU (Parametric Rectified Linear Unit) is an improved version of Leaky ReLU. In Leaky ReLU, the slope for negative values (like 0.01) is fixed by us. But in PReLU, that slope is learned automatically by the model during training. This makes it more flexible and adaptive.

The formula is same as Leaky relu



(Left) ReLU, (Middle) LeakyReLU and (Last) PReLU

Intuition (Easy Way)

ReLU: Negative values are killed (output = 0).

Leaky ReLU: Negative values are given a tiny leak (e.g., 0.01x).

PReLU: Instead of fixing that leak, the model says "I'll learn the best leak slope myself."

Advantages

1. **Fixes dead neurons (like Leaky ReLU).**
2. **Adaptive – slope is learned, not fixed.**
3. **Better accuracy – often improves CNNs and deep networks.**

Limitations

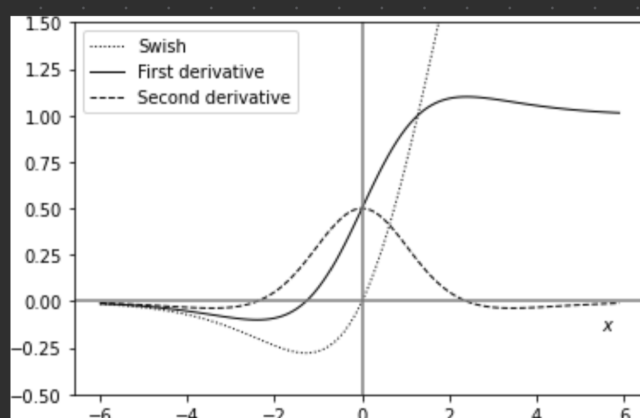
1. **Extra parameters – slope a adds more trainable values.**
2. **Risk of overfitting if dataset is small.**
3. **Slightly more complex than plain ReLU.**

7. Swish Activation Function

Swish is a smooth, non-linear activation function introduced by Google researchers.

It is Defined as :

$$f(x) = x \cdot \sigma(x)$$



Where σ(x) is the sigmoid function
so basically: Swish = X * Sigmoid(x)

Intuition (Easy Way)

Think of it as ReLU but smoother.

For large positive inputs → output ≈ input (like ReLU).

For large negative inputs → output is small but not strictly zero (like Leaky ReLU).

Around zero → the curve is smooth, not sharp like ReLU.

This smoothness often makes training deep networks easier.

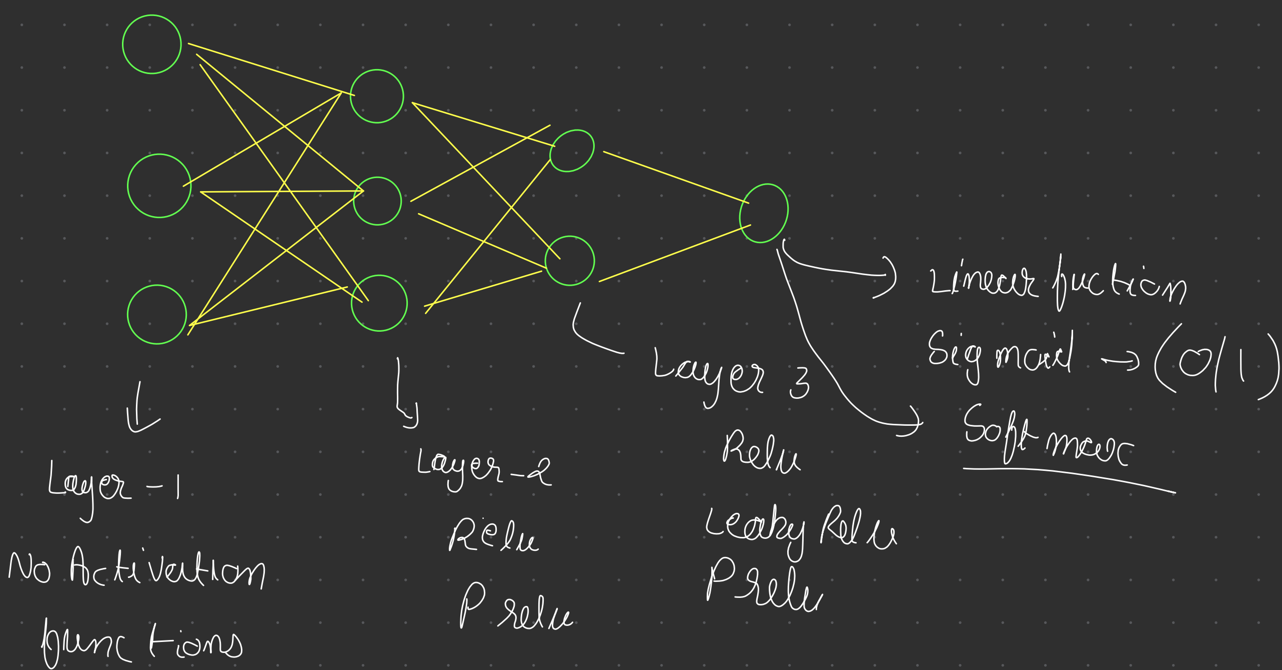
Advantages

1. **Smooth curve → better gradient flow, avoids sharp jumps like ReLU.**
2. **Non-monotonic → can adapt better to complex patterns.**
3. **Works well in deep networks (often improves accuracy over ReLU).**

Limitations

1. **More computation (needs sigmoid).**
2. **Not always better than ReLU (depends on problem).**
3. **Slight risk of slower training compared to simple ReLU.**

Usage of Activation functions



---

1. loss

This is the training loss.

It tells you how wrong your model's predictions are on the training set, according to the loss function you chose (in your case: binary cross-entropy).

Lower is better.

During training, the model tries to reduce this number by updating weights with gradient descent.

2. accuracy

This is the training accuracy.

It tells you what fraction of training samples the model is correctly predicting.

Example: If you have 100 samples and the model gets 80 correct → accuracy = 0.80 (80%).