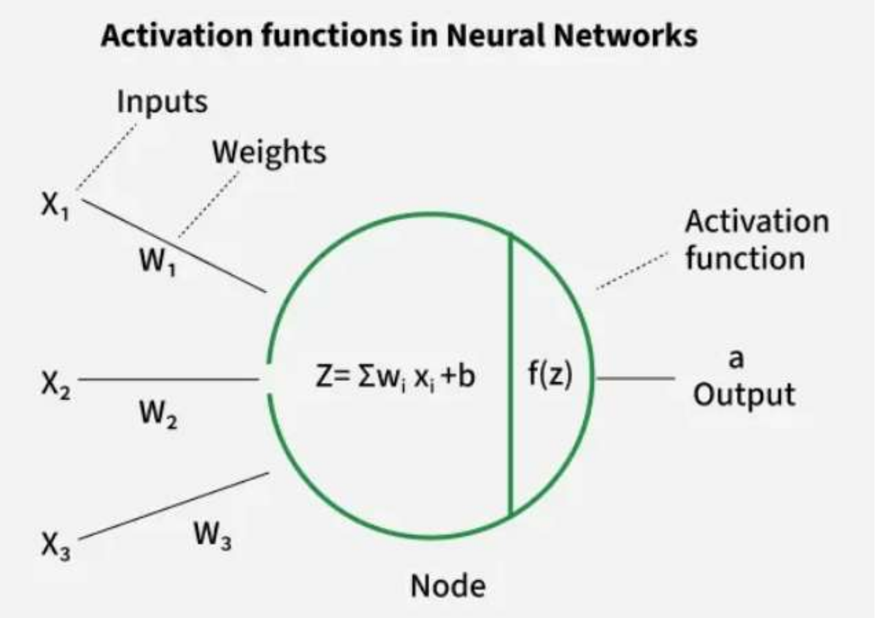
Types of Activation Functions

CT-466 | Week 1 - Lecture 2

Instructor: Mehar Fatima Shaikh

Activation function

- An activation function decides whether a neuron should be "activated" or not by applying a transformation to the weighted sum of inputs.
- It introduces non-linearity into the network, allowing deep learning models to learn complex patterns instead of just simple linear relationships.



Why do we need Activation Functions?

- Without them, a neural network would just be a linear model, no matter how many layers it has.
- With activation functions, neural networks can model complex mappings (e.g., speech recognition, image classification).
- They help with gradient flow during backpropagation.

Types of Activation Functions

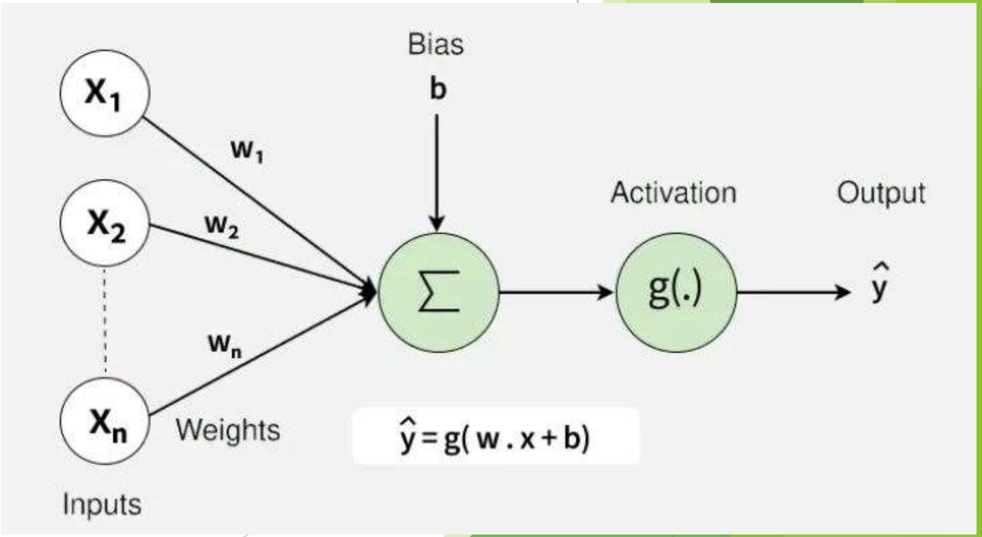
- Sigmoid Activation Function
- Tanh Activation Functions
- ReLU and Leaky ReLU Activation Functions

Sigmoid activation function

- The **sigmoid activation function** is a mathematical function often used in neural networks.
- It looks like this:

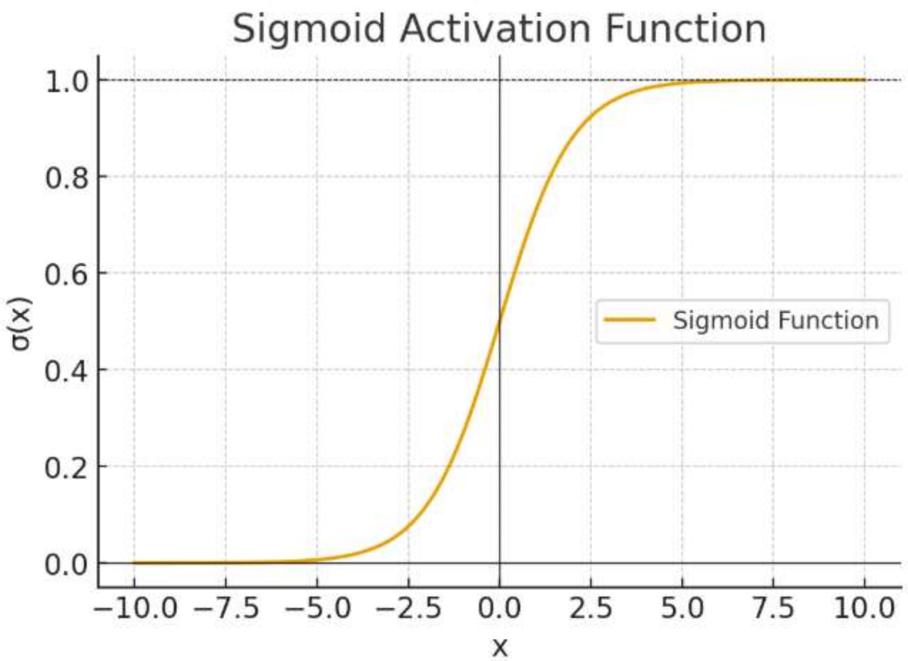
$$f(y) = \sigma(y) = \frac{1}{1 + e^{-y}}$$

- Applies on output layer.
- Range: Output is always between 0 and 1.
- Used for binary output
- Small inputs (negative x) → output is close to 0.
- Large inputs (positive x) → output is close to 1.
- At x = 0 → output is 0.5.
- Since output is always Positive it creates bias.



Why it’s used:

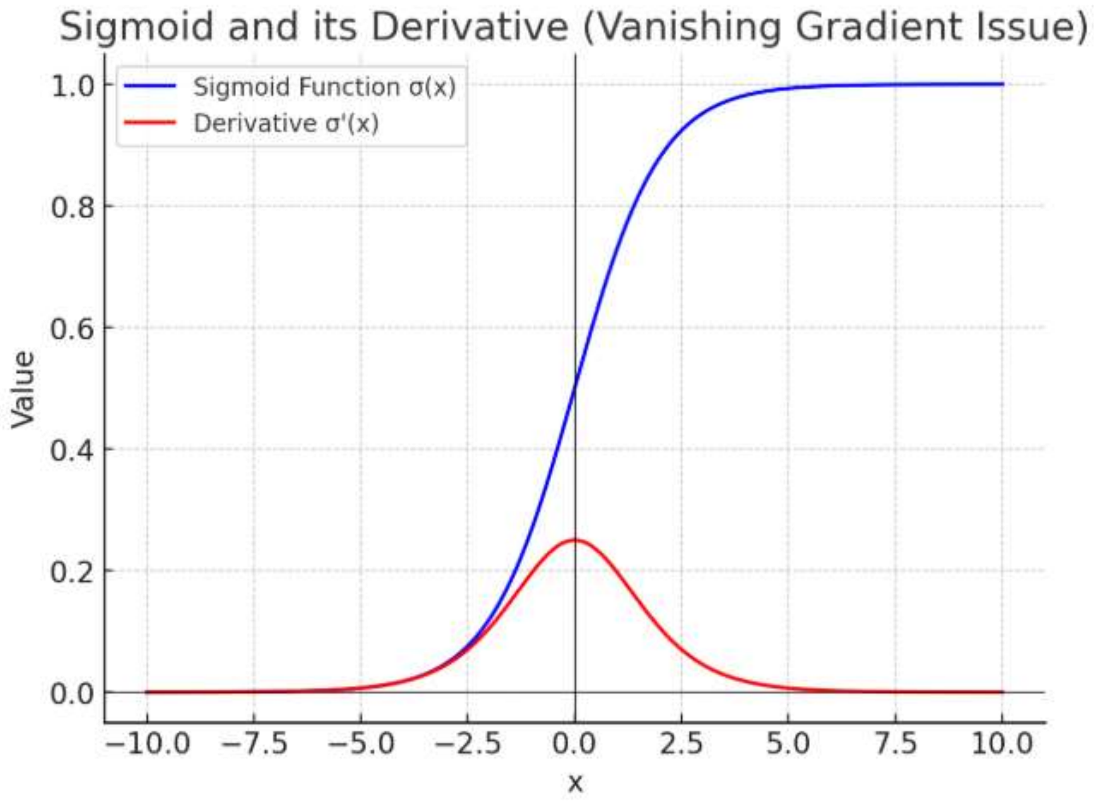
- Converts any input into a probability-like value between 0 and 1.
- Useful for binary classification problems.
- Limitation: It can cause vanishing gradients in deep networks, so nowadays ReLU and other functions are often preferred.



Sigmoid Derivation

- f(y) = σ(y)= $\frac{1}{1+e^{-y}}$
- $\sigma'(x) = \frac{d\,f(y)}{dy} = \frac{d}{dy}\left(\frac{1}{1+e^{-y}}\right) = (1+e^{-y})^{-1}$
- $\sigma'(x) = (-1)\,(1+e^{-y})^{-2}(-e^{-y})$
- $\sigma'(x) = \frac{e^{-y}}{(1+e^{-y})^2} \quad = \frac{e^{-y}}{1+e^{-y}} \; * \frac{1}{1+e^{-y}}$
- $\sigma'(x) = \frac{1+e^{-y}-1}{1+e^{-y}} \; * \frac{1}{1+e^{-y}} = (\frac{1+e^{-y}}{1+e^{-y}} - \frac{1}{1+e^{-y}}) * \frac{1}{1+e^{-y}}$
- $\sigma'(x) = (1 - \sigma(y)) * \sigma(y)$

Sigmoid derivative with the vanishing gradient problem.



Sigmoid derivative with the vanishing gradient problem.

$$\sigma'(x)=\sigma(x)\cdot(1-\sigma(x))$$

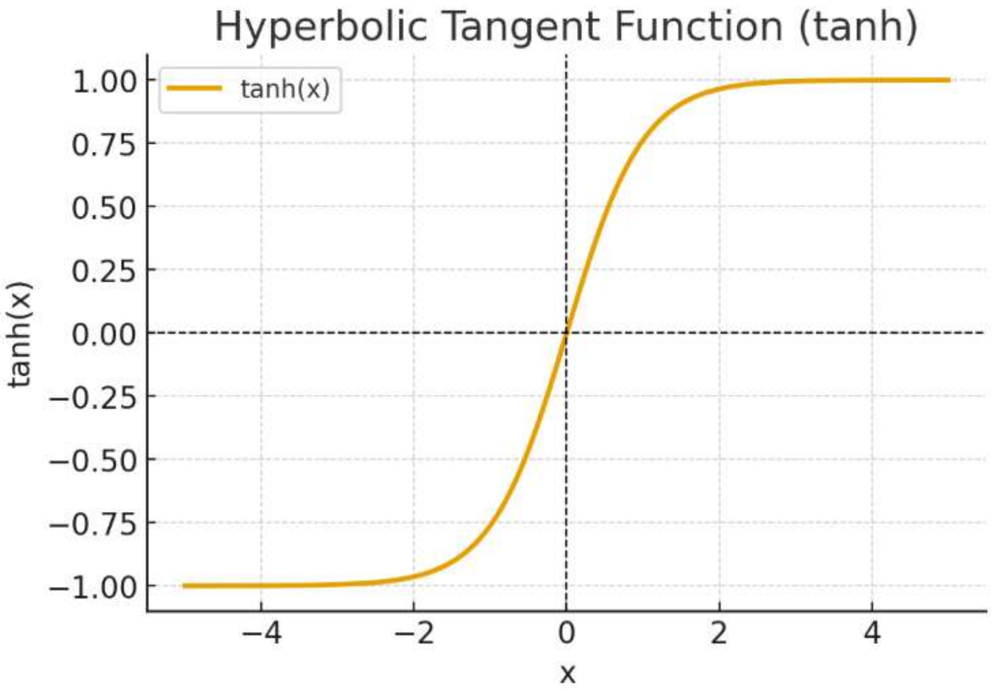
- Maximum value of derivative is 0.25 (when $x=0$, since $\sigma(0)=0.5$).
- For large positive or negative x, the derivative becomes very close to 0.
- In deep neural networks, backpropagation multiplies many derivatives together. If each derivative is small (like 0.1 or 0.01), then after multiplying across many layers:

$$0.1^{10} = 1e - 10(almost\ 0!)$$

- So, the gradients shrink towards 0, and earlier layers stop learning → this is the vanishing gradient problem.

Tanh (hyperbolic tangent function)

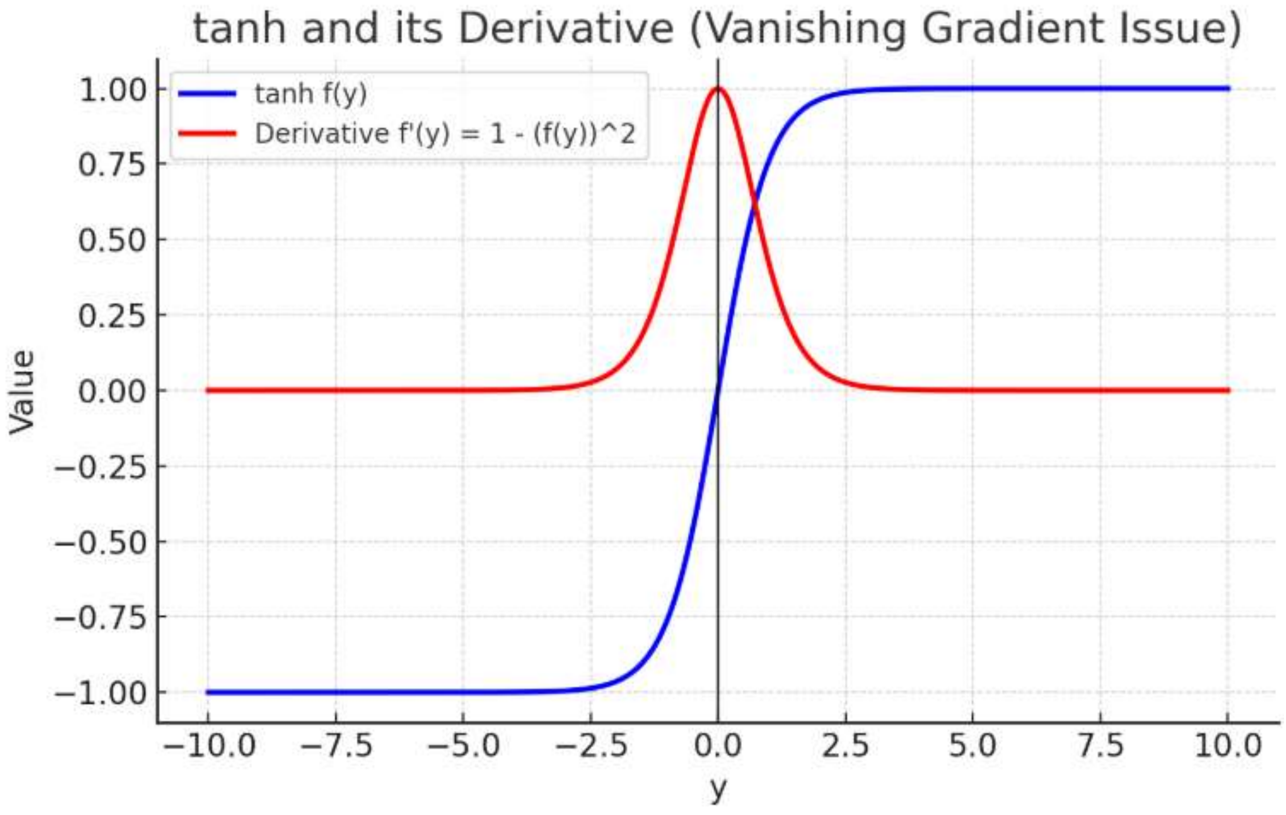
- In machine learning and neural networks, tanh is commonly used as an activation function because it outputs values in the range [−1,1], making it useful for handling both positive and negative inputs.
- It is a mathematical function often written as:
- $f(y) = \tanh(y) = \frac{sinh(y)}{\cosh(y)} = \frac{e^{y} - e^{-y}}{e^{y} + e^{-y}}$



Derivative of Tanh

- Let $f(y) = \tanh(y) = \frac{(e^{y} - e^{-y})}{(e^{y} + e^{-y})}$
- Define u(y) = $(e^{y} - e^{-y})$ and v(y) = $(e^{y} + e^{-y})$
- Then f(y) = $\frac{\text{u(y)}}{\text{v(y)}}$
- U\`(y) = $(e^{y} - (-1)e^{-y})$ = $(e^{y} + e^{-y})$ = v(y)
- V\`(y) = $(e^{y} + e^{-y})$ = $(e^{y} + (-1)e^{-y})$ = $(e^{y} - e^{-y})$ = u(y)
- $f`(y) = \frac{(\mathrm{u}`(y)\mathrm{v}(y) - \mathrm{u}(y)\mathrm{v}`(y))}{v(y)^2} = \frac{(\mathrm{v}(y)*\mathrm{v}(y) - \mathrm{u}(y)*\mathrm{u}(y))}{v(y)^2} = \frac{(v(y))^2 - (u(y))^2}{v(y)^2}$
- $f`(y) = \frac{(v(y))^2 - (u(y))^2}{v(y)^2} = \frac{(e^{y} + e^{-y})^2 - (e^{y} - e^{-y})^2}{(e^{y} + e^{-y})^2}$
- $f`(y) = \frac{(e^{y} + e^{-y})^2}{(e^{y} + e^{-y})^2} - \frac{(e^{y} - e^{-y})^2}{(e^{y} + e^{-y})^2}$
- $f`(y) = 1 - (\frac{e^{y} - e^{-y}}{e^{y} + e^{-y}})^2$
- $f`(y) = 1 - (f(y))^2$

Tanh derivative with the vanishing gradient problem.

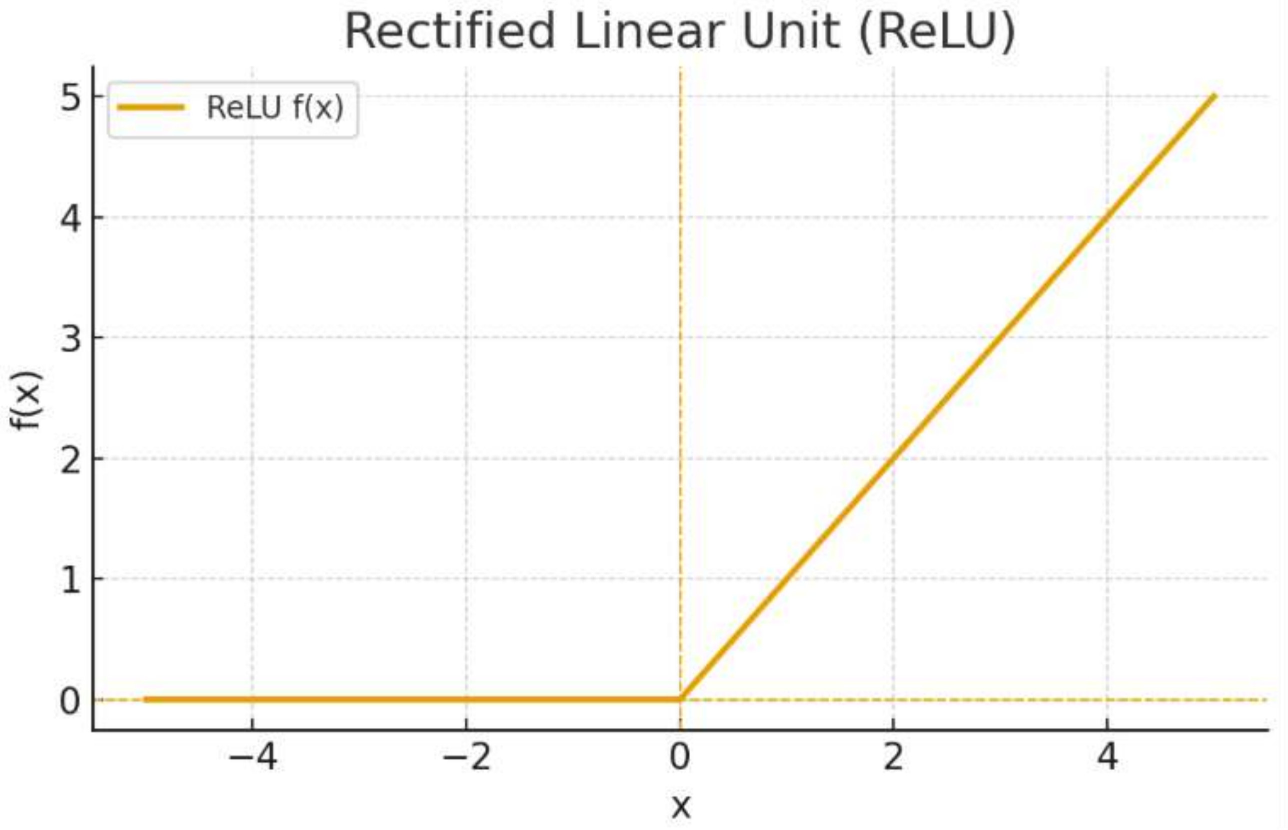


Vanishing gradient problem.

- The tanh function takes any real input and squashes it smoothly between –1 and +1.
- Its derivative (slope) is largest when the input is near 0. At that point, the slope can be close to 1, so gradients flow well during backpropagation.
- But as the input moves far away (very positive or very negative), the tanh curve flattens. The slope (derivative) then becomes almost 0.
- When training a deep neural network, gradients are multiplied layer after layer during backpropagation. If each derivative is a small number (close to 0), the product becomes tiny. This is called the vanishing gradient problem.
  - It means that early layers stop learning because their weight updates shrink almost to zero.
  - In practice, this slows down or completely blocks training of deep networks.
- tanh helps center data between –1 and +1, but still suffers from vanishing gradients when inputs are far from 0.

ReLU (Rectified Linear Unit)

- The **ReLU (Rectified Linear Unit) activation function** is one of the most widely used functions in deep learning.
- It looks like this:
- f(y)=max(0,y)
- $f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$

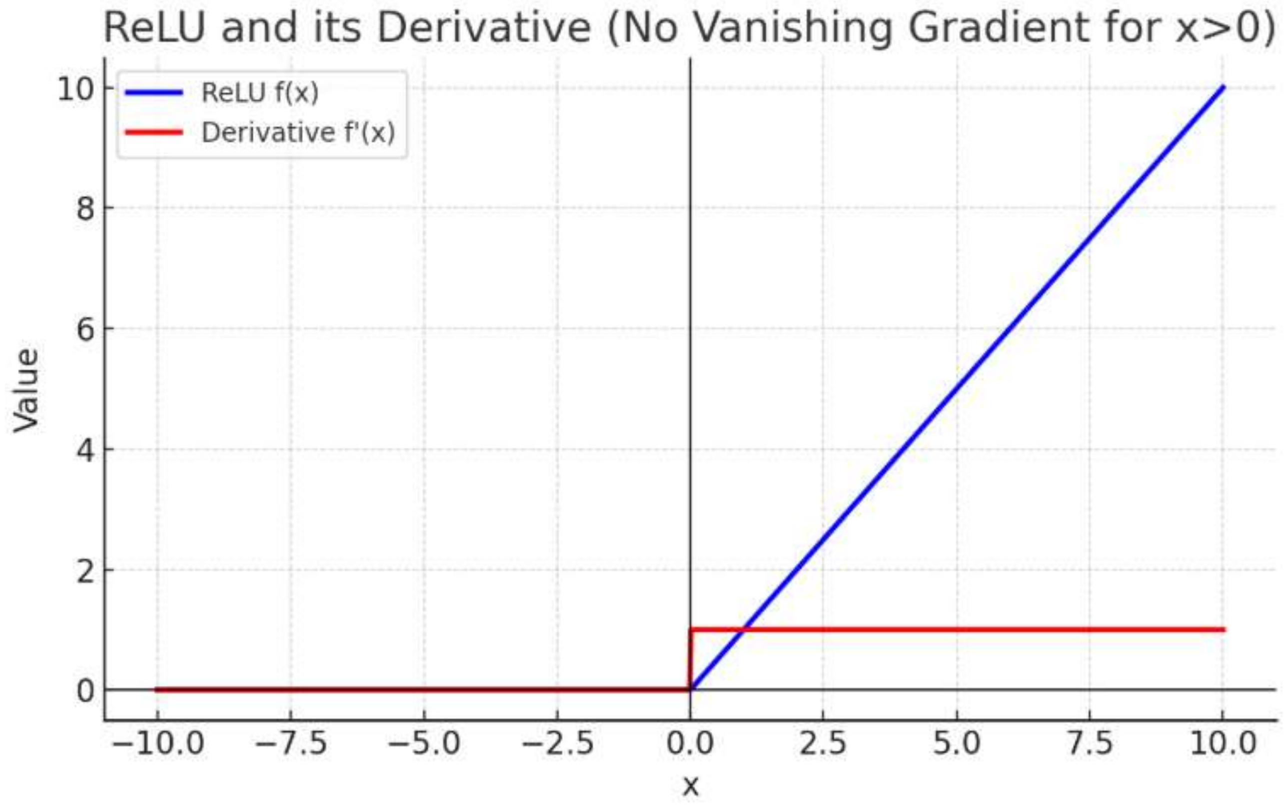


ReLU (Rectified Linear Unit)

- Applies on hidden layers (commonly used in CNNs, ANNs, RNNs).
- Range: Output is between 0 and ∞
- Used for introducing non-linearity while keeping computation simple.
- Small/negative inputs ($y<0$) → output is 0.
- Large/positive inputs ($y>0$) → output is equal to the input itself.
- At $y=0$ → output is 0.
- Does not saturate for positive inputs, so it helps avoid vanishing gradients.
- Can cause the “dying ReLU” problem, where many neurons output 0 and stop learning if inputs are negative.

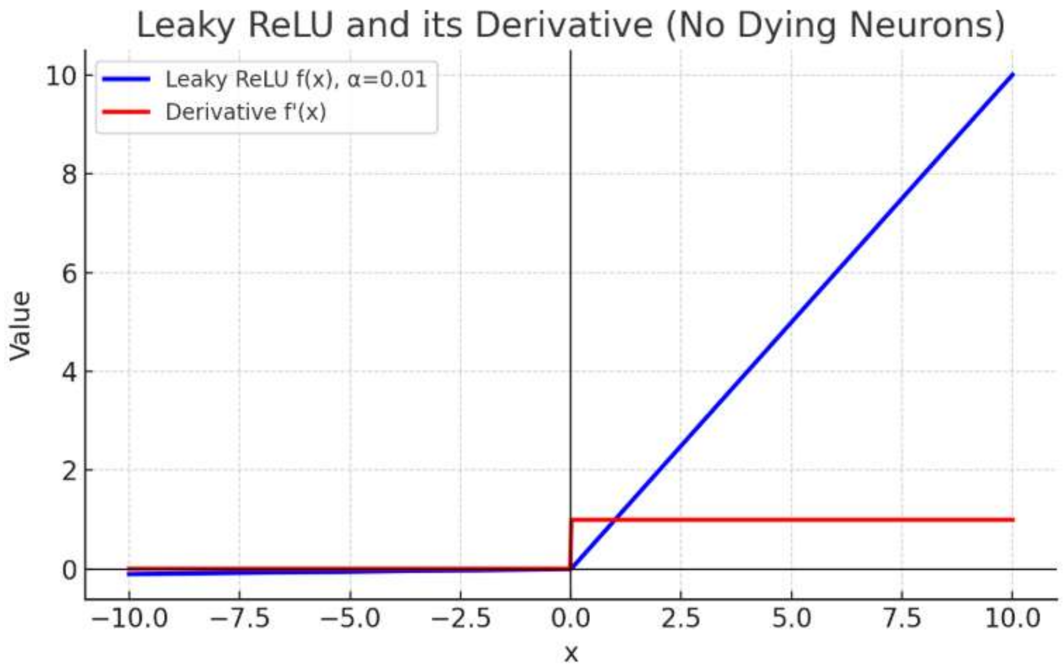
ReLU derivative with the vanishing gradient problem.

- $f`(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$



Leaky ReLU

- $f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$
- Where α is a small constant (e.g., 0.01)
- For x>0: behaves like ReLU (linear, slope = 1).
- For x≤0: instead of being flat, the function has a small negative slope (α=0.01), so the derivative is also small but nonzero.



Python code

```
# Inputs and weights
x_inputs = [0.1, 0.5, 0.2]
W_weights = [0.4, 0.3, 0.4]
threshold = 0.5

# Step function
def step(weighted_sum):
    if weighted_sum > threshold:
        return 1
    else:
        return 0

# Perceptron function
def perceptron(x_inputs, W_weights):
    weighted_sum = 0
    for x, w in zip(x_inputs, W_weights):
        weighted_sum += x * w
    print("Weighted sum:", weighted_sum)
    return step(weighted_sum)

# Run perceptron
output = perceptron(x_inputs, W_weights)
print("Output:", output)
```