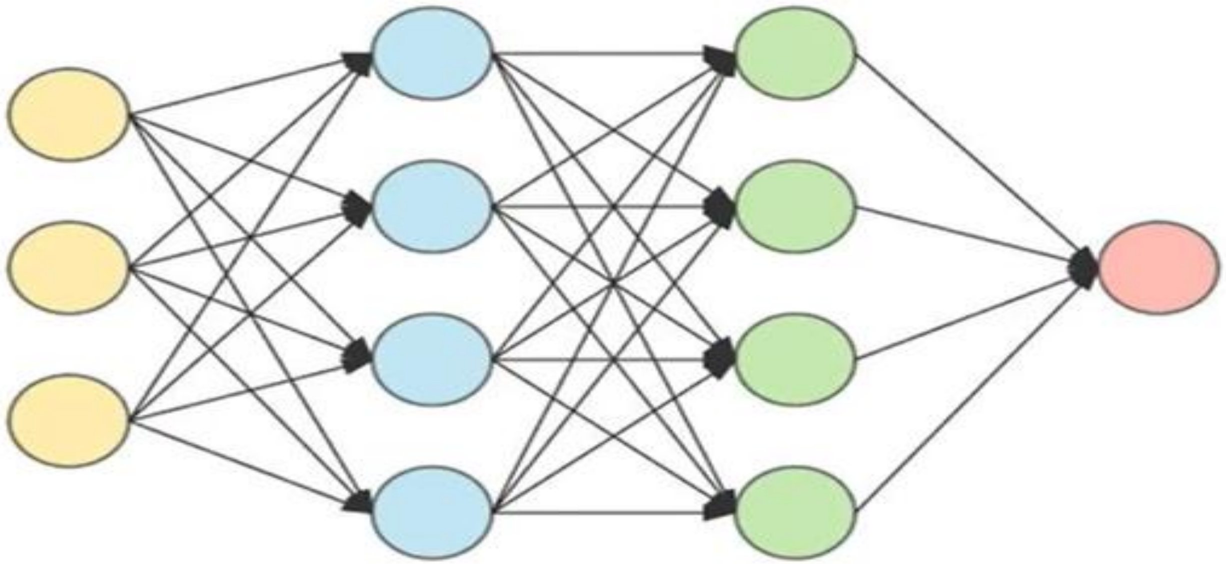
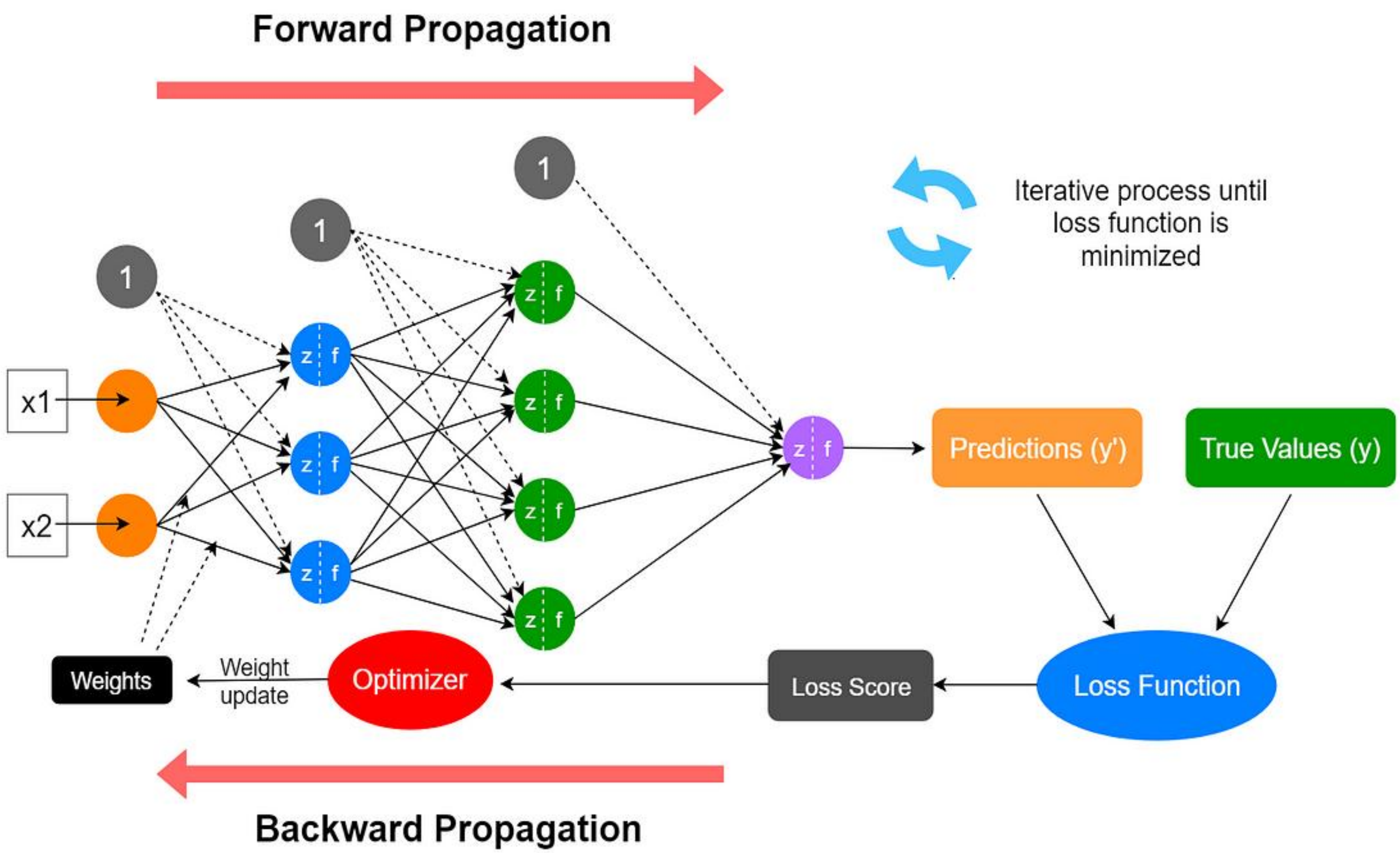
KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanaguda, Hyderabad.

# NATURAL LANGUAGE PROCESSING (21CM601PC)

BY
ASHA
ASSISTANT PROFESSOR
CSE(AI&ML)
KMIT

# Activation Functions in Neural Networks




Forward Propagation
1
1
1
x1
x2
z f
z f
z f
z f
z f
z f
z f
z f
Iterative process until
loss function is
minimized
Predictions (y')
True Values (y)
Weights
Weight
update
Optimizer
Loss Score
Loss Function
Backward Propagation

**Forward Propagation**

- The process of computing the output of a neural network. It involves:

1. Multiplying inputs by weights.
2. Adding biases.
3. Applying the activation function to produce the output.

# Why do we need Activation functions?

Activation functions are crucial in neural networks because they introduce **non-linearity** into the model, enabling the network to learn and model complex patterns in the data. Without activation functions, a neural network would essentially behave like a linear regression model, limiting its ability to capture the true underlying structure in most real-world problems.

$$Z1 = W1 * A0 + B1$$
$$\cancel{A1 = f(Z1)}$$

$$Z2 = W2 * A1 + B2$$
$$\cancel{A2 = f(Z2)}$$

$$Z3 = W3 * A2 + B3$$
$$\cancel{A3 = f(Z3)}$$

$$A2 = W2 * (W1 * A0 + B1) + B2$$

$$A2 = \underbrace{W2 * W1}_{} * A0 + \underbrace{W2 * B1 + B2}_{}$$

$$A2 = W' * A0 + B'$$

$$A3 = W'' * A0 + B''$$

## 1. Real-Life Analogy: The Light Dimmer

•Without activation → neurons behave like a simple *on/off switch* (just linear scaling).

•With activation → neurons behave like a **dimmer** switch → brightness adjusts differently at different levels.

👉 This shows that activation functions give neurons *nonlinear control*, not just a fixed slope.

## 1. How real neurons work

- A neuron in the brain receives inputs from thousands of other neurons.
- Each input has a *weight* (strength of connection).
- The neuron **adds up all these inputs**.
- If the total **crosses a threshold**, the neuron **fires** (sends a signal forward).
- Otherwise, it stays silent.

👉 So, the brain is **not linear** → it's full of **nonlinear thresholding and modulation**.

## 2. Artificial Neurons mimic this

In Artificial Neural Networks (ANNs), each neuron also **sums up inputs**:

$$z = w1.x1 + w2.x2 + ... + b$$

But if we just pass z forward as it is → it's linear.

So we need an **activation function** to decide:

- Should the neuron fire strongly?
- Should it suppress weak signals?
- Should it pass values partially (not just ON/OFF)?

“Activation functions in ANNs exist because, just like biological neurons, we don’t want every signal to pass through linearly. We want the neuron to **decide how much to fire** – weakly, strongly, or not at all.”

## Non-Linearity

- Why needed? → To learn complex, nonlinear patterns in data

# Step Function (Threshold Function)

- Formula:

$$f(x) = 1 \text{ if } x > 0, \text{ else } 0.$$

  Intuition: Neuron fires (1) or stays silent (0).

$$f(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{if } z < \theta \end{cases}$$

- z=wx+b is the neuron’s pre-activation (weighted sum + bias).

- **Use:**
  - Early **perceptrons** (1950s–60s).
  - Rarely used today → not differentiable, so **not good for backpropagation**.
- **When?** Only for very simple **binary decision rules**.

# ReLU (Rectified Linear Unit)

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

Formula:

Derivative:

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

- **Intuition**: Pass positive signals, block negatives (like a switch).

**Use:**

**Most popular activation** for hidden layers.

Simple, efficient, avoids vanishing gradient (mostly).

**Limitations:**

“Dying ReLU” problem → neurons stuck at 0.

**When?**

Default choice in **hidden layers of deep networks**.

**Dying ReLU problem** happens when a neuron **always outputs 0** for all inputs.

This means the neuron is effectively **dead**: it never activates, never contributes to learning.

The gradient through it is also **0** (since slope for $x<0$ is zero).

Once dead, it **never recovers**, because weight updates depend on gradients.

**How to fix dying ReLU?**

Leaky ReLU:

# Leaky ReLU / Parametric ReLU

**Formula:**

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

Derivative:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \le 0 \end{cases}$$

**Intuition**: Fixes dead neurons problem.

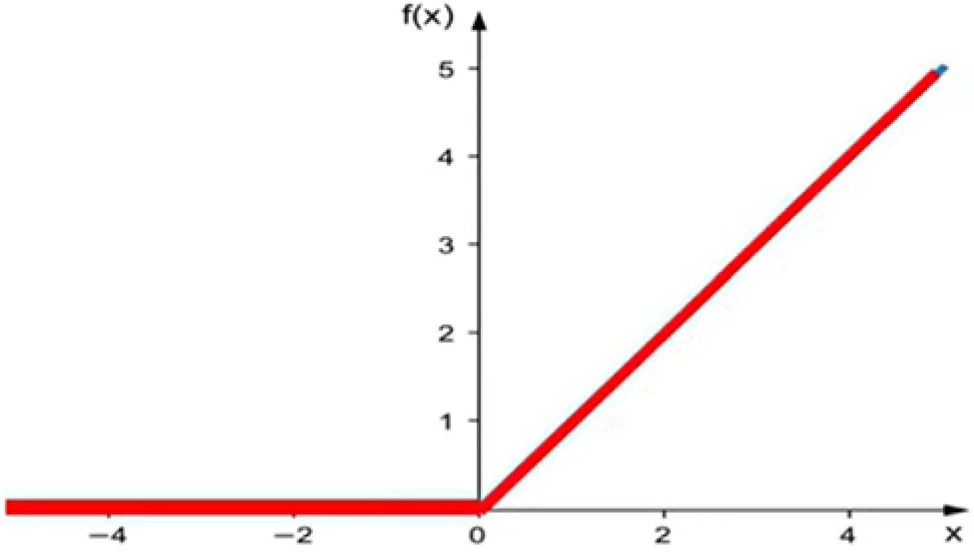
**Use:**

Fixes “dying ReLU” by allowing a small slope for $x<0$.

**When?**

Same places as ReLU, but safer if you notice many dead neurons.

# ReLU (Rectified Linear Unit)




Piece-wise Linear

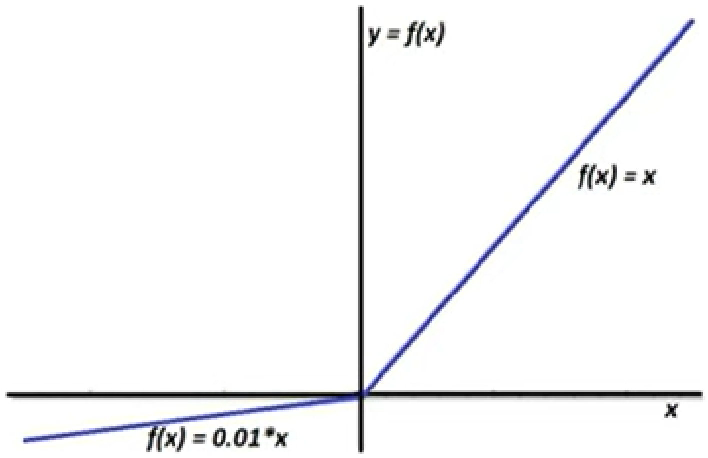
*Advantages of both linear and non-linear property*

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

Overcome the Vanishing Gradient Problem

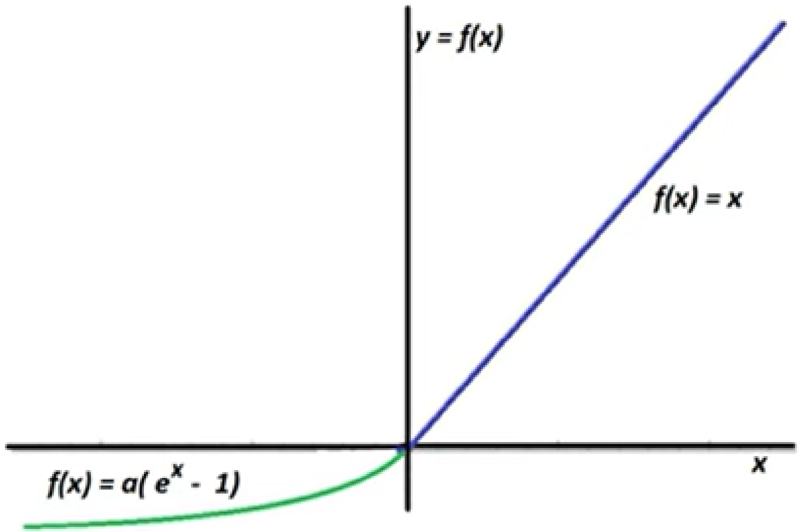
$$\frac{\partial f(x)}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$$

# Variations of ReLU




Leaky ReLU

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




ELU (Exponential Linear Unit)

$$f(x) = max(\ \alpha*(e^{x} - 1,\ x),\ x)$$

# Tanh Function

**Formula:**

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

Derivative:

$$f'(x) = 1 - \tanh^2(x)$$

Range: (-1,1)

**Intuition**: Similar to sigmoid, but centered at zero (better for optimization).

**Problem**: Still suffers from vanishing gradients.

**Use:**

Outputs in **[-1,1]** → zero-centered, often better than sigmoid.

Historically popular in RNNs.

**Limitations:**

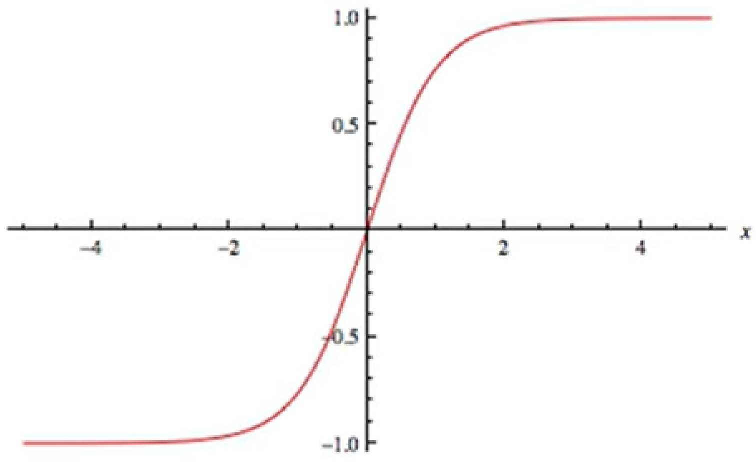
Still suffers from vanishing gradients.

**When?**

Good choice in hidden layers **if you need negative outputs**.

Sometimes in old RNN architectures (before LSTMs/GRUs).

# Tanh Function




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

# Sigmoid Function

**Formula:**

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

Derivative:

$$f'(x) = f(x)(1 - f(x))$$

Range: (0,1)

**Intuition**: Smooth "S" curve → squashes values to probability-like output.

**Use Case**: Logistic regression, binary classification.

**Problems**: Vanishing gradients for very large/small x, slow convergence.

**Use:**

Outputs between **0 and 1** → good for **probabilities**.

Still used in **binary classification (output layer)**.

**Limitations:**

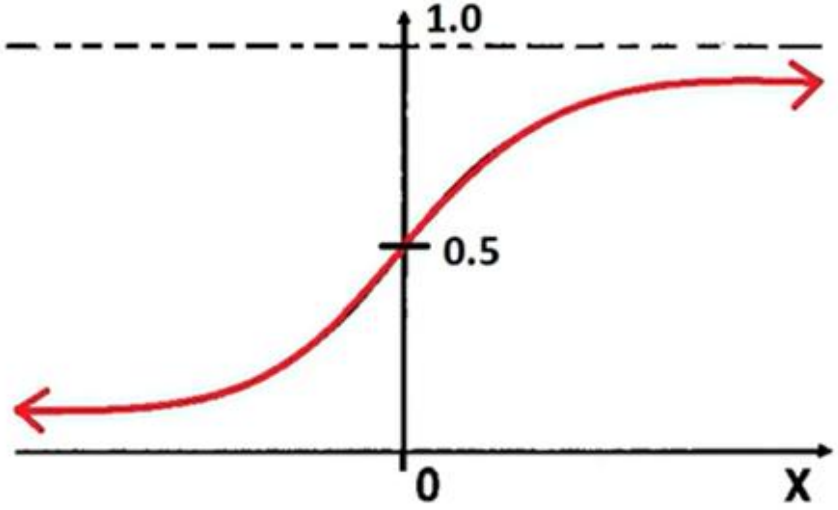
Causes **vanishing gradients** for large $|x|$.

**When?**

Final output layer of binary classification models.

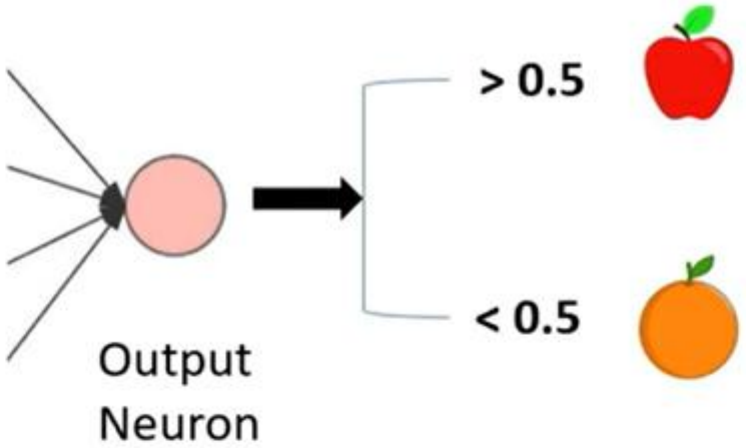
Logistic regression.

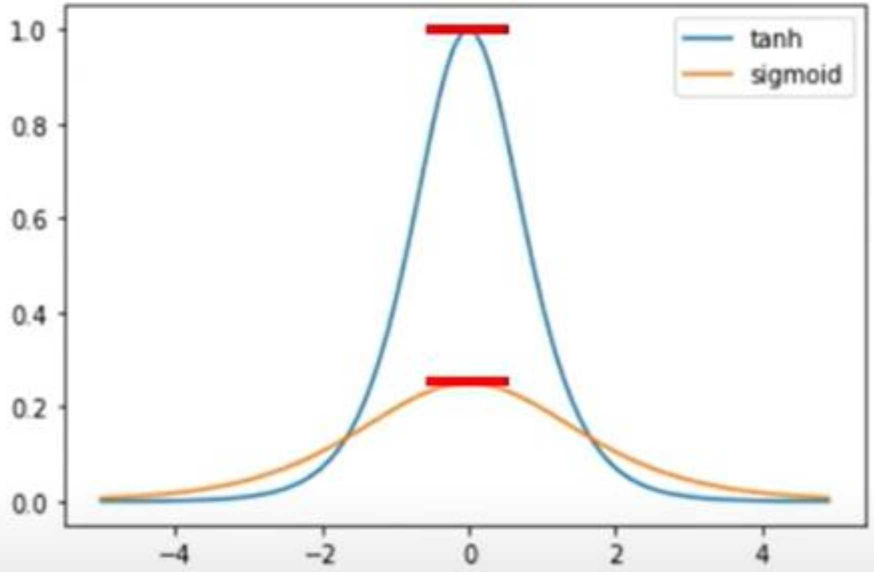
# Sigmoid Function



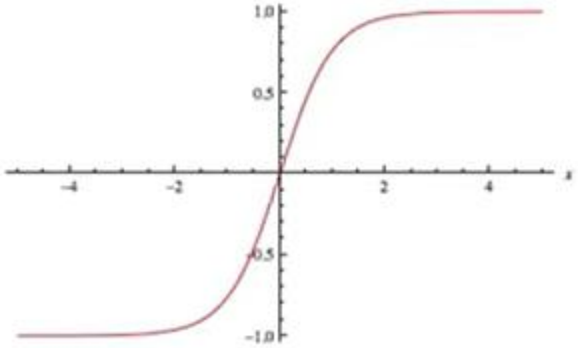

*Sigmoid*

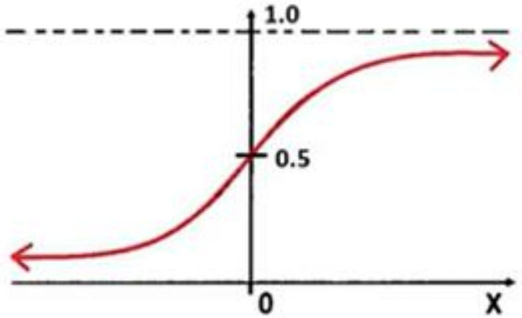
$$f(x) = \frac{1}{1+e^{-x}}$$

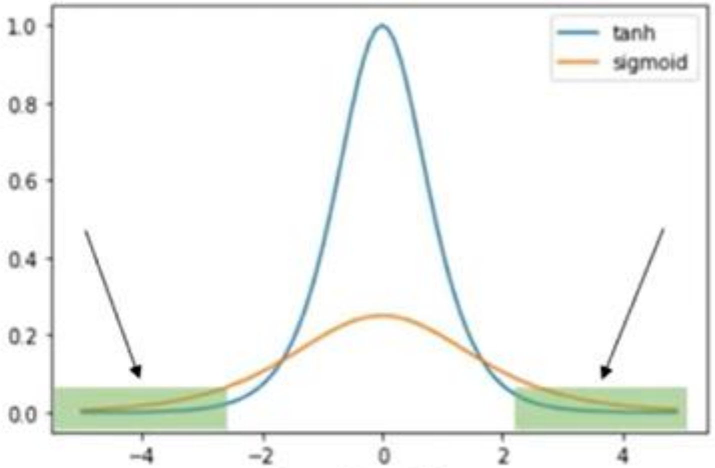





**Derivatives**



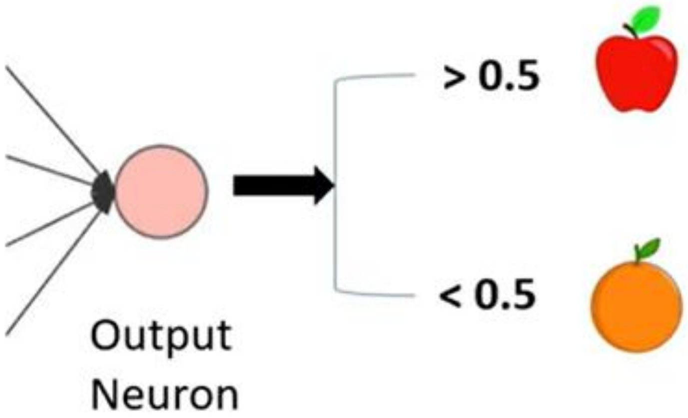



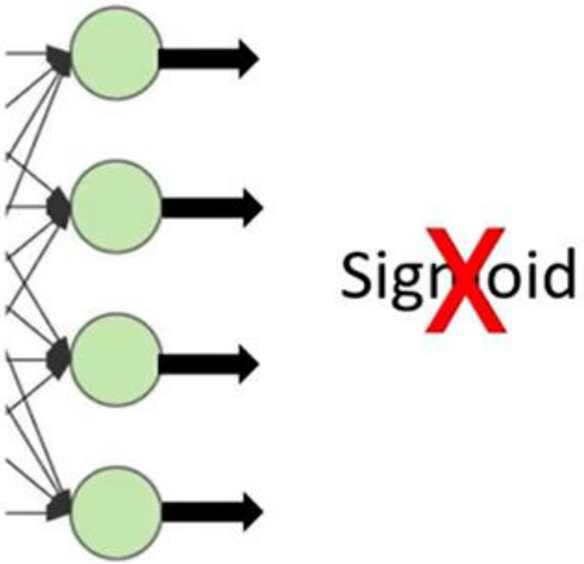

**Derivatives**

Smaller value of derivative leads to very slow learning

Vanishing Gradient Problem




Binary Classification




Multi-Class Classification
??

# Softmax (for output layers in classification)

## Formula:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Derivative (for backprop):

$$\frac{\partial f(x_i)}{\partial x_j} = f(x_i)(\delta_{ij} - f(x_j))$$

**Intuition**: Converts raw scores into probabilities across classes.

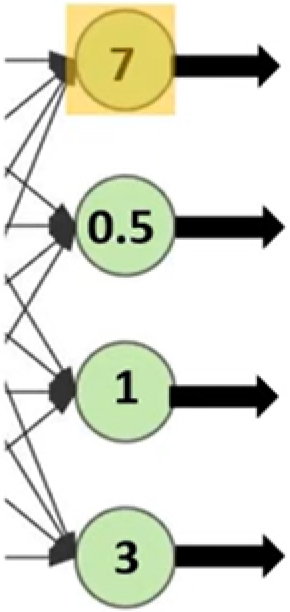
**Use:**

Converts a vector of logits into a **probability distribution**.

Used only in **output layer of multi-class classification**.

**When?**

Final layer for multi-class problems (e.g., image classification with 10 labels).




$$\begin{bmatrix}7\\0.5\\1\\3\end{bmatrix} \rightarrow \begin{bmatrix}e^{7}\\e^{0.5}\\e^{1}\\e^{3}\end{bmatrix} \rightarrow \begin{bmatrix}\dfrac{e^{7}}{e^{7}+e^{0.5}+e^{1}+e^{3}}\\[2ex]\dfrac{e^{0.5}}{e^{7}+e^{0.5}+e^{1}+e^{3}}\\[2ex]\dfrac{e^{1}}{e^{7}+e^{0.5}+e^{1}+e^{3}}\\[2ex]\dfrac{e^{3}}{e^{7}+e^{0.5}+e^{1}+e^{3}}\end{bmatrix} \rightarrow \begin{bmatrix}0.9782\\0.00147\\0.00242\\0.0179\end{bmatrix}$$

$$\begin{bmatrix}7\\0.5\\1\\3\end{bmatrix} \rightarrow \begin{bmatrix}\dfrac{7}{7+0.5+1+3}\\[2ex] \dfrac{0.5}{7+0.5+1+3}\\[2ex] \dfrac{1}{7+0.5+1+3}\\[2ex] \dfrac{3}{7+0.5+1+3}\end{bmatrix} \rightarrow \begin{bmatrix}0.6086\\0.0435\\0.0869\\0.2608\end{bmatrix} \rightarrow \begin{bmatrix}0.9782\\0.00147\\0.00242\\0.0179\end{bmatrix}$$

## Why *mostly* these few activation functions?

- **History & Experiments:**

  Many activations (e.g., Gaussian, sine, polynomial) were tried in early neural networks. But they caused:

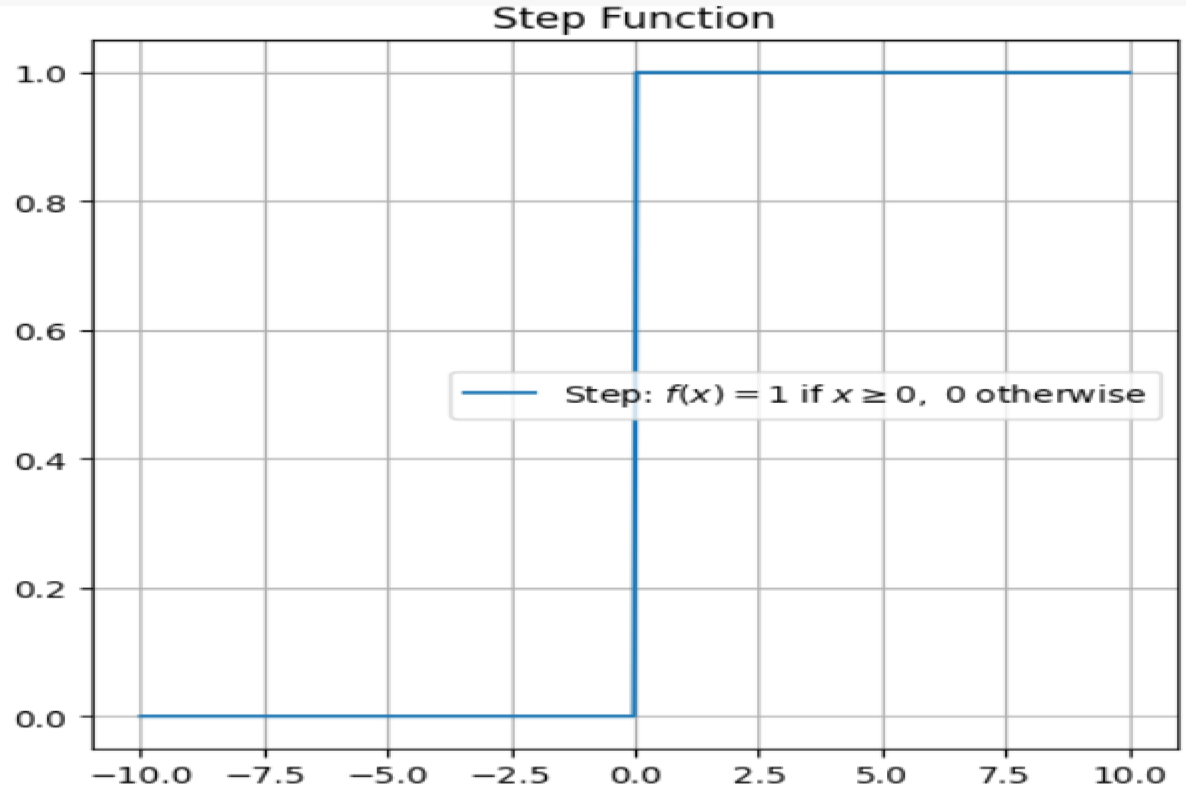
  - Very slow training.
  - Vanishing or exploding gradients.
  - Difficulty in generalizing.

- Through trial-and-error, researchers found **Sigmoid, Tanh, ReLU, and their variants** worked best in practice.

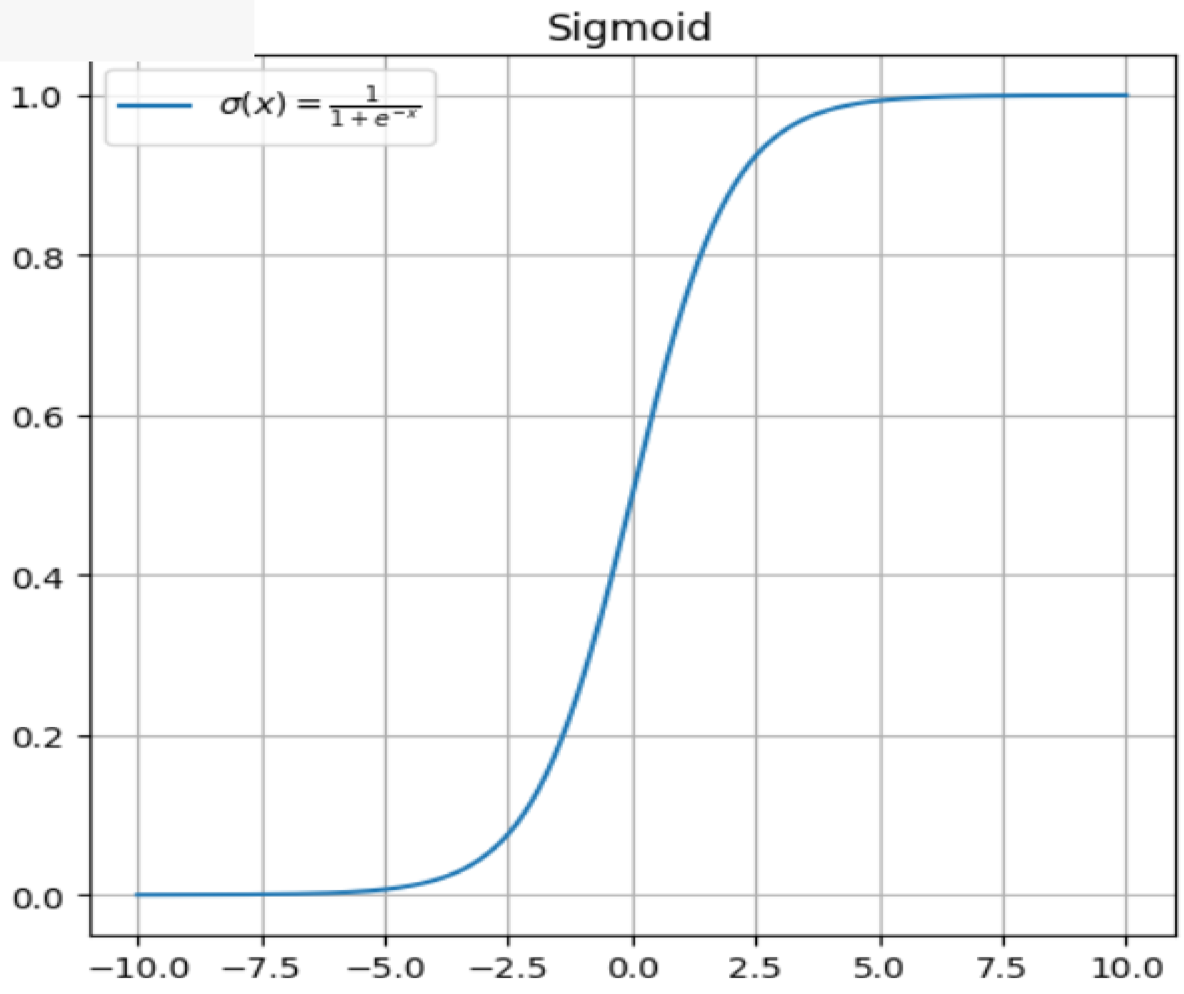
- **Practical Needs:**
  - We need functions that are **non-linear**, otherwise the whole network is just a linear regression.
  - They must be **differentiable** (or piecewise differentiable) so gradient descent can work.
  - Their **derivatives** should not vanish or explode for most input ranges.

```python
# Activation Functions Demo with Formulas & Explanations
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

# Range of x values
x = np.linspace(-10, 10, 400)

# Functions
step = np.where(x >= 0, 1, 0)                         # Step Function
sigmoid = 1 / (1 + np.exp(-x))                        # Sigmoid
tanh = np.tanh(x)                                     # Tanh
relu = np.maximum(0, x)                               # ReLU
leaky_relu = np.where(x > 0, x, 0.1*x)                # Leaky ReLU

# Softmax needs 2D input → shape (1, n)
softmax = tf.nn.softmax(x.reshape(1, -1)).numpy().flatten()
```

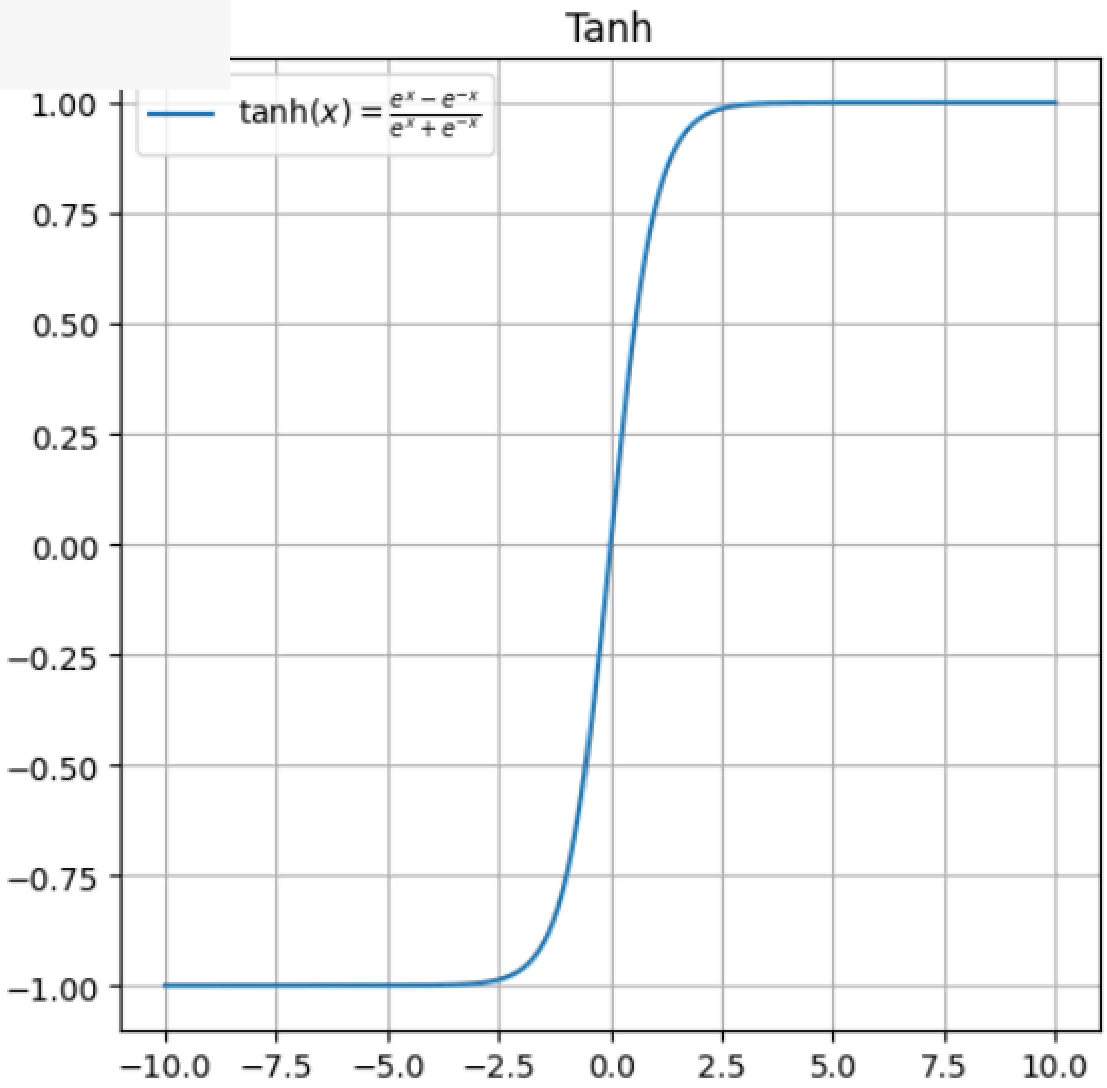
```python
# Plotting
plt.figure(figsize=(15,10))

# Step
plt.subplot(2,3,1)
plt.plot(x, step, label=r'Step: $f(x)=1 \; \text{if } x \geq 0, \; 0 \; \text{otherwise}$')
plt.title("Step Function")
plt.grid(True); plt.legend()
```



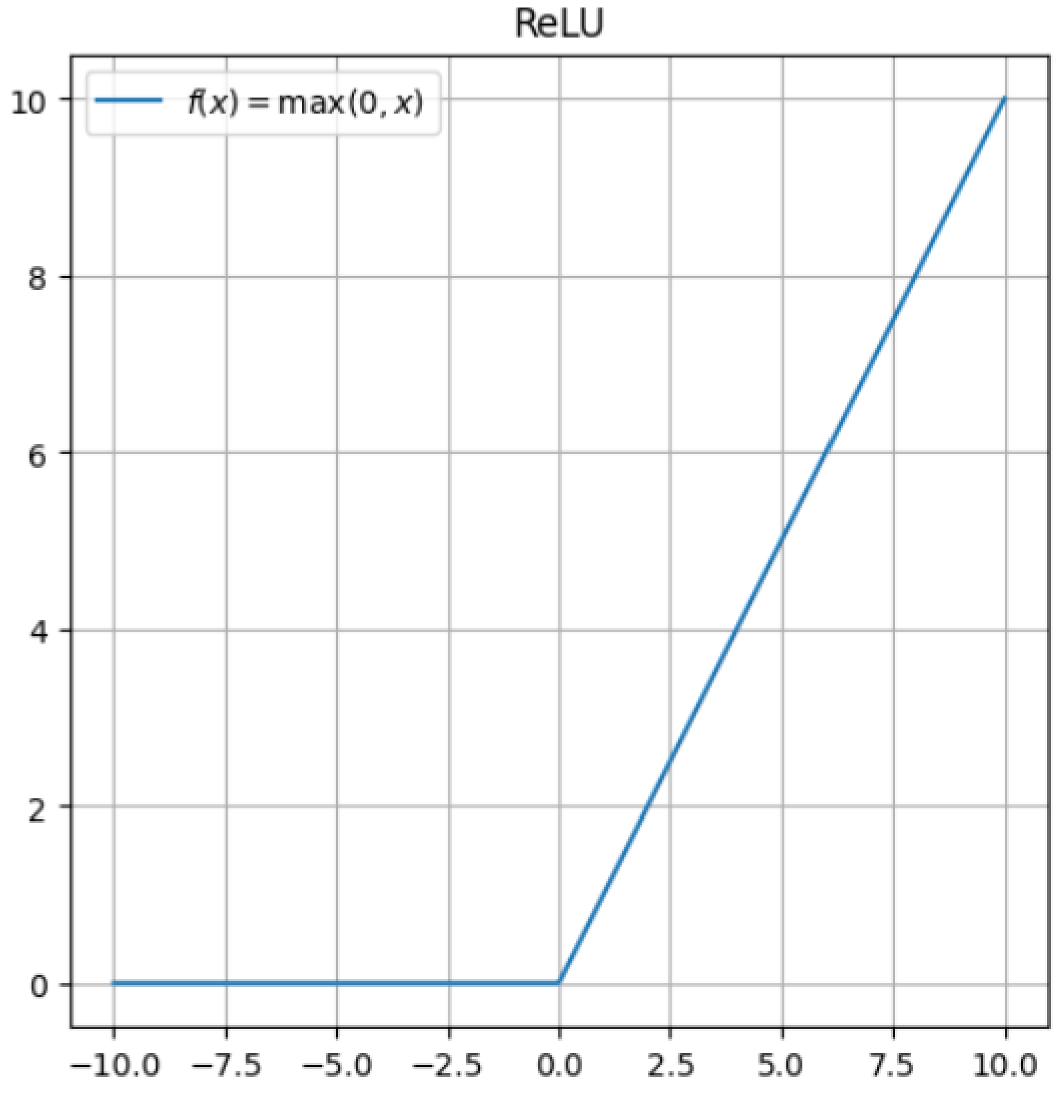
```python
# Sigmoid
plt.subplot(2,3,2)
plt.plot(x, sigmoid, label=r'$\sigma(x)=\frac{1}{1+e^{-x}}$')
plt.title("Sigmoid")
plt.grid(True); plt.legend()
```



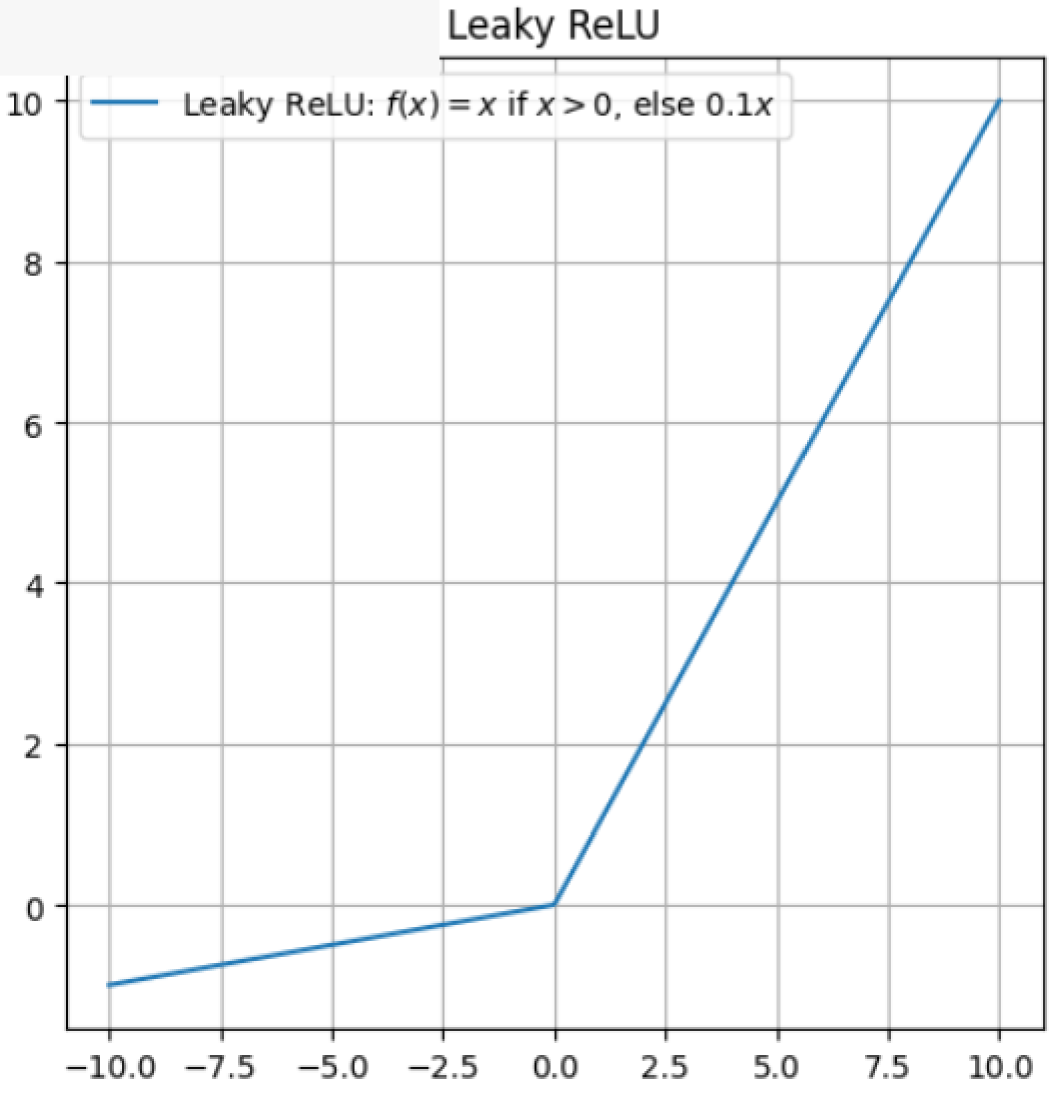
```python
# Tanh
plt.subplot(2,3,3)
plt.plot(x, tanh, label=r'$\tanh(x)=\frac{e^x - e^{-x}}{e^x+e^{-x}}$')
plt.title("Tanh")
plt.grid(True); plt.legend()
```



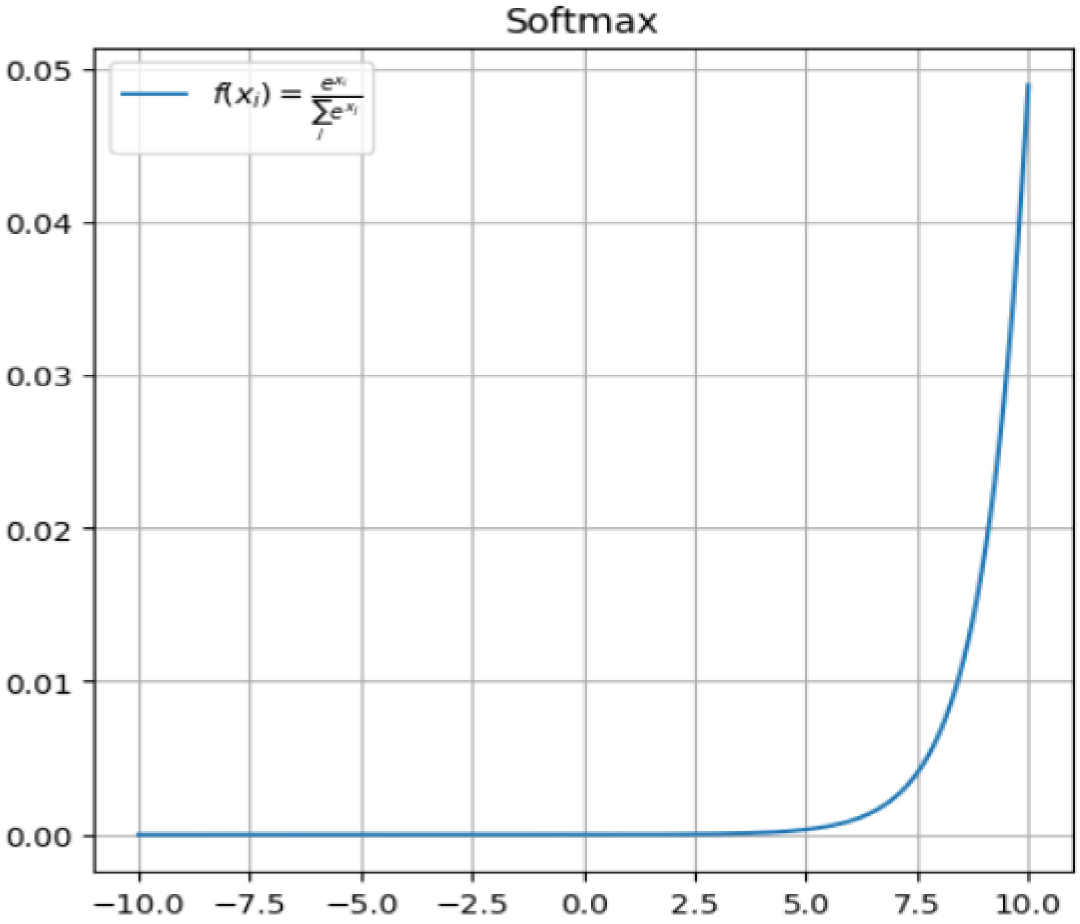
```python
# ReLU
plt.subplot(2,3,4)
plt.plot(x, relu, label=r'$f(x)=\max(0,x)$')
plt.title("ReLU")
plt.grid(True); plt.legend()
```



```python
# Leaky ReLU
plt.subplot(2,3,5)
plt.plot(x, leaky_relu, label=r'Leaky ReLU: $f(x)=x$ if $x>0$, else $0.1x$')
plt.title("Leaky ReLU")
plt.grid(True); plt.legend()
```



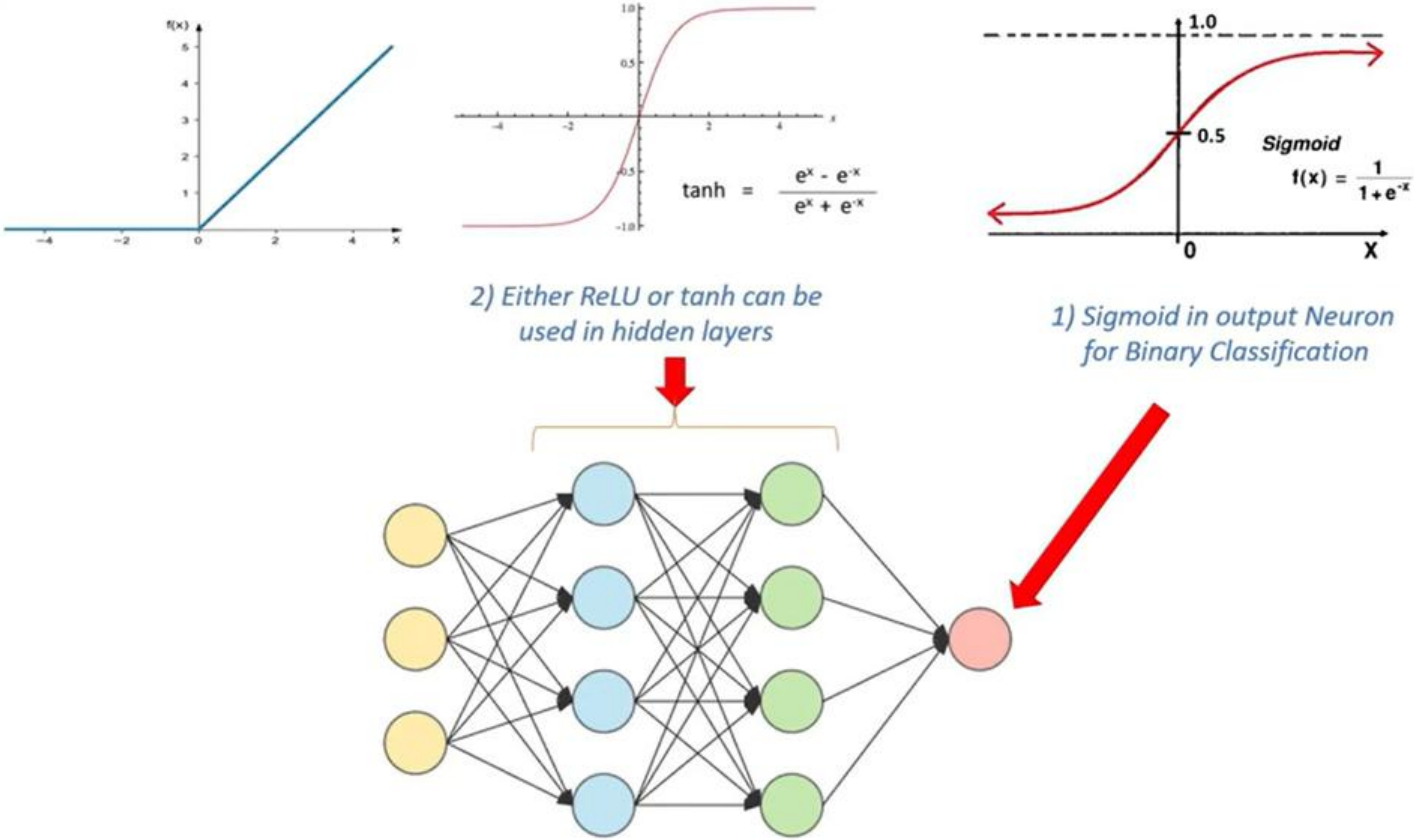
```python
# Softmax
plt.subplot(2,3,6)
plt.plot(x, softmax, label=r'$f(x_i)=\frac{e^{x_i}}{\sum_j e^{x_j}}$')
plt.title("Softmax")
plt.grid(True); plt.legend()

plt.tight_layout()
plt.show()
```



1. Step Function: Outputs 0 or 1. Used in early perceptrons but not differentiable.
2. Sigmoid: Smoothly maps input to (0,1). Good for probabilities but causes vanishing gradients.
3. Tanh: Maps input to (-1,1). Zero-centered but still suffers vanishing gradients.
4. ReLU: Outputs positive values as is, else 0. Very popular, avoids vanishing gradients (mostly).
5. Leaky ReLU: Like ReLU but allows small negative slope. Solves 'dying ReLU' problem.
6. Softmax: Converts vector into probability distribution. Common in output layers for classification.

## Rule of Thumb:

- **Hidden layers** → ReLU (or Leaky ReLU if dead neurons problem).
- **Output layer** →
  - **Sigmoid** for binary classification.
  - **Softmax** for multi-class classification.
  - **No activation (linear)** for regression.


f(x)
x
tanh = $\frac{e^x - e^{-x}}{e^x + e^{-x}}$
1.0
0.5
Sigmoid
f(x) = $\frac{1}{1+e^{-x}}$
0
X
2) Either ReLU or tanh can be used in hidden layers
1) Sigmoid in output Neuron for Binary Classification

| Activation | Formula | Range | Pros | Cons | When to Use |
|---|---|---|---|---|---|
| **Step** | $f(x) = 1$ if $x \geq 0$, else 0 | {0,1} | Simple, mimics biological neurons | Not differentiable → can't train with backprop | Rarely; only toy models or binary hard thresholds |
| **Sigmoid** | $\sigma(x) = \frac{1}{1+e^{-x}}$ | (0,1) | Smooth, interpretable as probability | Vanishing gradients, not zero-centered | Output layer of **binary classification** |
| **Tanh** | $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ | (-1,1) | Zero-centered, stronger gradients than sigmoid | Still vanishing gradient | Sometimes in **hidden layers**; classic RNNs |
| **ReLU** | $f(x) = \max(0, x)$ | [0, ∞) | Simple, efficient, avoids vanishing gradients | "Dying ReLU" (neurons stuck at 0) | **Default for hidden layers** in deep nets |
| **Leaky ReLU** | $f(x) = x$ if $x > 0$, else $0.01x$ | (-∞, ∞) | Fixes dying ReLU, allows small gradient when x<0 | Small slope may affect learning | Hidden layers if ReLU has many dead neurons |
| **Softmax** | $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$ | (0,1), sums to 1 | Produces probability distribution | Sensitive to large input values | **Output layer for multi-class classification** |
| **Linear (Identity)** | $f(x) = x$ | (-∞, ∞) | Keeps values unchanged | No non-linearity | **Regression tasks (output layer)** |

**Role of Derivatives**

The **derivative of the activation** is what flows backward during backpropagation:

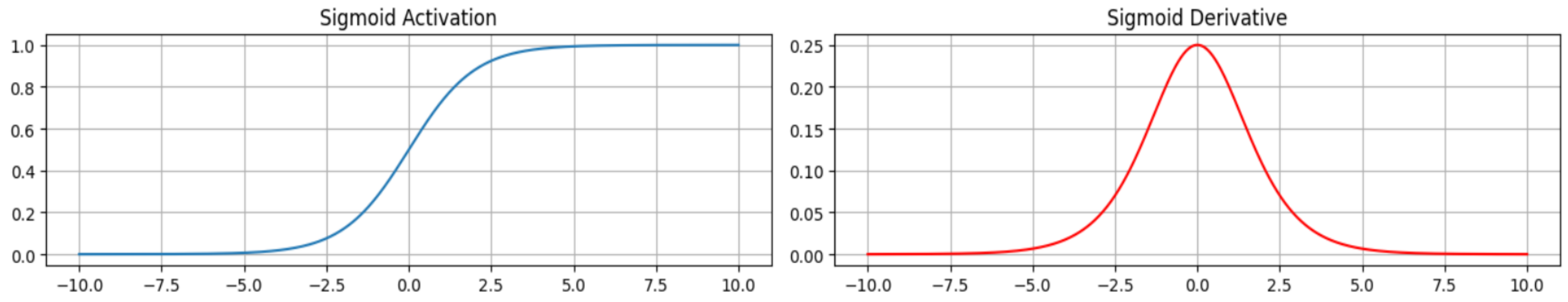
$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \cdot f'(z)$$

If f′(z)is too small (like sigmoid near saturation), the gradient vanishes → **no learning**.

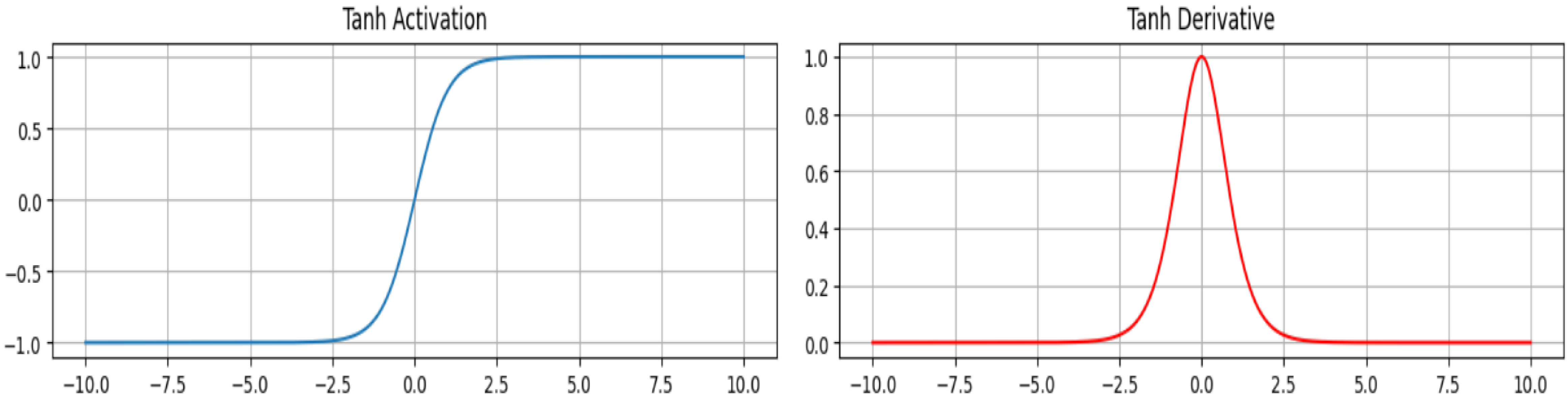
If f′(z) is huge, gradient explodes → **unstable training**.

```python
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

# Input range
x = np.linspace(-10, 10, 500, dtype=np.float32)

# Activation functions
sigmoid = tf.nn.sigmoid(x).numpy()
tanh = tf.nn.tanh(x).numpy()
relu = tf.nn.relu(x).numpy()
leaky_relu = tf.nn.leaky_relu(x, alpha=0.1).numpy()

# Derivatives
# Sigmoid derivative: σ(x)*(1 - σ(x))
sigmoid_deriv = sigmoid * (1 - sigmoid)

# Tanh derivative: 1 - tanh(x)^2
tanh_deriv = 1 - np.power(tanh, 2)

# ReLU derivative: 0 if x<0 else 1
relu_deriv = np.where(x > 0, 1, 0)

# Leaky ReLU derivative: alpha if x<0 else 1
leaky_relu_deriv = np.where(x > 0, 1, 0.1)
```

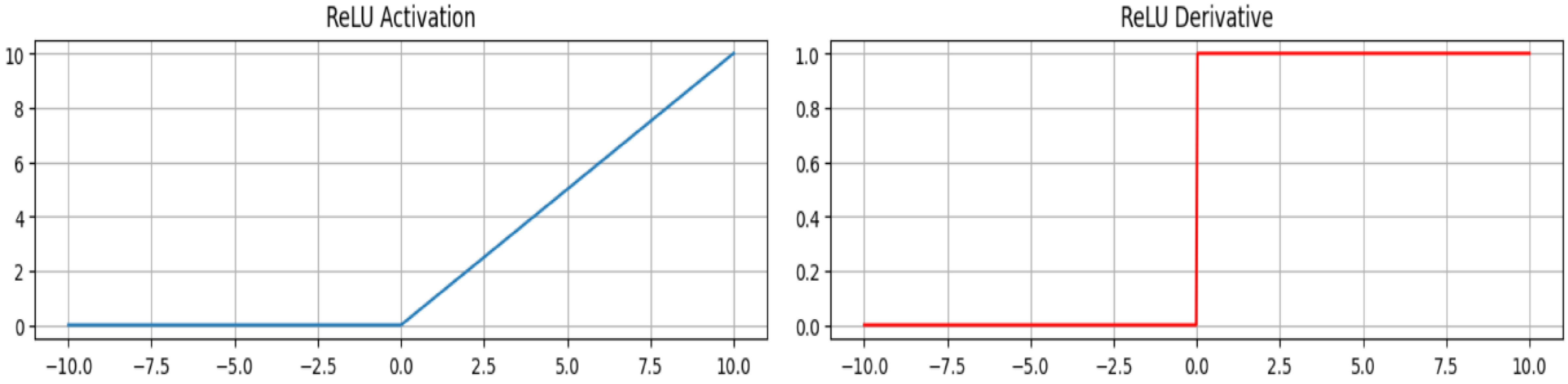
```python
# Plot
plt.figure(figsize=(14,10))

# Sigmoid
plt.subplot(4,2,1)
plt.plot(x, sigmoid, label="Sigmoid")
plt.title("Sigmoid Activation")
plt.grid(True)

plt.subplot(4,2,2)
plt.plot(x, sigmoid_deriv, label="Sigmoid'", color="red")
plt.title("Sigmoid Derivative")
plt.grid(True)
```



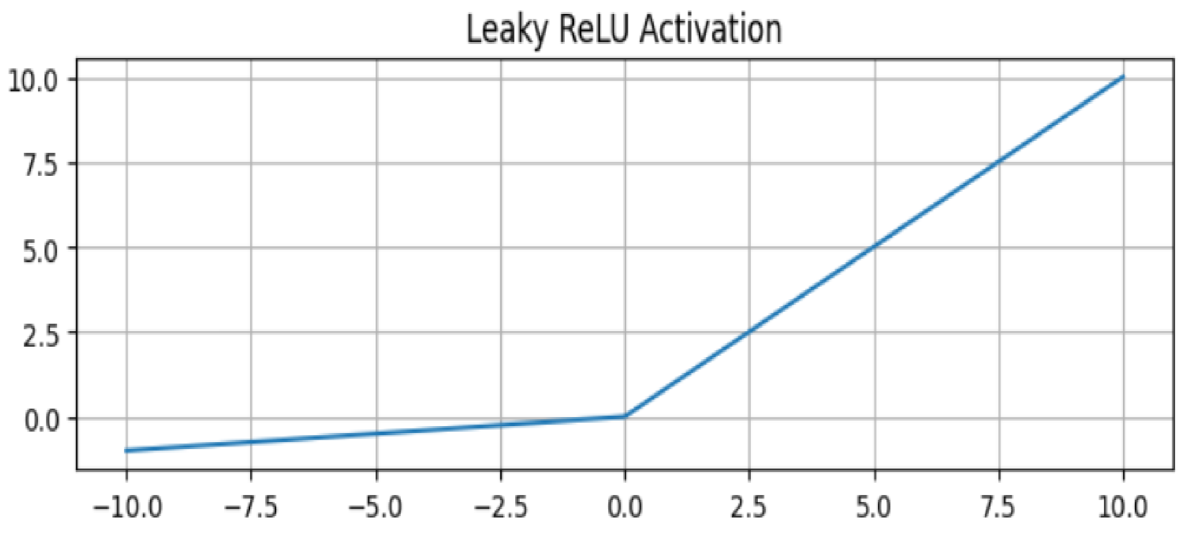
```python
# Tanh
plt.subplot(4,2,3)
plt.plot(x, tanh, label="Tanh")
plt.title("Tanh Activation")
plt.grid(True)

plt.subplot(4,2,4)
plt.plot(x, tanh_deriv, label="Tanh'", color="red")
plt.title("Tanh Derivative")
plt.grid(True)
```

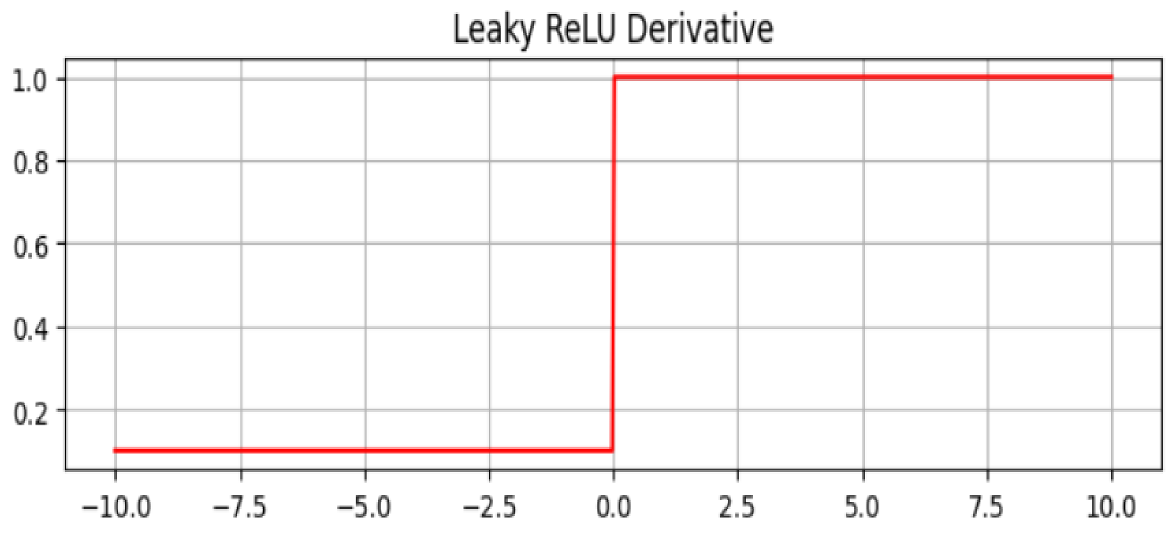


```
# ReLU
plt.subplot(4,2,5)
plt.plot(x, relu, label="ReLU")
plt.title("ReLU Activation")
plt.grid(True)

plt.subplot(4,2,6)
plt.plot(x, relu_deriv, label="ReLU'", color="red")
plt.title("ReLU Derivative")
plt.grid(True)
```



```python
# Leaky ReLU
plt.subplot(4,2,7)
plt.plot(x, leaky_relu, label="Leaky ReLU")
plt.title("Leaky ReLU Activation")
plt.grid(True)

plt.subplot(4,2,8)
plt.plot(x, leaky_relu_deriv, label="Leaky ReLU'", color="red")
plt.title("Leaky ReLU Derivative")
plt.grid(True)

plt.tight_layout()
plt.show()
```



Sigmoid → outputs between (0,1), but derivative becomes very small for |x|>5 → vanishing gradient problem.

Tanh → centered around 0, derivative stronger near 0, but still vanishes for large |x|.

ReLU → derivative is 0 for negative x (dead neurons) but 1 for positive x → efficient gradient flow.

Leaky ReLU → fixes dead ReLU by keeping a small slope on negative side.

# THANK YOU