Artificial Neural Network (ANN)

Basic Components of an Artificial Neural Network

1. Neurons (Nodes)
2. Layers
3. Connections
4. Activation Function

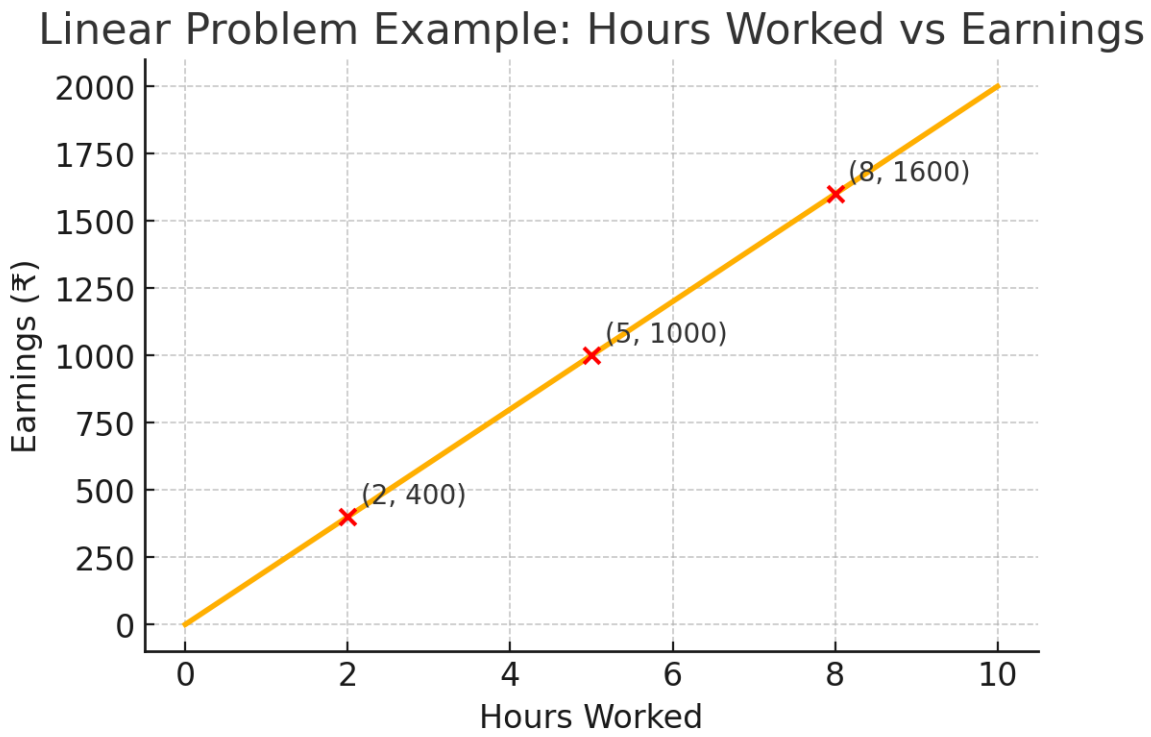
Linear Problem

Think of a **straight line** on a graph — if the relationship between inputs and outputs can be represented as a straight line, it's a **linear problem**.

Example (real life):

- You are paid **₹200 per hour**.
- Work more hours → earn more money.
- The formula is:
  Earnings = 200 × Hours worked
- This is a **linear** relationship because every extra hour gives you the same extra money.

Key idea: One direction, no bends, simple proportion.



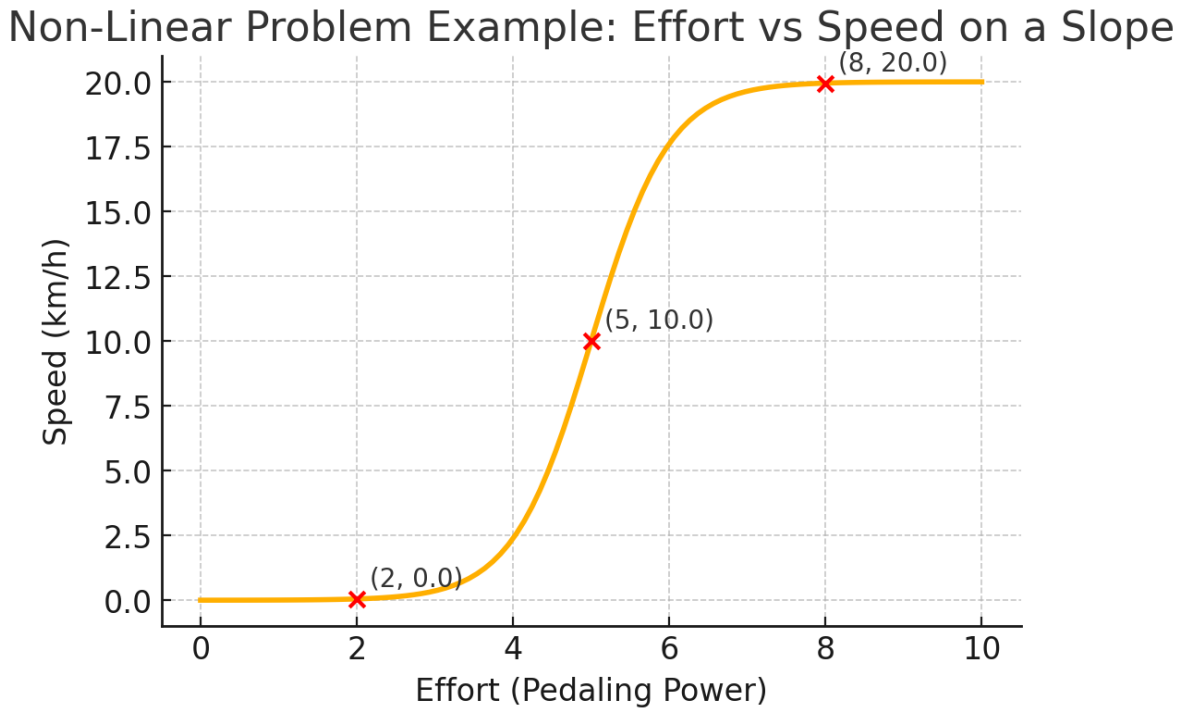
Non-Linear Problem

If the relationship **curves, bends, or changes** depending on where you are, it's **non-linear**.

Example (real life):

- You ride a cycle uphill.
- At first, speed increases fast when you pedal harder.
- But as the slope gets steeper, no matter how much harder you pedal, speed doesn't increase the same way.
- The graph of "Effort vs Speed" is **curved**, not a straight line.

Key idea: The effect of change depends on the situation — the "gain" isn't constant.

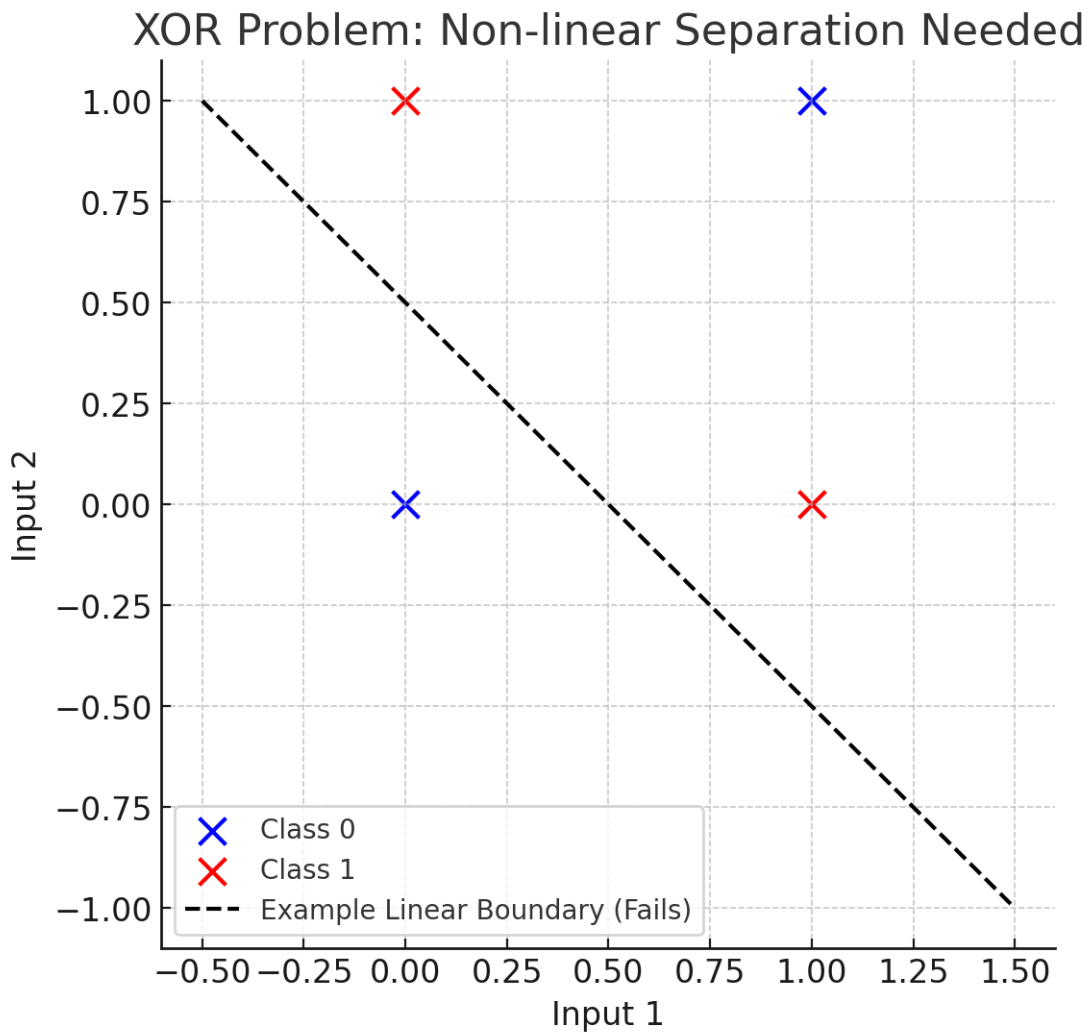


What is XOR?

XOR (Exclusive OR) is a logic rule:

- If **exactly one** of the two inputs is 1 → output is **1**.
- If both are the same (both 0 or both 1) → output is **0**.

Input 1	Input 2	Output (XOR)
0	0	0
0	1	1
1	0	1
1	1	0



The real issue

If you plot these four cases on a graph:

- **Blue dots** (output 0) are at **(0,0)** and **(1,1)**.
- **Red dots** (output 1) are at **(0,1)** and **(1,0)**.

The red and blue points are diagonally opposite.
No matter how you draw **a single straight line**, you cannot separate the red from the blue completely — at least one point will always be on the wrong side.

Why a linear function fails

A **linear function** creates a single flat boundary (like a ruler).
But XOR's points require **two boundaries** or a **curved boundary** to separate them — and a single straight line can't bend or split into two.

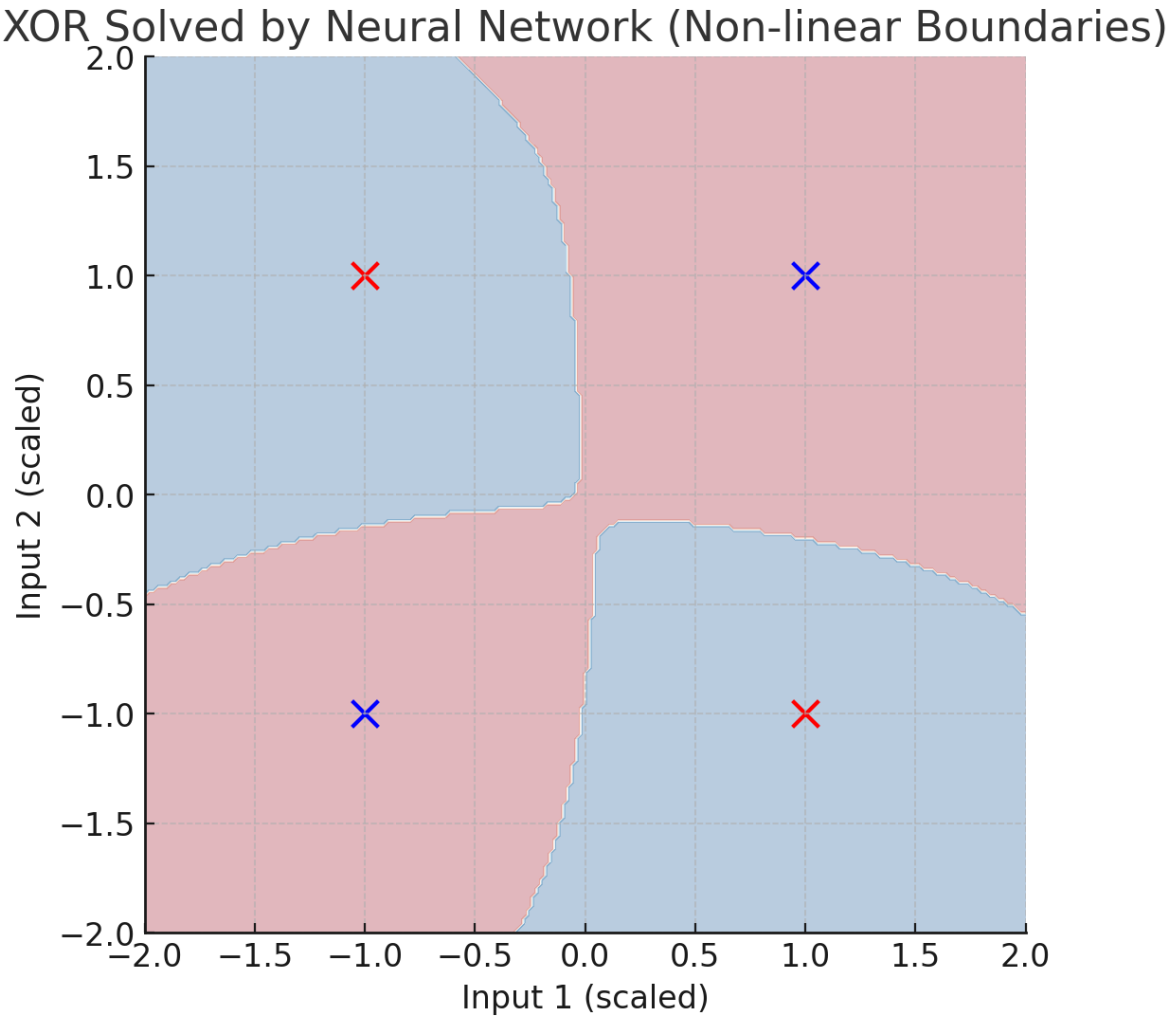
How non-linear activation solves it

When we add a hidden layer with **non-linear activation functions** (like ReLU, sigmoid, or tanh):

- The first layer can create multiple intermediate boundaries.

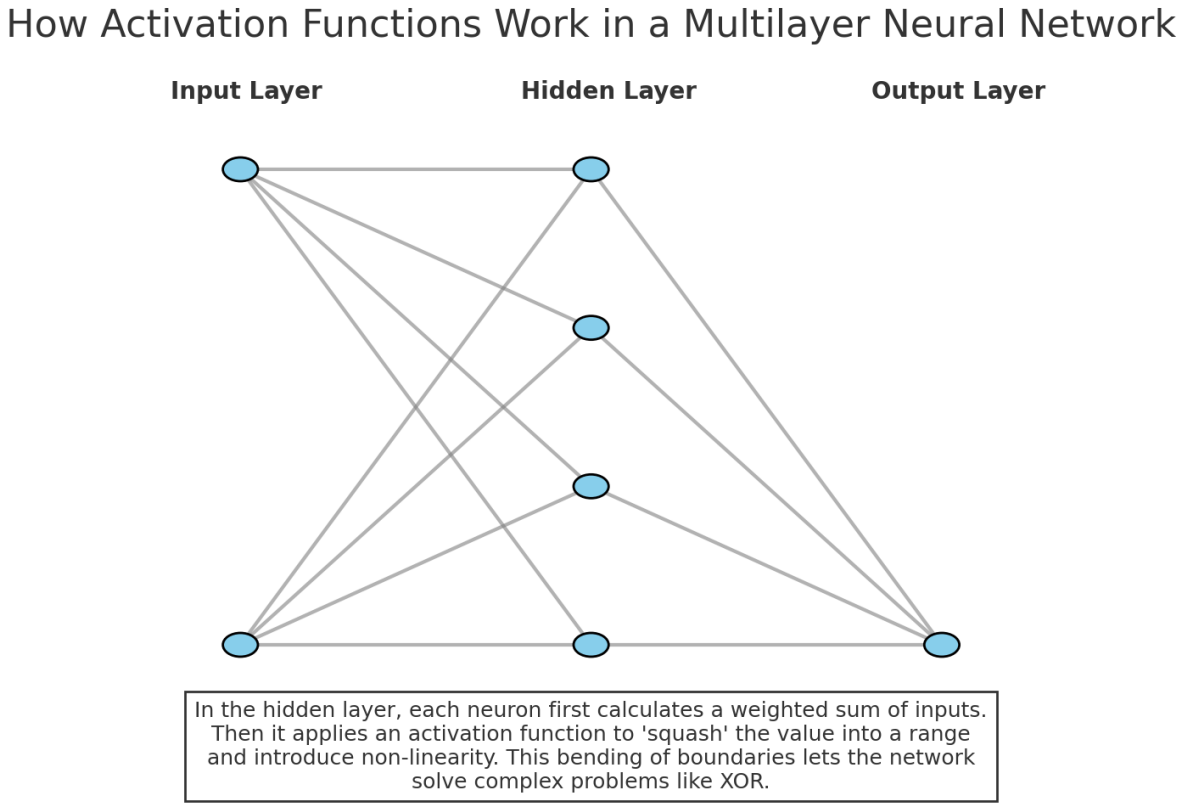
- The second layer can combine those boundaries into a shape that correctly separates red from blue.

Essentially, non-linearity lets the network **"bend the line"** into complex shapes — exactly what XOR needs.



Why Activation Function?

- An activation function introduces non-linearity into the model, enabling it to learn complex patterns beyond simple linear relationships.
- Without activation functions, a neural network would behave like a basic liner regression model



Types of Activation Function

Sigmoid Activation Function

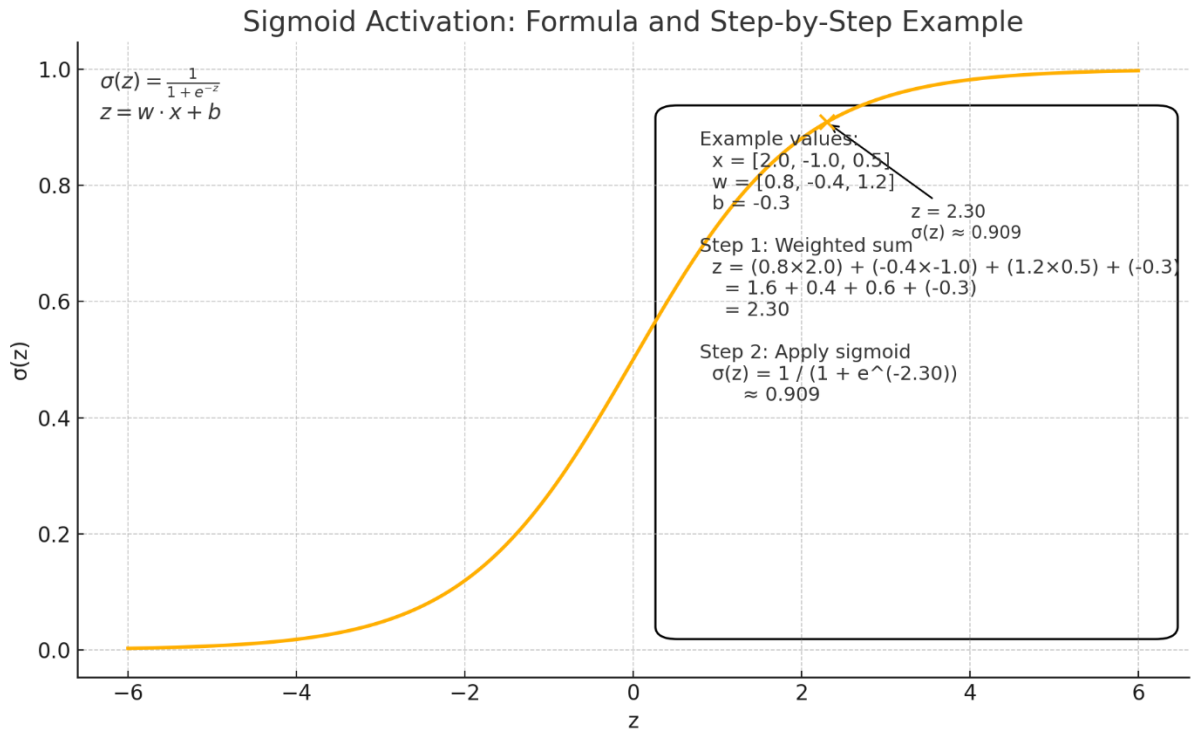
Used in binary classification problems

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

Output range: (0,1)

Advantage: It is useful binary classification

Disadvantage: Suffers from Gradient Vanishing Problem



Derivative of Sigmoid

If we differentiate $\sigma(z)$ with respect to z, we get:

$$f'(z) = f(z).(1 - f(z))$$

So — instead of re-calculating the full derivative from scratch each time, we can just use the sigmoid output we already have and multiply it by (1−output).

How it is derived?

1. **Start with the sigmoid (logistic) function**

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

2. **Differentiate using the chain rule (treat $u(z) = 1 + e^{-z}$)**

$$\boldsymbol{\frac{d}{dx}\left(u^{-1}\right) = -u^{-2}u'(z)}$$

We need $u'(z)$:

$$\boldsymbol{u'(z) = \frac{d}{dx}(1) + \frac{d}{dx}(e^{-z}) = 0 + (-e^{-z}) = -e^{-z}}$$

3. **Plug back into the chain rule**

$$\boldsymbol{\sigma'(z) = -(1 + e^{-z})^{-2}(-e^{-z}) = \frac{e^{-z}}{(1 + e^{-z})^2}}$$

4. **Express the result in terms of σ(z)σ(z) itself**

Notice:

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

Multiply these:

$$\boldsymbol{\sigma(z)\big(1-\sigma(z)\big) = \frac{1}{1+e^{-z}} * \frac{e^{-z}}{1+e^{-z}} = \frac{e^{-z}}{(1+e^{-z})^2}}$$

5. Conclude

$$f\sigma'(z) = \sigma(z).(1-\sigma(z))$$

Why is it important?

We use the derivative in **backpropagation** to update the weights in a neural network.

- In backpropagation, we need to know how much each weight influenced the error.
- The derivative tells us **how sensitive the output is to small changes in the input** at that point.
- For sigmoid, the derivative is largest near z = 0 (output ≈ 0.5) and very small near the extremes (output ≈ 0 or 1).

Key Insight

- If the derivative is too small (when sigmoid output is near 0 or 1), **learning becomes very slow** — this is part of the **vanishing gradient problem**.
- This is one reason why in deep networks, ReLU or other activation functions are often preferred.

Example

Say the sigmoid output is 0.9:

$$\sigma'(z) = 0.9 * (1 - 0.9) = 0.9 * 0.1 = 0.09$$

That means the slope is small, so changes in z will only slightly affect the output — the neuron is "less responsive" here.

Email Spam Detection (Binary Classification)

Problem:

We want to classify emails as either:

- **Spam** (1)
- **Not Spam** (0)

Why Sigmoid Works Well Here:

1. **Output Range Matches Probability**
    - Sigmoid outputs values between **0 and 1**.

- In spam detection, we want a probability:
  - Output ≈ 0.95 → *"95% chance this is spam"*
  - Output ≈ 0.1 → *"10% chance this is spam"*

2. **Smooth Thresholding**
   - We can set a decision boundary at **0.5**:
     - If output ≥ 0.5 → classify as spam.
     - If output < 0.5 → classify as not spam.
   - But we still have flexibility — we could use 0.7 if we want fewer false positives.
3. **Interpretable Output for Humans**
   - A business can use the probability directly to **rank emails by spam likelihood**.
   - Instead of just "spam/not spam," the system can say *"sort by spam probability."*
4. **Works Great for the Last Layer in Binary Classification**
   - Inside the network, we might use ReLU or other activations.
   - But for the **final output neuron**, sigmoid is ideal because it gives a probabilistic output between 0 and 1.

Workflow Example:

1. **Input:** Email features (number of links, sender reputation, suspicious keywords, etc.).
2. **Neural Network Processing:** Hidden layers with ReLU → Final layer with **sigmoid**.
3. **Sigmoid Output:** 0.88 → interpreted as *88% chance of being spam*.
4. **Decision:** Flag as spam because probability > 0.5.

💡 **Key takeaway:** Sigmoid is best for **binary classification problems** where the output is a probability. It's less useful for multi-class problems (there we use **SoftMax**) or very deep layers (due to vanishing gradient).

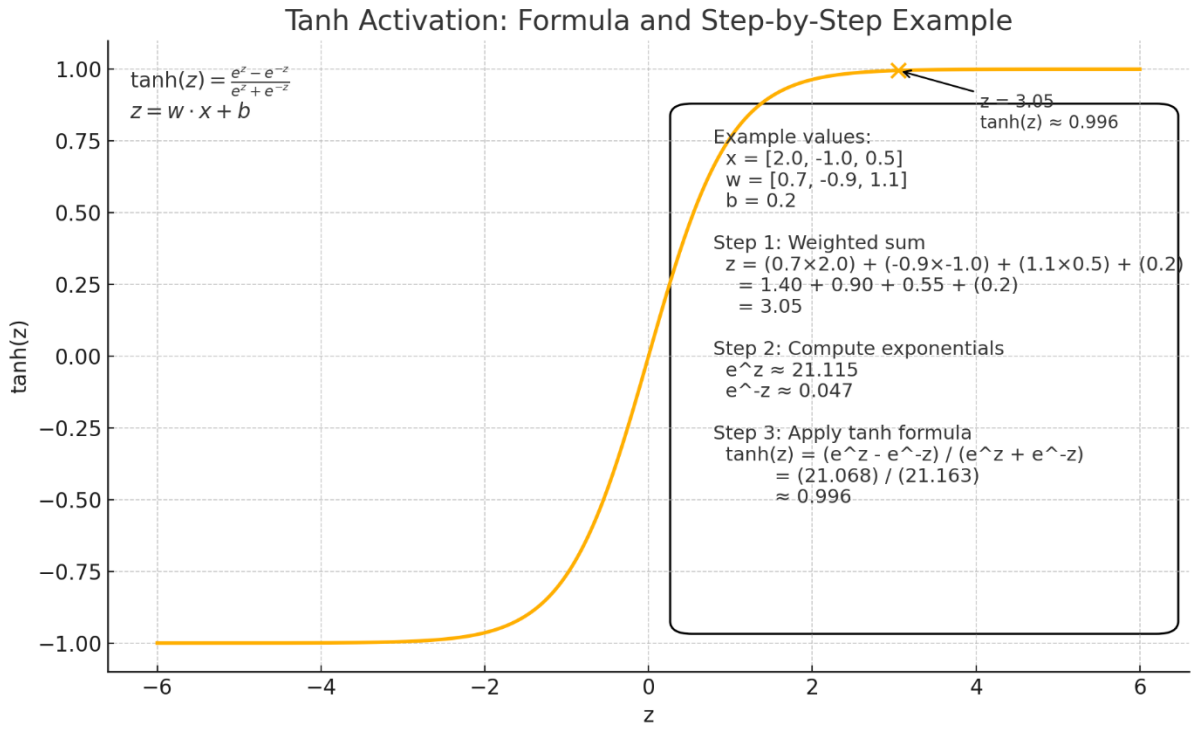
Hyperbolic Tangent (TanH) Activation Function

Similar to Sigmoid but with better gradient properties.

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

Output range: (-1, 1)

Advantage: This function is Zero-Centered.



Derivative of TanH

$$f'(z) = 1 - (f(z))^2$$

Meaning:
If you already have $\tanh(z)$ from the forward pass, you can compute its derivative just by doing **1 minus the square of that output** — very fast to calculate.

Why do we use the derivative?

In **backpropagation** (training neural networks):

- We adjust weights based on how much each neuron's output affects the loss.
- The derivative tells us **how sensitive the tanh output is to changes in z**.
- This slope is multiplied by the error signal to decide how much to change the weights.

4. Advantages of tanh derivative

- **Centered around zero**: Outputs in range (-1, 1), so the derivative can be larger in magnitude for inputs around zero → faster convergence compared to sigmoid (which outputs 0 to 1).
- **Smooth gradient**: Continuous and differentiable everywhere, making optimization stable.

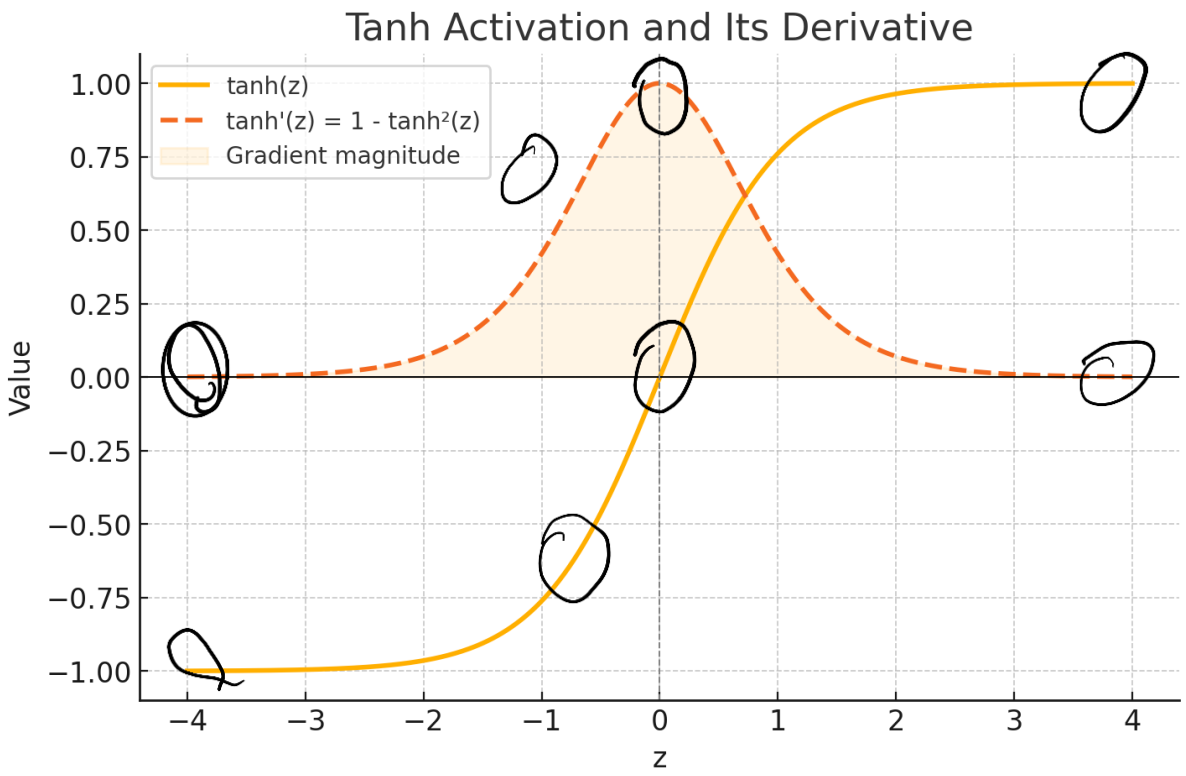
5. Example

If $\tanh(z) = 0.8$:

$$\frac{d}{dx}\tanh(z) = 1 - (0.8)^2 = 1 - 0.64 = 0.36$$

This means a small change in z changes the output by about **0.36× that change**.

💡 **Key takeaway:**
We use the derivative in training so the network knows *how much* and *in what direction* to adjust each weight.
For tanh, it's both easy to compute and gives better-centered gradients than sigmoid — which often makes training faster.



Here's the chart:

- **Blue curve**: $\tanh(z)$ — the activation output, ranging from -1 to 1.
- **Orange dashed curve**: $1 - tanh(z)^2$ — the derivative (slope).
- The shaded area shows the **gradient magnitude** — largest near z=0, meaning the neuron learns fastest there.
- At the extremes (output near -1 or 1), the slope is near zero → **learning slows down**.

Start with the definition

$$\boldsymbol{\tanh(z) = \frac{\sinh(z)}{\cosh(z)}}$$

Where

$$\sinh(z) = \frac{e^z - e^{-z}}{2}$$

$$\cosh(z) = \frac{e^z + e^{-z}}{2}$$

Step 1: Apply the quotient rule

For $u(z) = \sinh(z)$ and $v(z) = \cosh(z)$,

$$\left(\frac{u}{v}\right)' = \frac{u'v - uv'}{v^2}.$$

Plug in u, u', v, v':

$$\frac{d}{dz}\tanh(z) = \frac{\cosh(z)\cdot\cosh(z) - \sinh(z)\cdot\sinh(z)}{\cosh^2(z)} = \frac{\cosh^2(z) - \sinh^2(z)}{\cosh^2(z)}.$$

Meaning: you're finding how fast the ratio sinh/cosh changes by combining the changes of top and bottom.

Step 2: Use the hyperbolic identity

A key identity is:

$$\cosh^2(z) - \sinh^2(z) = 1.$$

So the numerator becomes 1:

$$\frac{d}{dz}\tanh(z) = \frac{1}{\cosh^2(z)}.$$

This is also written as:

$$\frac{d}{dz}\tanh(z) = \operatorname{sech}^2(z),$$

since $\operatorname{sech}(z) = 1/\cosh(z)$.

Step 3: Express it in terms of $tanh(z)$

Because $\tanh(z) = \dfrac{\sinh(z)}{\cosh(z)}$,

$$\tanh^2(z) = \frac{\sinh^2(z)}{\cosh^2(z)}.$$

Then

$$1 - \tanh^2(z) = \frac{\cosh^2(z) - \sinh^2(z)}{\cosh^2(z)} = \frac{1}{\cosh^2(z)}.$$

Therefore:

$$\boxed{\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z)}.$$

Quick intuition check

- At $z = 0$: $\tanh(0) = 0 \rightarrow tanh'(0) = 1 - 0 = 1$.
  The slope is steepest at the center → the neuron learns fastest there.
- For large $|z|$: $\tanh(z) \rightarrow \pm 1 \rightarrow tanh'(z) \rightarrow 0$.
  The slope flattens at the extremes → learning slows down (small gradient).

Sentiment Analysis (Positive vs Negative)

Problem:

We want to classify a short customer review as:

- **Negative sentiment** → -1
- **Positive sentiment** → +1

Instead of using outputs in [0,1] like sigmoid, we want outputs centered around **0** so the model naturally treats negative and positive signals symmetrically.

Why Tanh is Advantageous Here:

1. **Zero-centered outputs (-1 to 1)**
   - When processing word embeddings, tanh maps strongly negative sentiment features to negative values and strongly positive ones to positive values.
   - This helps the next layers learn faster because the activations aren't all positive (avoids bias shifts in weight updates).

2. **Better gradient flow than sigmoid**
    - Around the center $(z \approx 0), tanh'(z) \approx 1$, so gradients are strong.
    - This means faster learning for inputs near the decision boundary.
3. **Derivative in Backpropagation**
    - During training, we compute:

      $$tanh'(z) = 1 - tanh'(z)$$

      If tanh output is 0.6 (moderately positive sentiment), the derivative is:

      $$1 - 0.36 = 0.64$$

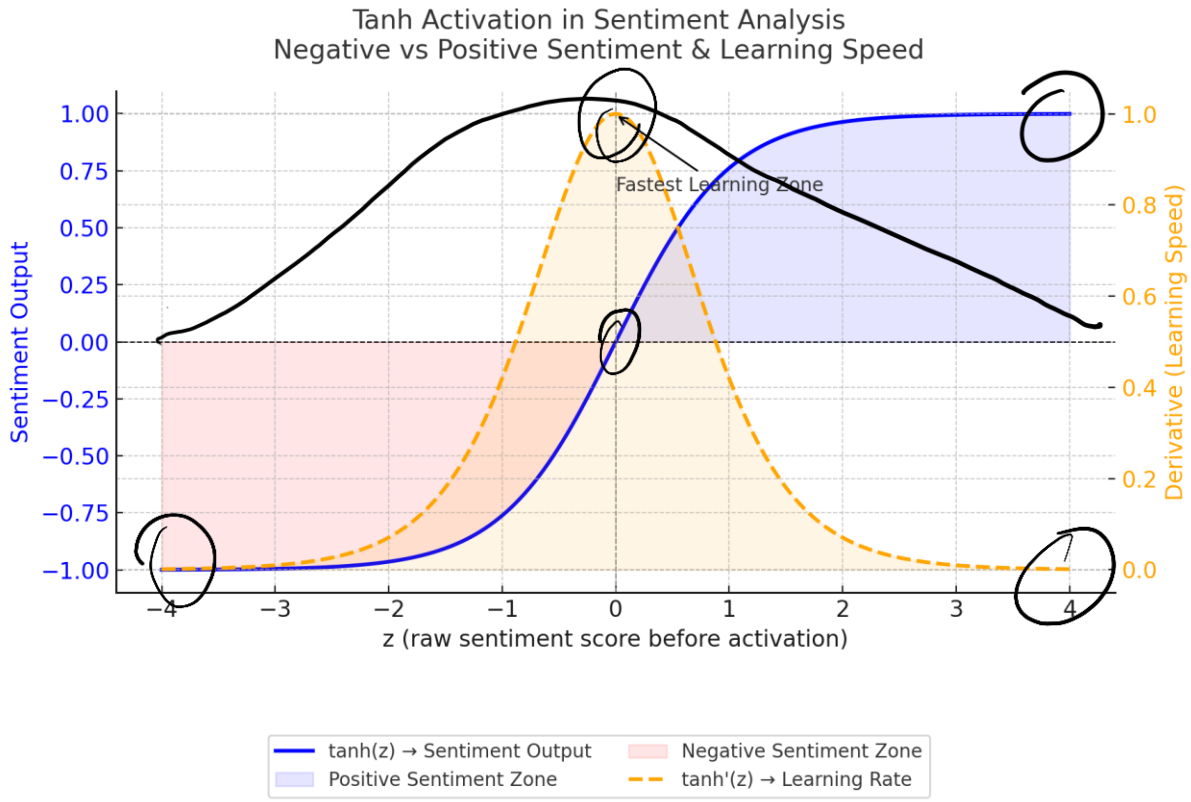
      This tells backprop how much to adjust the weights — moderate enough to fine-tune, but not too small.
4. **Symmetry for Opposite Classes**
    - If the same features appear but with opposite signs (positive vs negative words), tanh will output opposite activations, making classification easier.

Workflow Example

1. **Input:**
    - Word embeddings for:
        - "Terrible service" → negative numbers dominate.
        - "Excellent service" → positive numbers dominate.
2. **Hidden layer activation:**
    - Uses **tanh** to squash values between -1 and 1.
3. **Output layer:**
    - Could be a single neuron with sigmoid if you want probabilities or stay with tanh if you keep -1/+1 outputs.
4. **During training:**
    - Derivative $1 - \tanh(z)^2$ tells backprop how to adjust weights for each feature's influence on sentiment.

💡 **Key takeaway:**

- Use **tanh** when your data or decision boundary is naturally **centered around zero** and you want **symmetry between positive and negative outputs**.
- Its derivative is simple, quick to compute, and provides better gradient flow than sigmoid in the central region.



Here's the chart showing **tanh in sentiment analysis**:

- **Blue curve** = Sentiment output after tanh activation.
    - Left (red zone) → negative sentiment.
    - Right (blue zone) → positive sentiment.
- **Orange dashed curve** = Derivative (learning speed).
    - Highest at $z \approx 0$ → the model learns fastest for neutral or borderline reviews.
    - Slower learning at extreme positive or negative sentiments.

This visually shows why tanh is useful for **zero-centered** tasks like sentiment analysis

ReLU (Rectified Linear Unit) Activation Function

Used in hidden layers to introduce non-linearity while keeping computation efficient.

$$f(z) = \max(0, z) = \begin{cases} z, if\ z \geq 0 \\ 0, if\ z < 0 \end{cases}$$

Output range: $[0, \infty]$

Advantage: ReLU doesn't suffer from vanishing gradients for positive inputs only

Derivative of ReLU

The derivative is:

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

Meaning:

- For **positive inputs**, the slope is **1** → output changes at the same rate as input.
- For **negative inputs**, the slope is **0** → no change in output, no gradient flows backward.

3. Why We Use It

- **Efficient computation**: No exponentials, just a simple comparison — faster than sigmoid or tanh.
- **Avoids vanishing gradient (mostly)**: For $z > 0$, gradient = 1, so training signals don't shrink.
- **Sparse activation**: Many outputs are exactly zero, making the network more efficient and helping with generalization.

4. Limitation

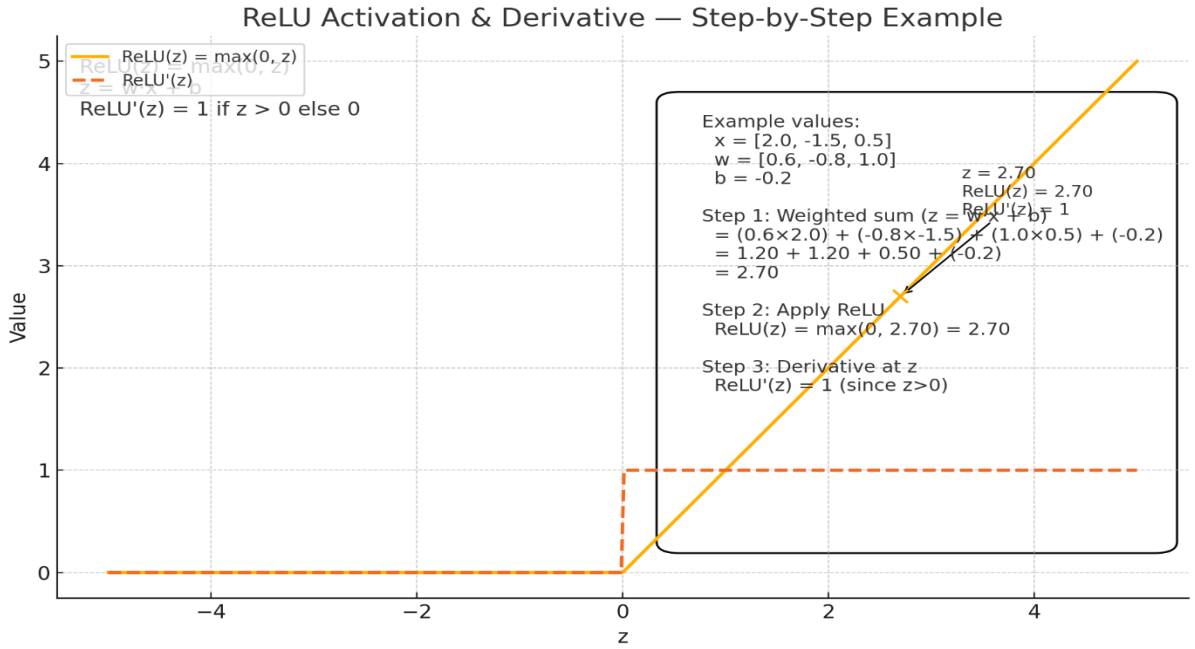
- **Dying ReLU problem**: If too many neurons get $z \leq 0$, their gradient is 0 forever, and they stop learning.
    - Solutions: **Leaky ReLU**, **Parametric ReLU** (PReLU), or **ELU**.

5. Example

If z=3.5 → ReLU = 3.5, derivative = 1.
If z=−2.0 → ReLU = 0, derivative = 0.

During **backpropagation**:

- Positive input neurons pass the gradient back unchanged.
- Negative input neurons block the gradient entirely.



1) Start with the formula

$$\mathrm{ReLU}(z) = \max(0, z) = \begin{cases} 0, & z \le 0 \\ z, & z > 0 \end{cases}$$

2) Take the derivative piece by piece

- **Case A:** $z > 0$

  Here ReLU(z) = z (it's just the identity line).

  The slope of z with respect to z is **1**.

$$\frac{d}{dz}\mathrm{ReLU}(z) = 1 \quad (z > 0)$$

- **Case B:** $z < 0$

  Here ReLU(z) = 0 (a flat constant).

  The slope of a constant is **0**.

$$\frac{d}{dz}\mathrm{ReLU}(z) = 0 \quad (z < 0)$$

- **Case C:** $z = 0$

  The graph has a **kink** at 0.

  - Left-hand slope (coming from $z < 0$) = 0
  - Right-hand slope (coming from $z > 0$) = 1

    Since left ≠ right, the derivative at exactly 0 **doesn't exist** (not differentiable).

    In practice for backprop, frameworks choose a **subgradient** there, most commonly **0**.

3) Put it together (what we use in code)

$$\mathrm{ReLU}'(z) = \begin{cases} 1, & z > 0 \\ 0, & z < 0 \\ (\text{choose } 0 \text{ at } z = 0 \text{ in practice}) \end{cases}$$

Example: Detecting Objects in Images (e.g., Self-driving Car Pedestrian Detection)

Problem:

A self-driving car's neural network needs to process camera images and decide:

- Is there a pedestrian in front?
- Where exactly is the pedestrian?

This requires **deep convolutional neural networks (CNNs)** — often with **dozens of layers** — to process pixel data.

Why ReLU Works Well Here

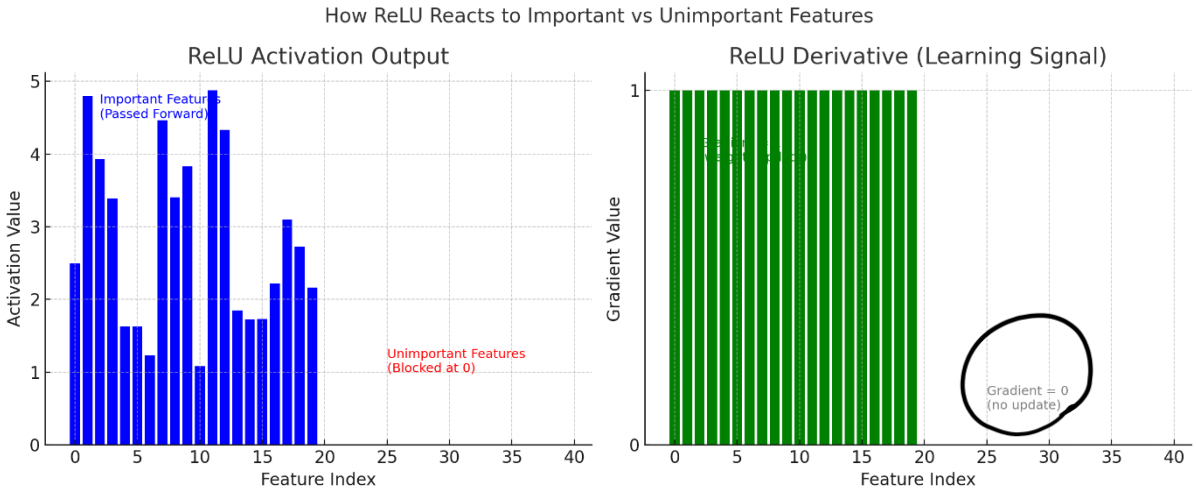
1. **Avoids Vanishing Gradient in Deep Networks**
    - Sigmoid/tanh squashes values, making gradients very small in deep layers.
    - ReLU's derivative is **1** for positive inputs, so gradients flow without shrinking → faster training and better convergence.
2. **Computationally Cheap**
    - ReLU just applies $\max(0, z)$, no exponentials.
    - This makes training CNNs on large image datasets much faster.
3. **Sparse Activation (Efficient Feature Learning)**
    - Negative inputs become 0 → neurons turn off.
    - This means only part of the network activates for a given image, helping reduce overfitting and improving efficiency.
4. **Derivative Behavior Matches Image Data Needs**
    - For positive activations (important features in the image), derivative = **1** → strong gradient signal to learn those features.
    - For irrelevant/negative activations, derivative = **0** → network ignores unimportant regions of the image.

Example in Action

Imagine a CNN layer detecting **edges** in an image:

- A strong vertical edge in front of a pedestrian → activation $z > 0$ → ReLU passes it, derivative = 1 → weights for detecting vertical edges get updated.
- A flat gray road region → $z < 0$ → ReLU outputs 0, derivative = 0 → no learning for irrelevant patterns.

💡 **Key Takeaway:**
ReLU and its derivative are ideal for **large, deep networks** like those in **image recognition, speech recognition, and object detection**, because they keep gradients strong for important features and allow deep models to train efficiently.



SoftMax: Used for multi class classification

The **Softmax** function converts a vector of real numbers into a probability distribution. It ensures that:

- All outputs are between **0 and 1**.
- The sum of all outputs equals **1**.

Formula for a vector $z = [z_1, z_2, \dots \dots, z_K]$:

$$Softmax(Z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Where:

- e^{z_i} is the exponential of the i[th] logit (raw score).
- K is the number of classes.

Derivative of Softmax

The derivative of Softmax is slightly more complex because it produces a vector output and is often used in combination with **cross-entropy loss**.

Jacobian Matrix of the Softmax output $\sigma(z)$:

$$\frac{\partial \sigma_i}{\partial z_j} = \begin{cases} \sigma_i(1 - \sigma_i) & \text{if } i = j \\ -\sigma_i \sigma_j & \text{if } i \neq j \end{cases}$$

Where:

- σ_i = Softmax output for class i.

Special Case (with Cross-Entropy Loss):

If we use Softmax followed by cross-entropy loss, the derivative simplifies to:

$$\frac{\partial L}{\partial z_i} = \sigma_i - y_i$$

Where y_i is the true label (one-hot encoded).

Example

Suppose we have three logits:

$$z = [2.0, 1.0, 0.1]$$

Step 1: Calculate exponentials

$$e^z = [e2, e1, e0.1] \approx [7.389, 2.718, 1.105]$$

Step 2: Sum of exponentials

$$S = 7.389 + 2.718 + 1.105 \approx 11.212$$

Step 3: Divide each exponential by the sum

$$Softmax(z) \approx [0.659, 0.242, 0.099]$$

This means:

- Class 1 has a 65.9% predicted probability
- Class 2 has a 24.2% predicted probability
- Class 3 has a 9.9% predicted probability

Use Cases

- **Multi-class classification** problems (output layer) — e.g., image classification in CNNs.
- **Language modeling** — to get probability distribution over vocabulary words.
- **Attention mechanisms** in Transformers — to normalize attention scores.
- **Reinforcement learning** — for stochastic policy output.

Advantages

1. **Probability Interpretation** – Outputs are easy to interpret as probabilities.
2. **Normalization** – Ensures outputs sum to 1, making comparisons straightforward.
3. **Works Well with Cross-Entropy Loss** – The math simplifies gradients, making optimization efficient.
4. **Stable for Multi-class Outputs** – Suitable for handling many classes.

Disadvantages

1. **Sensitive to Large Logits** – Large values can cause **numerical instability** (mitigated using log-sum-exp trick).
2. **Not Sparse** – All outputs are nonzero, which can slow training compared to ReLU.
3. **Poor for Imbalanced Data** – Might be biased towards majority classes if not handled properly.
4. **Can Overfit in High Confidence Cases** – Tends to produce overconfident predictions if regularization is not applied.