# Module 2

## Deep Learning Fundamentals

## 📌 Gradient Descent

In neural networks, learning consists of **optimizing the weights and biases** of the model to reduce prediction error. This is achieved by minimizing a **cost function** (also known as a loss function), which quantifies the difference between the predicted output and the actual target values.

Before discussing how neural networks perform this optimization, it's essential to understand **gradient descent**—the foundational algorithm that drives the training process in deep learning.

### 🔹 Understanding the Cost Function

Suppose we have a simple linear relationship between two variables:

To find the best possible value of the weight www that minimizes the prediction error between the actual values of and the predicted values , we define a **cost function**.

This function penalizes the squared differences between the predicted and actual values, and the ​ term ensures that the cost is independent of the dataset size.

* : number of samples
* ​: actual target value
* ​: predicted value using current weight

For a perfectly linear case like, this cost function has a parabolic shape with a unique global minimum at .

### 🔹 What is Gradient Descent?

**Gradient descent** is an iterative optimization algorithm used to find the **minimum** of a function. In machine learning, it is used to minimize the cost function by adjusting model parameters.

**🧭 Basic Intuition:**

* Imagine standing on a curved surface representing the cost function.
* The **gradient** at any point tells you the direction of **steepest ascent**.
* To minimize the cost, move in the opposite direction: **steepest descent**.

The algorithm begins with an initial guess for the parameter , denoted as ​, and gradually updates it in the direction that reduces the cost.

### 🔹 Gradient Descent Formula

Given:

* ​: Initial weight
* : Learning rate (controls step size)
* ​: Gradient of the cost function with respect to .

The weight update rule:

Each iteration involves:

1. Computing the gradient (slope) at the current weight.
2. Scaling the gradient by the learning rate.
3. Subtracting this value from the current weight to move toward the minimum.

This process continues until convergence—i.e., until the gradient becomes close to zero or the cost stops decreasing significantly.

The **learning rate** is a critical hyperparameter that affects the behavior of the algorithm.

Choosing an appropriate learning rate is essential for stable and efficient training. In practice, it’s often determined through experimentation or learning rate scheduling techniques.

Scenario:

* **Too Large () 🡪** Overshooting the minimum; possibly diverging or oscillating around it.
* **Too Small () 🡪** Very slow convergence; many iterations needed to reach the minimum.
* **Optimal Range 🡪** Balanced updates; smooth and efficient convergence.

### 🔹 Behavior Around the Minimum

Gradient descent automatically adjusts its step size based on the steepness of the cost curve:

* **Far from the minimum**: Gradient is steep → large steps
* **Near the minimum**: Gradient flattens → smaller steps

This natural slowdown helps the algorithm fine-tune the parameters as it approaches the optimal value.

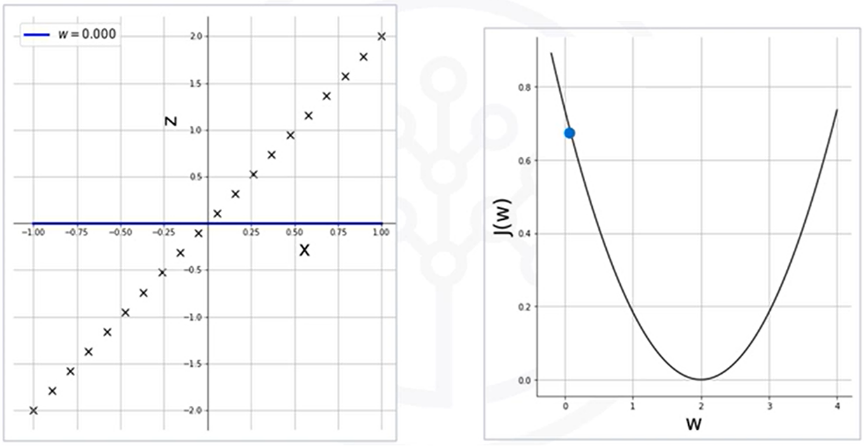
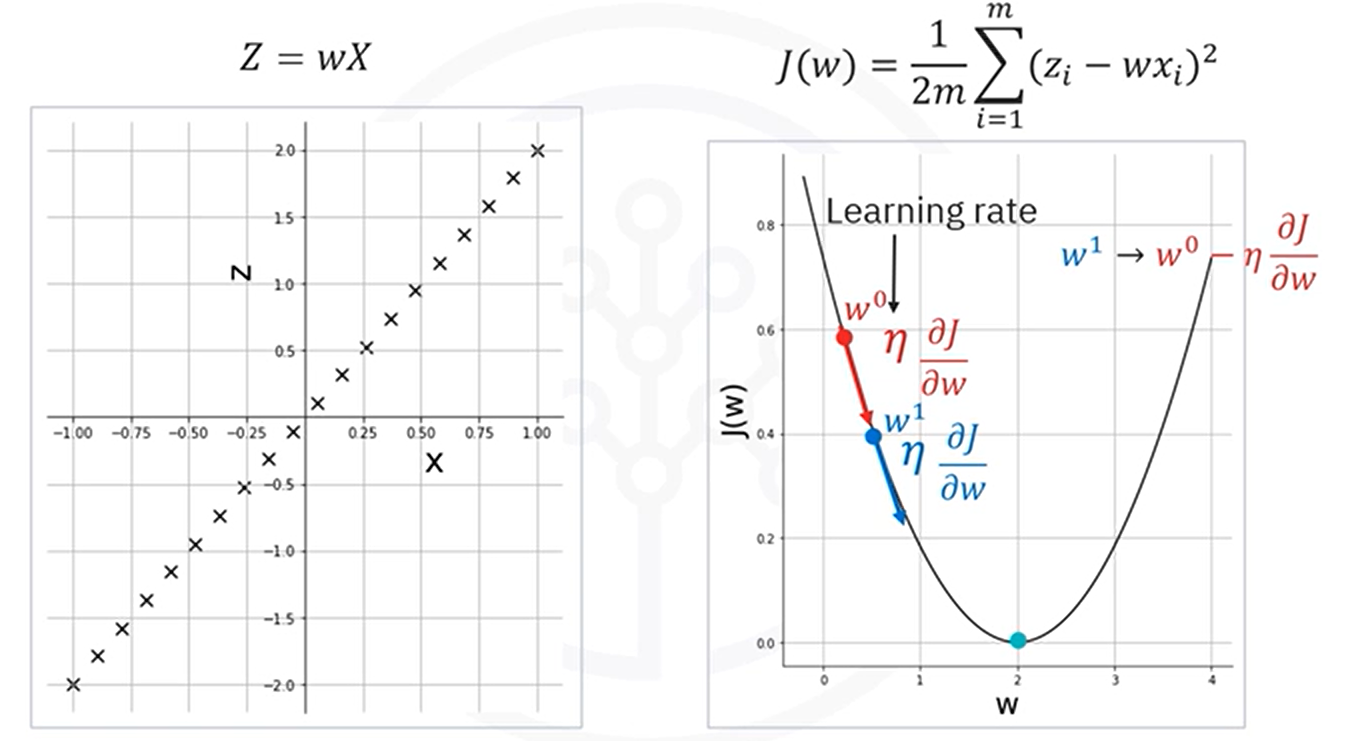
Additionally:

* If initialized **left of the minimum**, the gradient is **negative**, and the update moves to the right
* If initialized **right of the minimum**, the gradient is **positive**, and the update moves to the left

This symmetry ensures convergence to the optimal value from various starting points.

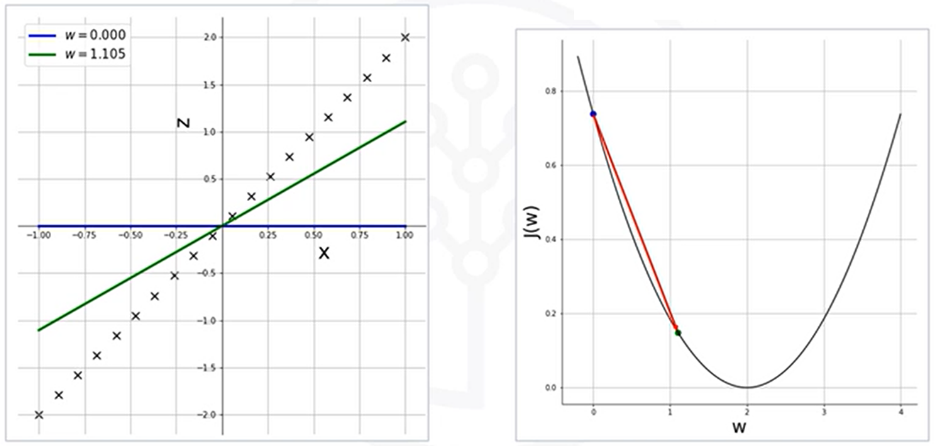
### 🔹 Visualizing the Iterative Optimization

Let’s walk through a numerical example using synthetic data where :

****

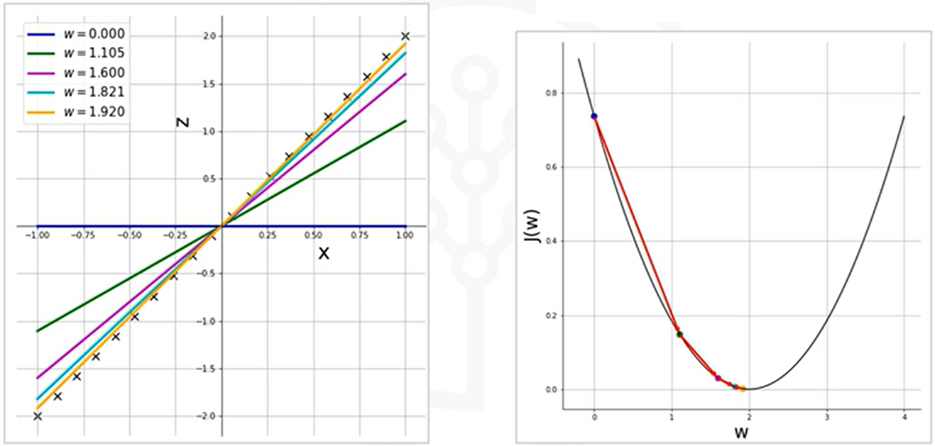
**Initialization:**

* Starts with 🡪 This results in predictions (a flat line).
* This is a **poor fit**, so the **cost is high**.

**Iteration 1:**

* Compute gradient at ​
* Update to ​ using the formula
* The weight jumps closer to 2, resulting in a lower cost and better line fit

**Iteration 2 and Beyond:**

* The slope at the new point is less steep, so the update is smaller
* Each subsequent step brings the weight closer to the optimal value
* The cost continues to drop, and the prediction line aligns more closely with the data

After four iterations, you can see how we're almost there at w equals 2, and the resulting line almost fits the scatter plot perfectly.

With each iteration, the weight is updated in a way that's proportional to the negative of the gradient of the function at the current point. Therefore, if you initialize the weight to a value that is to the right of the minimum, then the positive gradient will result in w moving to the left towards the minimum.

### 🔹 Takeaways

✅ The goal of training a neural network is to **minimize the cost function** by adjusting the weights and biases.

✅ **Gradient descent** is the core algorithm that performs this minimization through **iterative updates**.

✅ The size and direction of updates depend on the **gradient** and the **learning rate**.

✅ A **well-chosen learning rate** is crucial—too high can destabilize training, too low can make it inefficient.

✅ Although this example uses a single weight, the concept extends to **multivariate optimization** in deep networks where thousands or millions of parameters are adjusted simultaneously.

## 📌 Backpropagation

The learning process in a neural network involves minimizing the **error** between the predicted output and the actual (ground truth) labels using a procedure known as **backpropagation**.

This process is central to **supervised learning**, where the dataset includes known input-output pairs, and the goal is to make the model output match the target values as closely as possible.

### 🔹 What Is Backpropagation?

**Backpropagation** is the learning mechanism that allows a neural network to adjust its internal parameters — the **weights** and **biases** — to improve its predictions. It is applied during the training phase under **supervised learning**, where each input has a known, corresponding label (also called **ground truth**).

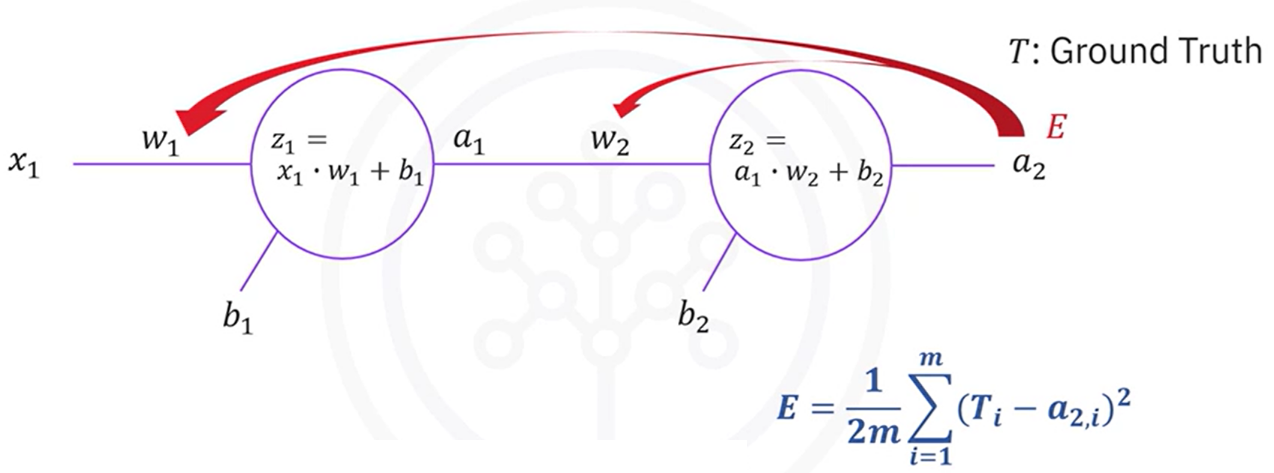
The purpose of backpropagation is to **minimize the error** between the network's predicted output and the actual target values by propagating that error backward through the network and using it to update each parameter in the model.

### 🔹 The Role of Backpropagation in Training

Training begins when the output generated by the neural network does **not match the ground truth** for a given input. This mismatch indicates that the model has made an error, and this error becomes the basis for learning.

The process starts with:

1. Performing a **forward pass**: The input moves through the network to produce a predicted output.
2. **Computing the error**: The difference between the predicted output and the actual label is measured using a cost or loss function.
3. Using the **error signal** to perform backpropagation: This step determines how each weight and bias contributed to the error and adjusts them accordingly using gradient descent.

****

The backpropagation algorithm applies the **chain rule of calculus** to compute the **partial derivatives** of the error with respect to each parameter in the network. These derivatives (gradients) tell us how sensitive the error is to changes in each weight or bias.

Because weights and biases affect the error **indirectly** through several intermediate computations (activations, pre-activations, etc.), their gradients are **not directly observable**. Instead, backpropagation breaks the entire derivative into smaller, local derivatives that can be chained together.

For Example:

**Updating w2:**

To update w2 we use the error

The cost E is **not directly a function of w2**, so, we compute:

Each term:

**Updating b2:**

It’s like w2, but the derivative is respective to b2.

**Updating w1:**

This requires deeper chaining because w1 **influences the error indirectly**:

Since E is not directly a function of w1, we must use chain rule to obtain the derivative of E respect w1.

Each term:

**Updating b1:**

It’s like w1, but the derivative is respective to b1.

Thus, a longer chain of dependencies is built and applied to calculate the gradient.

Each of these local derivatives is **multiplied** together in the correct order to compute the total gradient with respect to each parameter.

For parameters **closer to the output layer**, the gradients involve fewer steps.

For parameters **closer to the input layer**, more intermediate computations are involved:

* Output error → output activation → output pre-activation → hidden activation → hidden pre-activation → weight

The more distant a parameter is from the output; the more layers must be traversed backward to determine how it influenced the final output. Backpropagation systematically handles this traversal using the chain rule, regardless of how many layers are involved.

#### ⚙️ Parameter Updates

Once the gradient of the error is computed for a particular parameter (e.g., a weight or bias), it is updated using **gradient descent**:

* The parameter is moved **in the opposite direction of the gradient**, which is the direction that reduces the error.
* The **learning rate** controls how large each step is.
* This is done **simultaneously for all parameters** in the network.

#### 🔄 Repeating the Process

Backpropagation is performed **repeatedly**, across:

* Many **training samples**
* Multiple **epochs** (full passes through the dataset)

At each iteration:

1. A forward pass is computed for the current input.
2. The error is calculated by comparing the prediction with the true label.
3. Backpropagation is used to distribute this error backward.
4. Weights and biases are updated.
5. The process repeats with new inputs and updated parameters.

The goal is to continue this loop until:

* A predefined number of iterations (epochs) has been completed.
* The error becomes sufficiently small — below a defined **threshold**.

### 🔹 Takeaways

✅ **Backpropagation** is the algorithm that allows a neural network to learn from its mistakes.

✅ It works by using the **chain rule** to determine how each weight and bias influenced the final prediction error.

✅ These gradients are used to **update the parameters** using gradient descent.

✅ The process is repeated over many iterations, leading the network to converge toward a solution that minimizes the error.

## 📌 Vanishing Gradient

### 🔹 The Vanishing Gradient Problem: Overview

One of the most significant challenges in training deep neural networks—particularly before the rise of modern architectures—is the **vanishing gradient problem**. This issue, tied closely to certain activation functions such as the **sigmoid**, can severely slow down or even halt the learning process, especially in networks with many layers.

This problem is one of the reasons neural networks did not gain widespread success earlier, despite their theoretical potential.

In deep networks, the training process uses **backpropagation** to compute how the loss function responds to changes in each weight. This requires calculating **gradients** (partial derivatives) that are propagated backward through the network.

The problem becomes apparent when the network:

* Uses the **sigmoid activation function**
* Has **multiple layers**, especially more than two

The **sigmoid function**:

maps all inputs to the range (0, 1). As a result:

* Its output values are **bounded**, never becoming very large or very small
* The **derivative** of the sigmoid is also between 0 and 0.25
* Gradients calculated during backpropagation will also be **less than 1**

#### ⚠️ Why Sigmoid is Problematic

The **sigmoid function**, and similar bounded activation functions (like tanh), are especially prone to this issue:

* Their output is **bounded** between 0 and 1
* Their gradients are **less than or equal to 0.25**
* Repeated application in deep networks leads to exponentially vanishing gradients

While the sigmoid was historically popular due to its smoothness and probabilistic output interpretation, its use in deep networks is now discouraged for this reason.

### 🔹 How Gradients Disappear

During backpropagation, gradients are computed using the **chain rule**, meaning the gradient at a given layer depends on the product of all gradients in the layers that follow it.

When these gradients are each **less than 1**, repeated multiplication causes them to **shrink exponentially** as they propagate backward through the network. This means:

* Gradients in the **last layers** (closer to the output) may still have meaningful values
* But gradients in the **early layers** (closer to the input) become **extremely small**
* This leads to **very slow updates** for those weights during training

As a result:

* Early-layer neurons learn **very slowly**
* Training becomes **inefficient or unstable**
* Final prediction performance suffers due to **undertrained lower layers**

#### ⚠️ Consequences in Training

**🔸 Learning Imbalance**

* Later layers continue to update and learn features
* Earlier layers remain nearly static due to minuscule gradients
* The model may overfit shallow patterns while ignoring deeper structure

**🔸 Slower Convergence**

* More iterations (epochs) are needed to reach reasonable performance
* Optimization becomes inefficient, especially in large networks

**🔸 Compromised Accuracy**

* Incomplete learning in earlier layers leads to suboptimal representations
* This directly impacts the model’s ability to generalize on unseen data

### 🔹 Takeaways

✅ The **vanishing gradient problem** occurs when gradients become very small as they are propagated backward through a deep neural network.

✅ This problem is primarily caused by **activation functions like sigmoid**, whose derivatives are always less than 1.

✅ As a result, **earlier layers learn very slowly**, making training inefficient and hurting model accuracy.

✅ To avoid this issue, modern neural networks typically use **activation functions like ReLU**, which do not suffer from vanishing gradients in the same way.

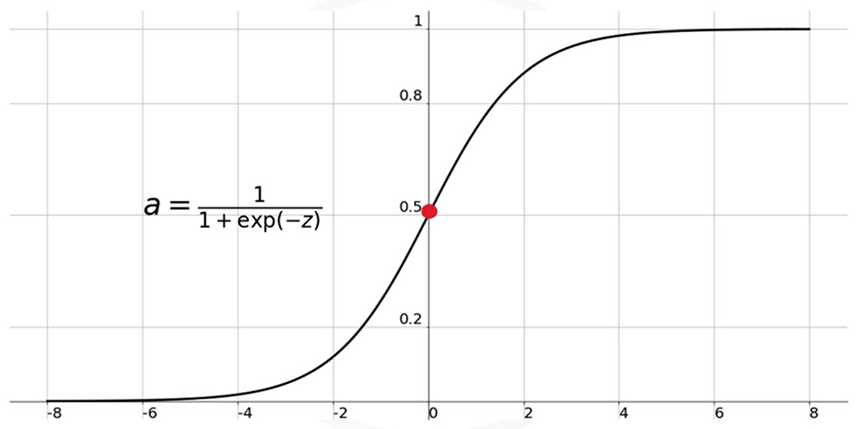
## 📌 Activation Functions

### 🔹 Role of Activation Functions in Neural Networks

**Activation functions** are essential components in neural networks. They introduce **non-linearity** into the model, enabling it to learn complex patterns, capture interactions between inputs, and represent highly flexible decision boundaries. Without activation functions, neural networks would reduce to a series of linear operations and would not be able to learn or approximate non-linear mappings.

In addition to their functional role in transforming inputs, activation functions influence **training behavior**, **gradient flow**, and **computational efficiency**. The choice of activation function can significantly affect model convergence, depth, and generalization.

### 🔹 Limitations of the Sigmoid Function

The **Sigmoid (logistic)** function was historically used in neural networks due to its smooth gradient and interpretable output.

It maps all input values to a range between **0 and 1**, with:

Despite its early popularity, the Sigmoid function has significant drawbacks:

**🔸 Vanishing Gradient Issue**

* The function **saturates** for large positive or negative inputs.
* In these saturation zones (e.g., beyond ±3), the derivative becomes **extremely small**, often near zero.
* During backpropagation, gradients get multiplied through the layers. When these values are small, the **gradients diminish** rapidly as they move backward through the network.
* This leads to the **vanishing gradient problem**, where **earlier layers stop learning** effectively.

**🔸 Lack of Symmetry**

* Sigmoid outputs are always **positive**.
* This causes all subsequent layers to receive inputs of the same sign, potentially **biasing the network’s learning** and slowing convergence.

### 🔹 Commonly Used Activation Functions

#### 🔧 Sigmoid Function

**ℹ️ Range:** (0,1)

**💡 Advantages:**

* + Historically popular due to smooth, bounded output.

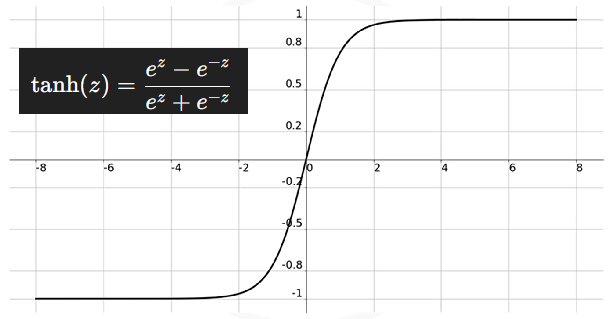
**⚠️ Limitations:**

* + **Vanishing gradient problem**: The function flattens out for large inputs, producing gradients close to zero. This slows or stops learning in earlier layers.
  + **Non-zero-centered output:** All values are positive, which can bias neuron activations and hinder learning.
  + **Poor performance** in deep networks.

**✅ Usage:**

* + Rarely used in hidden layers today due to these issues.

#### 🔧 Hyperbolic Tangent (Tanh) Function

**ℹ️ Range:** (-1,1)

**💡 Advantages:**

* + **Symmetric about the origin,** addressing the imbalance issue of sigmoid.
  + Slightly better than sigmoid in terms of convergence.

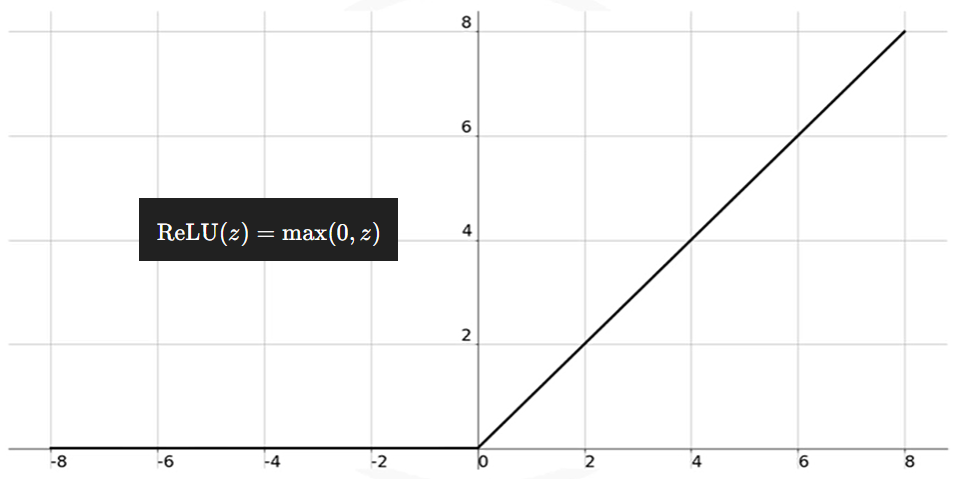
**⚠️ Limitations:**

* + Still suffers from the **vanishing gradient problem**, especially in deep networks.
  + Poor scalability in deep learning when used in hidden layers.

**✅ Usage:**

* + Occasionally used in shallow networks, but largely replaced by ReLU in modern practice.

#### 🔧 Hyperbolic Tangent (Tanh) Function

**ℹ️ Range:** {0,∞)

**💡 Advantages:**

* + **Avoids vanishing gradients** for positive inputs.
  + Enables **sparse activations**, as only neurons with positive input values activate.
  + **Computationally efficient** and simple to implement.
  + Enabled deep learning advancements due to its robustness.

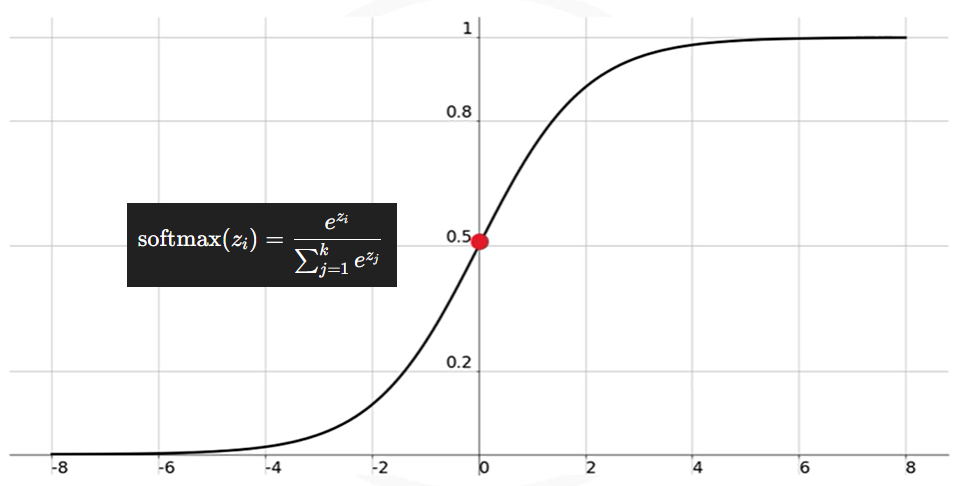
**⚠️ Limitations:**

* + **Inactive neurons**: If inputs are negative, the neuron is not activated (output is 0). This can lead to some neurons never updating (known as the **"dying ReLU"** problem).

**✅ Usage:**

* + Default choice for hidden layers in modern deep networks.
  + Not used in the output layer.

#### 🔧 Softmax Function

**ℹ️ Range:** (0, 1), and all outputs sum to 1.

**💡 Advantages:**

* + Converts raw output scores into **probability distributions** over classes.
  + Helps **classify inputs in multi-class** problems.

**⚠️ Limitations:**

* + Computationally heavier than ReLU or sigmoid.
  + Not suitable for hidden layers; limited to output layer in classification.

**✅ Usage:**

* + Used **exclusively in the output layer** of classifiers where multi-class probability prediction is needed.

#### 🔧 Overview of Activation Function Types

| **Activation Function** | **Output Range** | **Common Usage** | **Notes** |
| --- | --- | --- | --- |
| Binary Step | {0, 1} | Simple thresholding | Not used in practice due to lack of gradient |
| Linear (Identity) | −∞,∞ | Regression models | No non-linearity; unsuitable for deep learning |
| Sigmoid | (0, 1) | Historically in hidden layers | Leads to vanishing gradients and slow convergence |
| Tanh | (-1, 1) | Hidden layers (less common now) | Symmetric over the origin; still suffers from vanishing gradients |
| ReLU | [0, ∞) | Hidden layers (standard today) | Sparse activation; efficient; avoids vanishing gradients |
| Leaky ReLU | (-∞, ∞) | Hidden layers | Variant of ReLU; avoids dead neurons |
| Softmax | (0, 1), sums to 1 | Output layer of classifiers | Outputs probabilities for multi-class problems |

### 🔹 Practical Guidelines for Activation Function Choice

**🧠 Avoid** using **sigmoid** and **tanh** in hidden layers of deep networks due to vanishing gradient problems.

**🧠** Use **ReLU** as the **default** activation function in hidden layers.

**🧠** Use **softmax** in the **output layer** of multi-class classifiers.

**🧠** If ReLU yields suboptimal performance, consider variants like **Leaky ReLU** or explore other activation strategies (e.g., Swish, GELU — not covered here).

### 🔹 Takeaways

✅ Activation functions are central to a network’s ability to **learn non-linear patterns**.

✅ The **choice of activation function** affects training speed, gradient stability, and model accuracy.

✅ The **sigmoid and tanh** functions are mostly avoided in deep architectures due to their tendency to produce **vanishing gradients**.

✅ **ReLU** has become the dominant choice for hidden layers, thanks to its simplicity and effectiveness.

✅ **Softmax** is used in the output layer when the model must produce **probabilistic class predictions**.