Natural Language Processing
NLP_CLT_1st_May_11th_2025

Eng. Maytham Ghanoum
Artificial Intelligence & Deep Learning Specialist
MTN Syria – SCS – SVU CLT
+963947222064 - +963982018359
https://www.linkedin.com/in/maytham-ghanoum-697aa5207/
https://www.facebook.com/maytham.ghanoum

Welcome

What is Deep Learning?

Artificial Intelligence

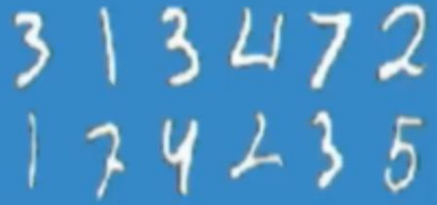
Any technique that enables computers to mimic human behavior

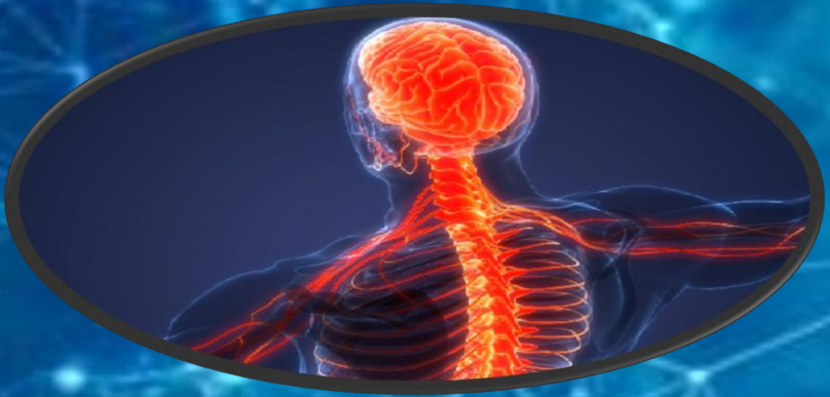
Machine Learning

Ability to learn without explicitly being programmed

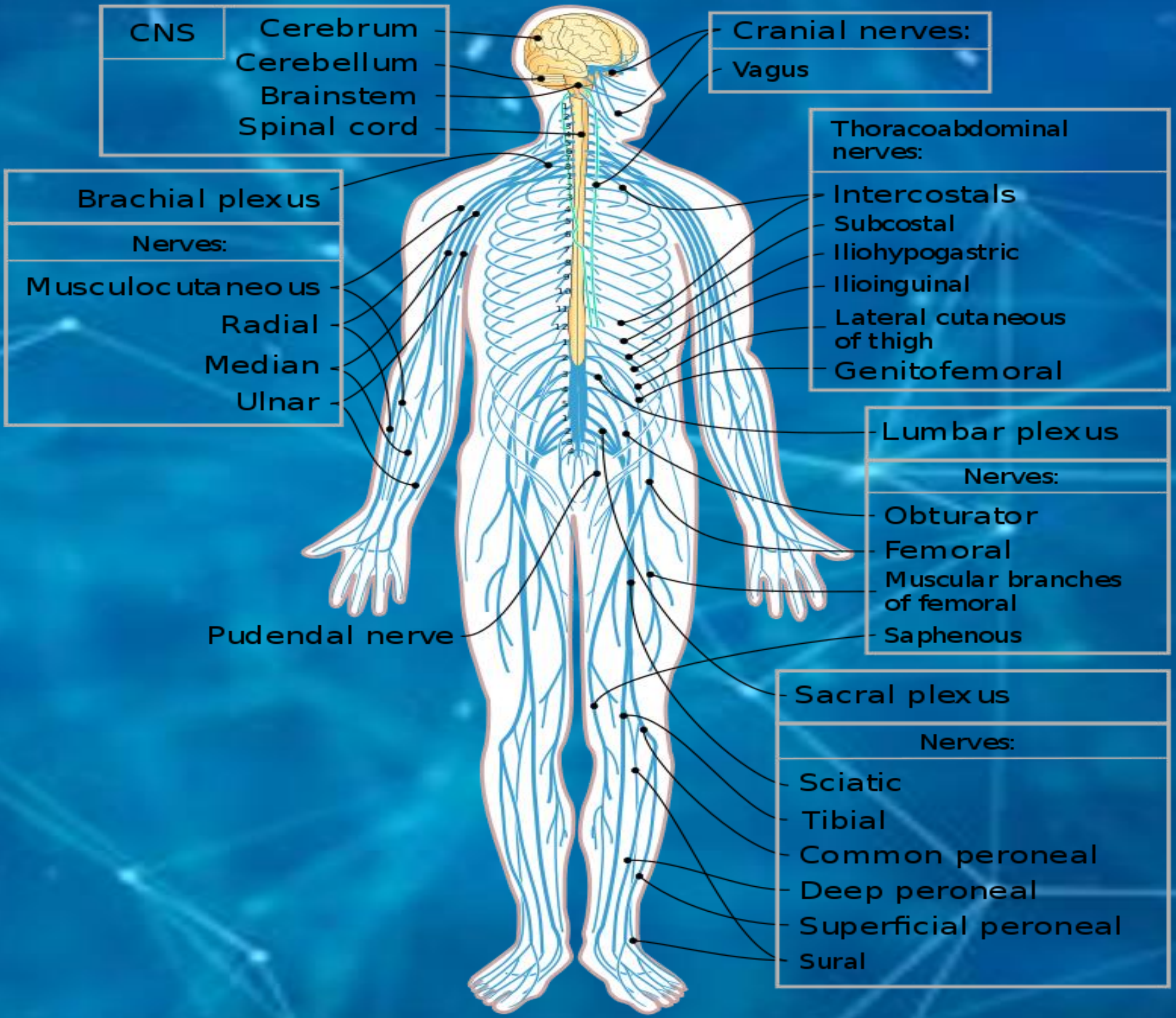
Deep Learning

Extract patterns from data using neural networks

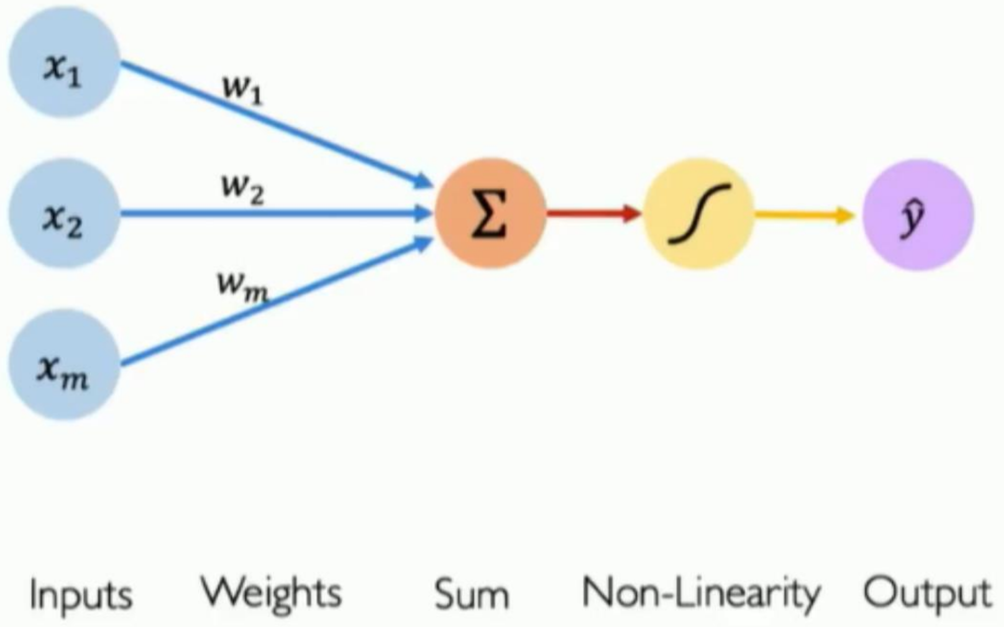




HUMAN NERVOUS SYSTEM!!



The Perceptron: Forward Propagation



Output

Linear combination of inputs

$$\hat{y} = g\left(\sum_{i=1}^{m} x_i \, w_i \right)$$

Non-linear activation function

Forward Pass in a Perceptron

A perceptron is the simplest form of an artificial neural network. It mimics the way biological neurons process inputs and generate outputs. Let's go through the entire process mathematically, from inputs to output.

1. Inputs and Weights

The perceptron takes multiple inputs, represented as a vector X.
Each input has an associated weight, represented as a vector W.
Mathematically, if we have n inputs, we define:

$$X = [x_1, x_2, x_3, \ldots\ldots, x_n]$$
$$W = [w_1, w_2, w_3, \ldots\ldots, w_n]$$

X: represents each input feature.
W: represents the weight associated with each input.

The perceptron also includes a **bias** term b, which helps the model shift the decision boundary.

Summation Function (Weighted Sum)

The perceptron computes a weighted sum of inputs:

$$Z = \sum_{i=1}^{n} x_i w_i + b$$

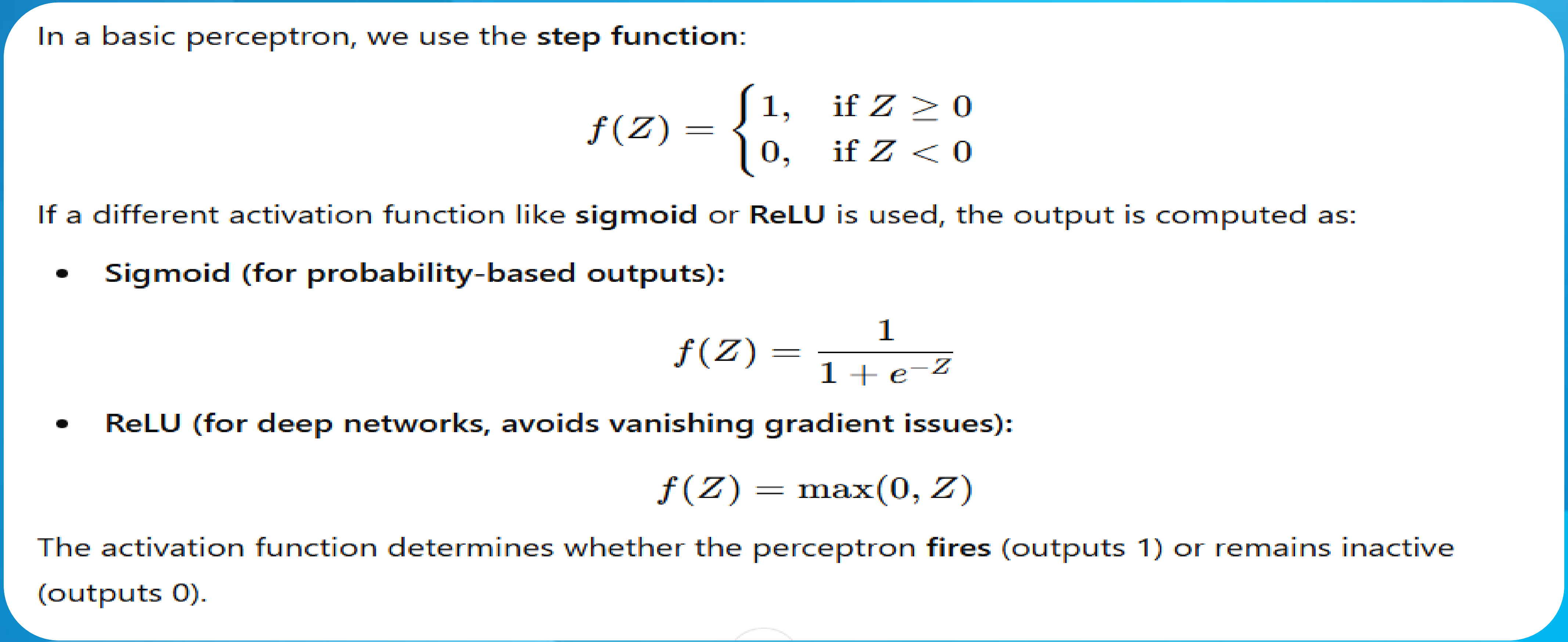
This operation is a **dot product** between the input vector X and the weight vector W, plus the bias *b*.

Activation Function

The perceptron applies an activation function to decide the output.

Activation Function

The perceptron applies an activation function to decide the output.



In a basic perceptron, we use the **step function**:

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

If a different activation function like **sigmoid** or **ReLU** is used, the output is computed as:

- **Sigmoid (for probability-based outputs):**

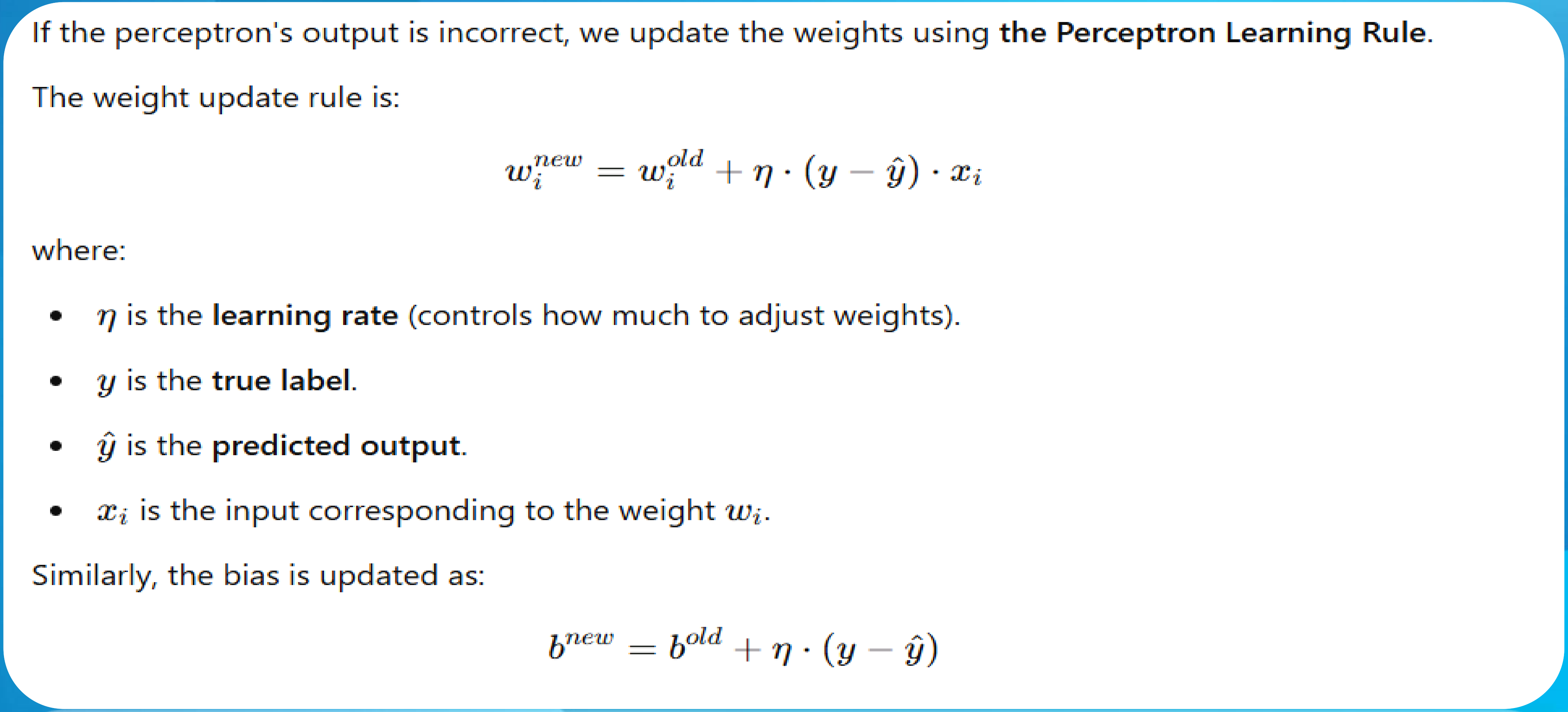
$$f(Z) = \frac{1}{1 + e^{-Z}}$$

- **ReLU (for deep networks, avoids vanishing gradient issues):**

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

The activation function determines whether the perceptron **fires** (outputs 1) or remains inactive (outputs 0).

Weight Adjustment (Learning)



If the perceptron's output is incorrect, we update the weights using **the Perceptron Learning Rule**.

The weight update rule is:

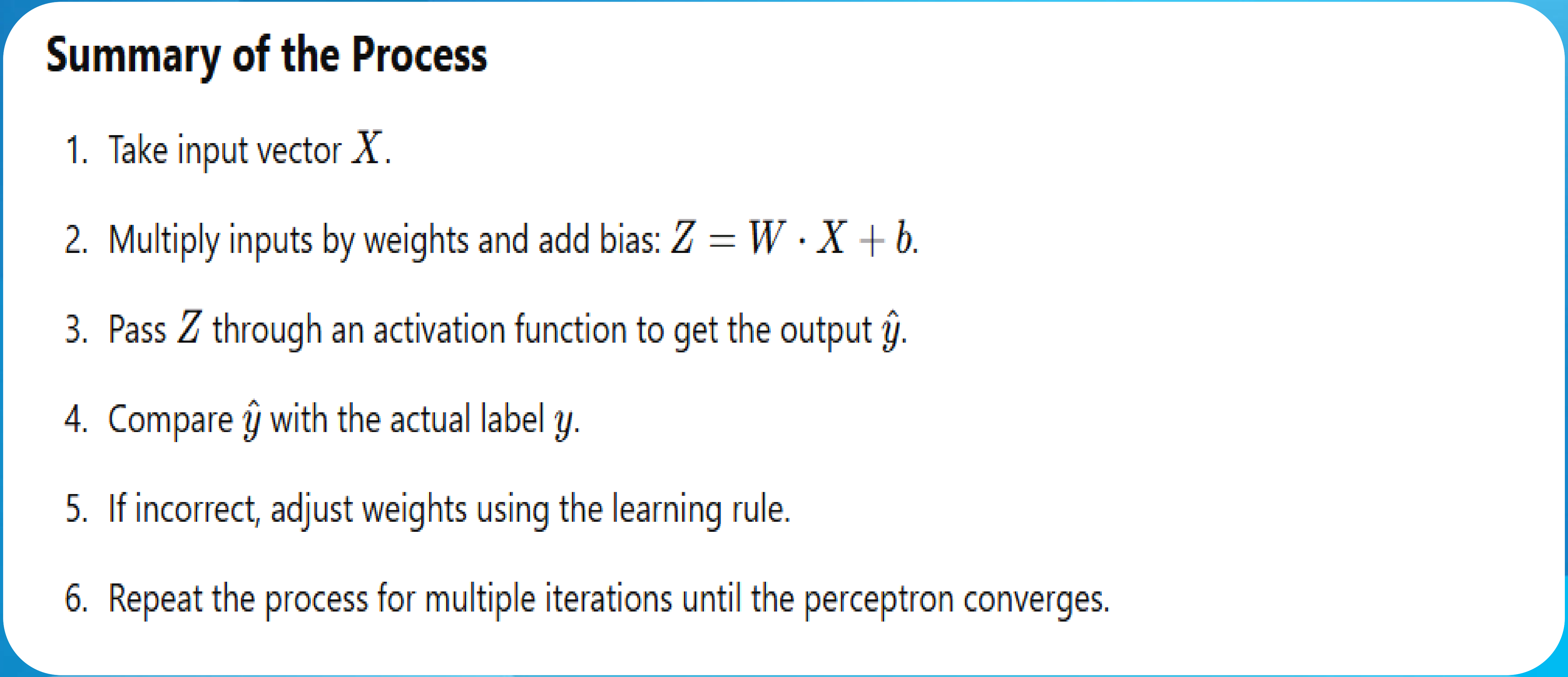
$$w_i^{new} = w_i^{old} + \eta \cdot (y - \hat{y}) \cdot x_i$$

where:

- η is the **learning rate** (controls how much to adjust weights).
- y is the **true label**.
- $\hat{y}$ is the **predicted output**.
- x_i is the input corresponding to the weight w_i.

Similarly, the bias is updated as:

$$b^{new} = b^{old} + \eta \cdot (y - \hat{y})$$



Summary of the Process

1. Take input vector X.
2. Multiply inputs by weights and add bias: $Z = W \cdot X + b$.
3. Pass Z through an activation function to get the output $\hat{y}$.
4. Compare $\hat{y}$ with the actual label y.
5. If incorrect, adjust weights using the learning rule.
6. Repeat the process for multiple iterations until the perceptron converges.

Mathematical Example

We will implement a perceptron to solve a simple **binary classification** problem. Let's say we have a dataset with two features, and we want to classify points into two classes (0 or 1).

Dataset

x_1	x_2	Label (y)
0	0	0
0	1	0
1	0	0
1	1	1

This is the **AND logic gate**, where the output is 1 **only when both inputs are 1**.

Step 1: Initialize Weights and Bias

We randomly initialize the weights and bias.

- Let's assume $w_1 = 0.5$, $w_2 = 0.5$, and $b = -0.7$.

Step 2: Compute the Weighted Sum (Linear Combination)

For each input $X = (x_1, x_2)$, compute:

$$Z = x_1w_1 + x_2w_2 + b$$

Step 3: Apply Activation Function

Using the step function:

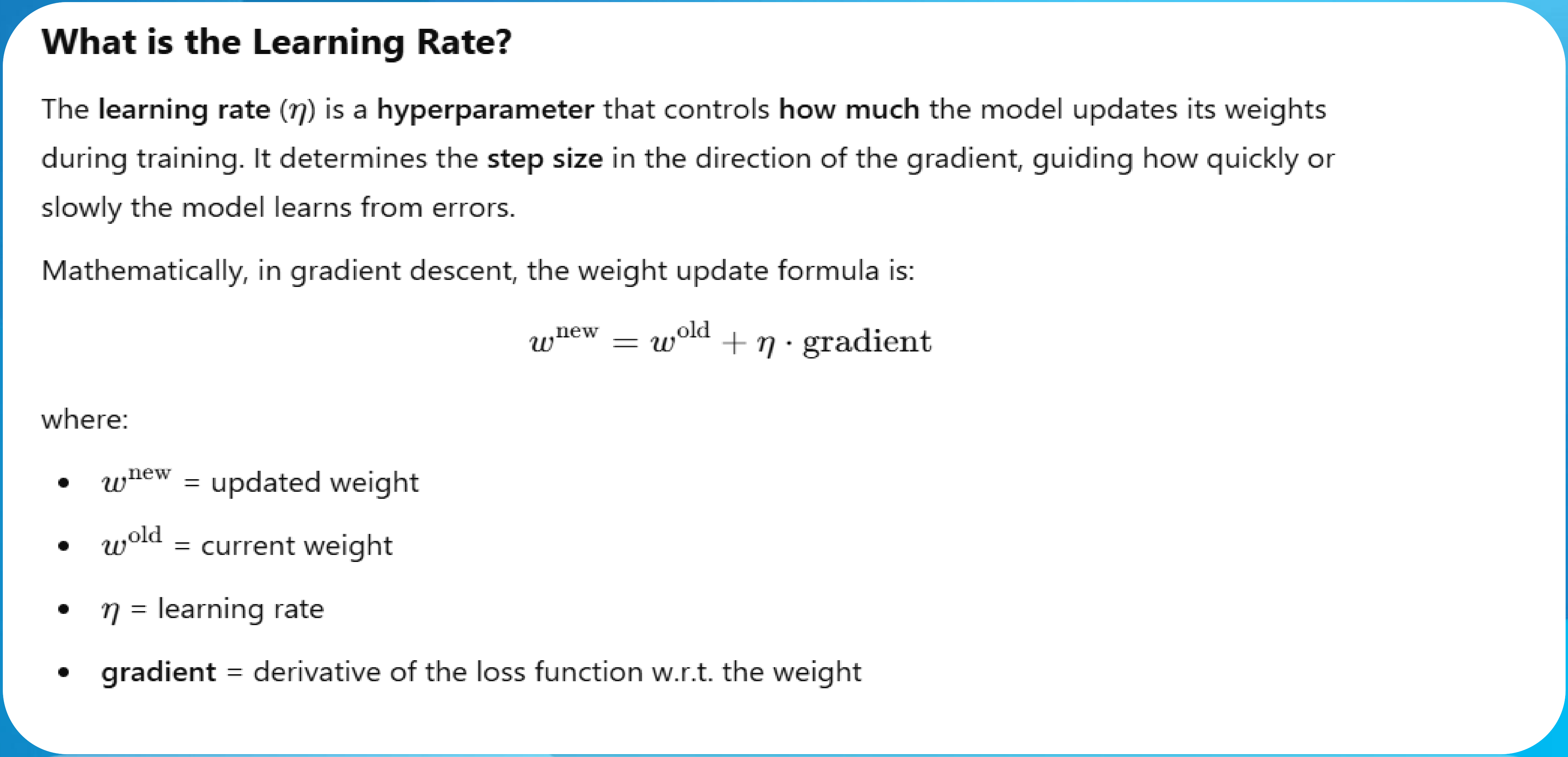
$$f(Z) = \begin{cases} 1, & \text{if } Z \geq 0 \\ 0, & \text{if } Z < 0 \end{cases}$$

Step 4: Update Weights if the Prediction is Wrong

If $\hat{y} \neq y$, update the weights using:

$$w_i^{new} = w_i^{old} + \eta(y - \hat{y})x_i$$

where η is the learning rate.



What is the Learning Rate?

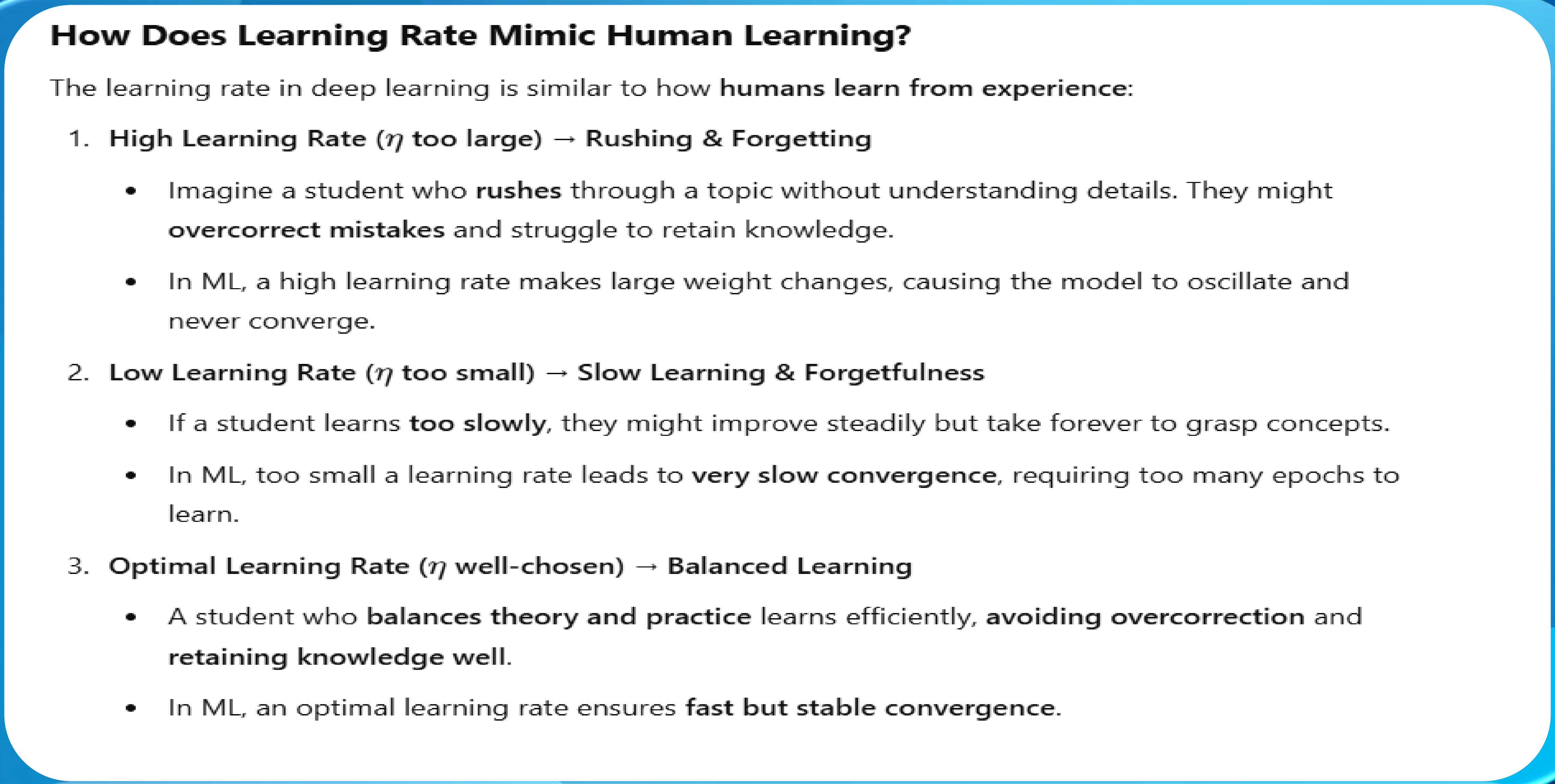
The **learning rate** (η) is a **hyperparameter** that controls **how much** the model updates its weights during training. It determines the **step size** in the direction of the gradient, guiding how quickly or slowly the model learns from errors.

Mathematically, in gradient descent, the weight update formula is:

$$w^{\text{new}} = w^{\text{old}} + \eta \cdot \text{gradient}$$

where:

- w^{new} = updated weight
- w^{old} = current weight
- η = learning rate
- **gradient** = derivative of the loss function w.r.t. the weight



How Does Learning Rate Mimic Human Learning?

The learning rate in deep learning is similar to how **humans learn from experience**:

1. **High Learning Rate (η too large) → Rushing & Forgetting**
   - Imagine a student who **rushes** through a topic without understanding details. They might **overcorrect mistakes** and struggle to retain knowledge.
   - In ML, a high learning rate makes large weight changes, causing the model to oscillate and never converge.
2. **Low Learning Rate (η too small) → Slow Learning & Forgetfulness**
   - If a student learns **too slowly**, they might improve steadily but take forever to grasp concepts.
   - In ML, too small a learning rate leads to **very slow convergence**, requiring too many epochs to learn.
3. **Optimal Learning Rate (η well-chosen) → Balanced Learning**
   - A student who **balances theory and practice** learns efficiently, **avoiding overcorrection** and **retaining knowledge well**.
   - In ML, an optimal learning rate ensures **fast but stable convergence**.

How to Choose the Learning Rate?

1. **Experimentation (Trial & Error)**
   - Start with a moderate value (e.g., **0.01 or 0.001**).
   - Increase if the model is learning too slowly.
   - Decrease if the loss fluctuates wildly.
2. **Using a Learning Rate Scheduler**
   - **Adaptive methods** (e.g., **Adam, RMSprop**) adjust the learning rate automatically.
   - **Step decay:** Reduce learning rate at fixed epochs.
   - **Exponential decay:** Reduce learning rate gradually over time.
3. **Using Learning Rate Range Tests**
   - Train the model with increasing learning rates and **find the best range** (e.g., **Learning Rate Finder**).

Using the Sigmoid Activation Function in the Perceptron

In the original example, we used the **step function** as the activation function:

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

Now, we will replace it with the **sigmoid activation function**, which is smoother and differentiable, making it useful for gradient-based optimization:

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

This function **squashes the output** to a value between 0 and 1, making it suitable for **binary classification**.

Mathematical Breakdown with Sigmoid Activation

Let's apply the **sigmoid function** to the perceptron and perform a forward pass.

1. **Compute the weighted sum (linear transformation):**

$$Z = x_1 w_1 + x_2 w_2 + b$$

2. **Apply the sigmoid activation function:**

$$y_{\text{pred}} = \sigma(Z) = \frac{1}{1 + e^{-Z}}$$

3. **Compute the error:**

$$\text{error} = y - y_{\text{pred}}$$

4. **Update the weights using gradient descent:**

$$w_i^{\text{new}} = w_i^{\text{old}} + \eta \cdot \text{error} \cdot x_i \cdot \sigma(Z) \cdot (1 - \sigma(Z))$$

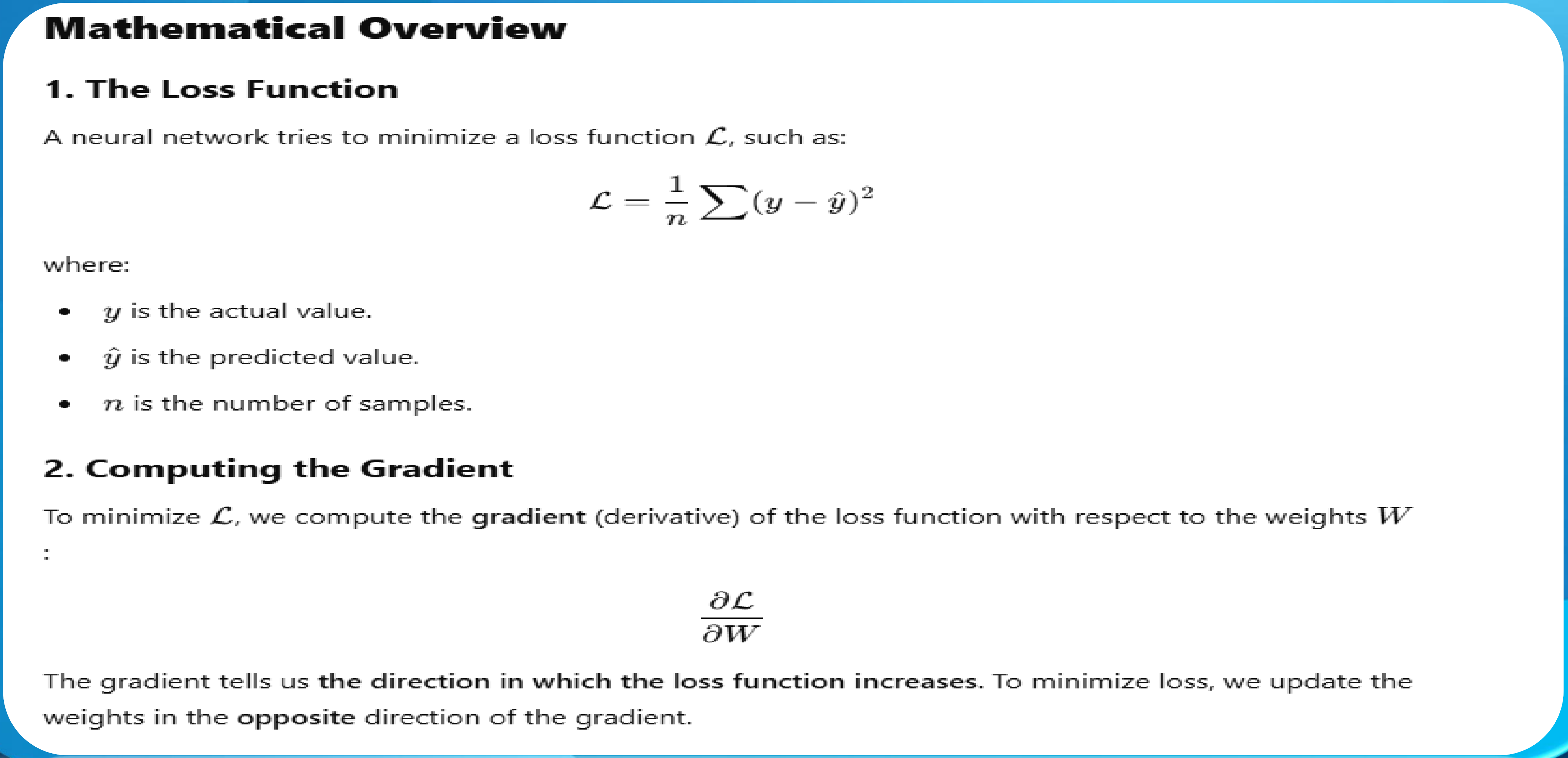
where η is the **learning rate**, and $\sigma(Z) \cdot (1 - \sigma(Z))$ is the derivative of the sigmoid function.

Gradient Descent: The Core of Learning in Deep Learning

What is Gradient Descent?

Gradient Descent is an **optimization algorithm** used to **minimize the loss function** by iteratively updating the model's parameters (weights and biases). It helps the neural network learn by adjusting weights in the direction that **reduces the error**.

Imagine you're hiking down a mountain in the fog—you take small steps downward to reach the lowest point (the global minimum of the loss function).



Mathematical Overview

1. The Loss Function

A neural network tries to minimize a loss function $\mathcal{L}$, such as:

$$\mathcal{L} = \frac{1}{n} \sum (y - \hat{y})^2$$

where:

- y is the actual value.
- $\hat{y}$ is the predicted value.
- n is the number of samples.

2. Computing the Gradient

To minimize $\mathcal{L}$, we compute the **gradient** (derivative) of the loss function with respect to the weights W :

$$\frac{\partial \mathcal{L}}{\partial W}$$

The gradient tells us **the direction in which the loss function increases**. To minimize loss, we update the weights in the **opposite** direction of the gradient.

3. Weight Update Rule

$$W^{\text{new}} = W^{\text{old}} - \eta \cdot \frac{\partial \mathcal{L}}{\partial W}$$

where:

- η (learning rate) controls the step size.
- $\frac{\partial \mathcal{L}}{\partial W}$ is the gradient.

4. The Importance of Learning Rate (η)

- **Too large**: We overshoot and never converge.
- **Too small**: The process is too slow.
- **Optimal**: Helps reach the minimum efficiently.

Gradient Descent Variants

1. Batch Gradient Descent

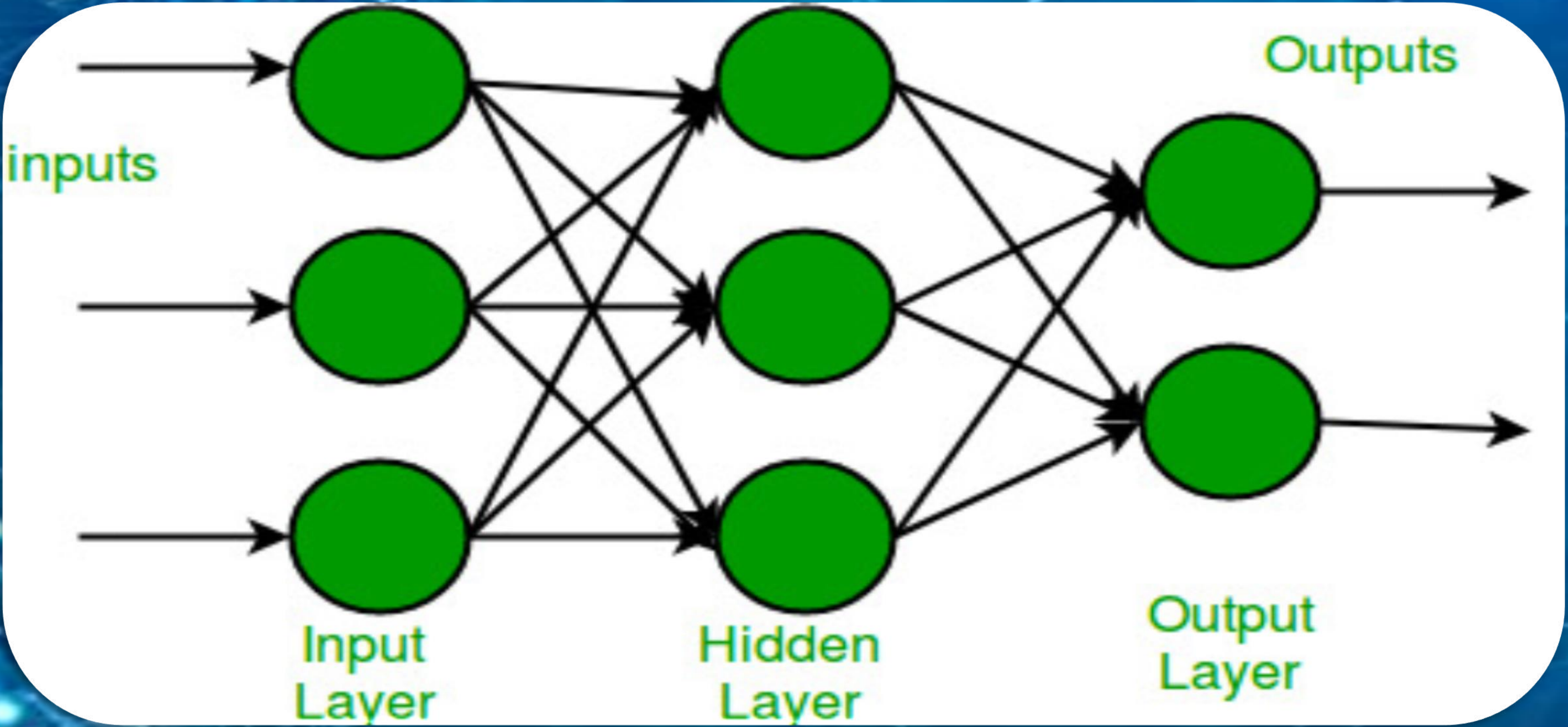
- Uses the **entire dataset** to compute the gradient.
- Slow for large datasets.

2. Stochastic Gradient Descent (SGD)

- Uses **one sample** at a time to compute gradients.
- Faster but noisier.

3. Mini-Batch Gradient Descent

- Uses **a small batch** of samples at each step.
- Balances efficiency and stability.


Outputs
inputs
Input Layer
Hidden Layer
Output Layer

Multi-Layer Perceptron (MLP) with Backpropagation

A **Multi-Layer Perceptron (MLP)** is an artificial neural network composed of multiple layers of neurons:

1. **Input Layer** (accepts data)
2. **Hidden Layers** (extracts features and patterns)
3. **Output Layer** (produces predictions)

Each neuron performs:

$$z = W \cdot X + b$$

where:

- W = weights
- X = input features
- b = bias
- z = linear combination of weights and inputs
- Activation function $f(z)$ introduces **non-linearity**

Why Use Backpropagation?

Backpropagation is the learning algorithm that allows MLP to **update weights** efficiently.
It **minimizes the error** between predicted output and actual labels by adjusting weights **using gradient descent.**

Backpropagation Steps

1. **Forward Pass:** Compute output of each neuron layer-by-layer.
2. **Loss Calculation:** Compare predicted and actual output using a loss function.
3. **Backward Pass:** Compute gradients of the loss w.r.t. weights using the chain rule.
4. **Weight Update:** Adjust weights using **gradient descent**.

Adam Optimizer: The King of Optimization

Why Use Adam?

Adam (Adaptive Moment Estimation) is a more advanced optimization algorithm that combines the advantages of:

- **Momentum** (smooths updates)
- **RMSprop** (adapts learning rates for different parameters)

Mathematical Breakdown

1. **Compute the gradient g_t at time t:**

$$g_t = \frac{\partial \mathcal{L}}{\partial W_t}$$

2. **Compute moving averages:**
   - **First moment estimate** (Momentum):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

   - **Second moment estimate** (RMSprop effect):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

3. **Bias correction:**

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

4. **Update weights:**

$$W_t = W_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

where:

- $\beta_1 = 0.9$, $\beta_2 = 0.999$ (default values)
- $\epsilon = 10^{-8}$ (to avoid division by zero)
- η (learning rate) is adaptive.