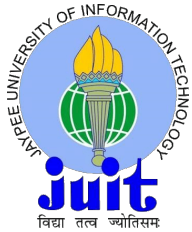


Machine Learning

(Course Code: 18B1WCI634/18B11BI611)

Artificial Neural Network

Mr. Sandeep Kumar Patel

Assistant Professor

Jaypee University of Information Technology

Waknaghat, Solan, HP-173234

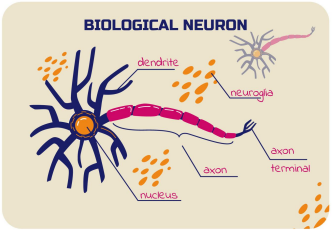
Human Brain Analogy

- 1. The human brain has $\sim 10^{11}$ neurons, each connected to $\sim 10^4$ others.
- 2. Neurons are excited or inhibited via connections.
- 3. Fastest neuron switching time: $\sim 10^{-3}$ seconds (much slower than computers: $\sim 10^{-10}$ s).
- 4. Despite this, humans make complex decisions rapidly, e.g., recognizing a face in $\sim 10^{-1}$ seconds.
- 5. Suggests highly parallel processing with distributed representations.

Biological Motivation for ANNs

- 1. Artificial neural networks (ANNs) are inspired by biological learning systems.
- 2. Biological systems consist of complex, densely interconnected neurons.
- 3. ANNs mimic this by connecting simple computational units, each taking multiple real-valued inputs and producing a single real-valued output.

Biological Neuron → Inspiration for ANN



How does the brain work?

- The brain is made up of **billions of neurons**
- Each neuron performs three simple steps:
  - ▷ **Receives signals** (via dendrites)
  - ▷ **Processes information**
  - ▷ **Sends output** (via axon)

Key Idea: *Simple units → Complex intelligence*
Bridge to AI:

We mimic this using mathematical models

⇒ Artificial Neurons (ANN)

Artificial Neural Networks (ANN)

Definition

Artificial Neural Network (ANN) is a computational model inspired by the human brain, used for pattern recognition, classification, and prediction.

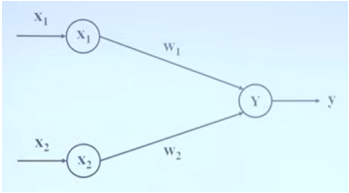
- -Consists of interconnected nodes called **neurons**, organized in layers.
- -Commonly used for predictive analytics, anomaly detection, and decision-making.
-

Perceptron: Basic Concept

Perceptron

It is a simple computational model that mimics a biological neuron.

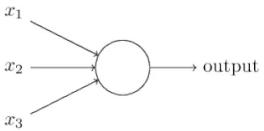
- -It takes several binary inputs $(x_1, x_2, ..., x_n)$ and produces a single binary output.
- -Each input is assigned a **weight** $(w_1, w_2, ..., w_n)$ representing its importance.
- -The perceptron computes a **weighted sum** and compares it with a **threshold**.
- -If the weighted sum exceeds the threshold $\Rightarrow$ output $= 1$, else output $= 0$.



Perceptron with 3 Inputs

- -Example: 3-input perceptron with weights w_1, w_2, w_3 and inputs x_1, x_2, x_3.
- -Weighted sum $= w_1x_1 + w_2x_2 + w_3x_3$.
- -If $\sum w_i x_i >$ threshold $\rightarrow$ output 1, else 0.
- -**Biasing:** Introduced to make activation easier or harder.

$$\text{Output: } y = \begin{cases} 1, & \text{if } w \cdot x + b > 0 \\ 0, & \text{otherwise} \end{cases}$$



PERCEPTRON WITH 3 INPUTS

Artificial Neuron – Working

An Artificial Neuron Performs Three Steps:

1. Receives Inputs

$$x_1,\ x_2,\ x_3,\ \ldots$$

Inputs are features (data)

2. Applies Weights and Bias

$$w_1x_1 + w_2x_2 + w_3x_3 + \cdots + b$$

$w_1, w_2, w_3 =$ importance, $\quad b =$ bias

3. Applies Activation Function

$$y = f(w_1x_1 + w_2x_2 + \cdots + b)$$

Key Idea:

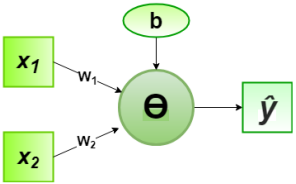
$$\text{Input} \rightarrow \text{Weighted Sum} \rightarrow \text{Output}$$

Perceptron for Boolean AND Function

- 1. Goal: Implement $Y = A$ AND B using a perceptron.
- 2. Perceptron output formula:

$$y = \begin{cases} 1 & \text{if } w_1 A + w_2 B + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- 3. Inputs: $A, B \in \{0, 1\}$



Step 1: Choose Weights and Bias

- To implement AND:
  - Only output 1 when $A = 1$ and $B = 1$
  - Assign weights: $w_1 = 1$, $w_2 = 1$
  - Bias: $b = -1.5$
- Weighted sum formula:

$$z = w_1 A + w_2 B + b = 1 \cdot A + 1 \cdot B - 1.5$$

- Output:

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

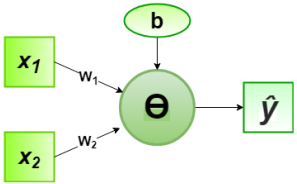
Step 2: Compute Output for All Inputs (Truth Table)

A	B	Weighted sum $z = w_1A + w_2B + b$	Output Y
0	0	$0 + 0 - 1.5 = -1.5$	0
0	1	$0 + 1 - 1.5 = -0.5$	0
1	0	$1 + 0 - 1.5 = -0.5$	0
1	1	$1 + 1 - 1.5 = 0.5$	1

Perceptron for Boolean OR Function

- Goal: Implement $Y = A$ OR B using a perceptron.
- Output formula:

$$y = \begin{cases} 1 & \text{if } w_1 A + w_2 B + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



Step 1: Choose Weights and Bias for OR

- Weights: $w_1 = 1,\ w_2 = 1$
- Bias: $b = -0.5$
- Weighted sum: $z = w_1 A + w_2 B + b = 1 \cdot A + 1 \cdot B - 0.5$
- Output:

$$y = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$

Step 2: Truth Table for OR Gate

A	B	Weighted sum $z = w_1A + w_2B + b$	Output Y
0	0	-0.5	0
0	1	0.5	1
1	0	0.5	1
1	1	1.5	1

Activation Function in ANN

Activation Function

It decides whether a neuron should be activated or not.

- -It introduces **non-linearity** into the neural network.
- -Without it, even multiple layers behave like a single linear model.

$$y = f(Wx + b)$$

- W – weights, x – inputs, b – bias
- $f(\cdot)$ – activation function

Why Activation Functions are Important

- -Enable the network to learn **complex, non-linear relationships**.
- -Help in **backpropagation** by providing differentiability.
- -Control the **range and behavior** of neuron outputs.

Types of Activation Functions

- **1. Linear Function:** $f(x) = x$
  - No non-linearity — used only in regression output layers.
- **2. Step Function:**

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

  - Used in perceptrons; not differentiable.

Linear Function Graph

$xf(x)$

Step Function Graph

$xf(x)$

Numerical 1: Linear Activation Function

Given: $x = [2, -3, 1]$, $W = [0.4, 0.2, -0.5]$, $b = 0.3$
Function: $f(x) = x$

$$z = W \cdot x + b = (0.4)(2) + (0.2)(-3) + (-0.5)(1) + 0.3$$

$$z = 0.8 - 0.6 - 0.5 + 0.3 = 0.0$$

$$\therefore f(z) = z = 0.0$$

Output: 0.0

Numerical 2: Step (Binary) Activation Function

Given: $x = 1.5$, threshold $= 0$

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

Since $1.5 \geq 0$, we have $f(1.5) = 1$.
Output: 1

Sigmoid and Tanh Functions

3. Sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}}, \quad \text{Range: } (0, 1)$$

4. tanh:

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \quad \text{Range: } (-1, 1)$$

Non- Linear: Sigmoid Function

$$xf(x)$$

Hyperbolic Tangent (tanh)

$xf(x)$

Numerical 3: Sigmoid Function

Function: $\sigma(x) = \dfrac{1}{1 + e^{-x}}$

$$\sigma(0) = 0.5, \quad \sigma(2) = 0.8808, \quad \sigma(-2) = 0.1192$$

Derivative Example:

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

$$\sigma'(2) = 0.8808(1 - 0.8808) = 0.10499$$

Output: $\sigma(2) = 0.8808$, $\sigma'(2) = 0.105$

Numerical 4: Hyperbolic Tangent (tanh)

Function: $\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$

$$\tanh(0.5) = 0.4621, \quad \tanh(2) = 0.9640$$

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

$$\tanh'(0.5) = 1 - (0.4621)^2 = 0.7864$$

Output: $\tanh(0.5) = 0.4621$, $\tanh'(0.5) = 0.7864$

ReLU and Leaky ReLU

5. ReLU:

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

6. Leaky ReLU:

$$f(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$$

ReLU Function

$$xf(x)$$

Leaky ReLU Function

$$xf(x)$$

Numerical 5: ReLU Function

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

$$f(-3) = 0, \quad f(0) = 0, \quad f(4) = 4$$

- Negative inputs are suppressed.
- Positive inputs remain unchanged.

Outputs: $0, 0, 4$

Numerical 6: Leaky ReLU Function

Function:

$$f(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$$

Let $\alpha = 0.01$

$$f(-5) = 0.01(-5) = -0.05, \quad f(3) = 3$$

Outputs: $f(-5) = -0.05,\ f(3) = 3$

check and if anything incorrect, correct it

Softmax Activation Function

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

- Converts raw outputs (logits) into probabilities.
- Used in the **output layer** for **multi-class classification**.
- Ensures all class probabilities sum to 1.

Numerical 7: Softmax Function

Given: $x = [2, 1, 0]$

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

$$e^x = [7.389, 2.718, 1], \quad \text{Sum} = 11.107$$

$$\text{Softmax} = [0.6652,\ 0.2447,\ 0.0900]$$

Outputs: $[0.665,\ 0.245,\ 0.090]$

Comparison Summary of Activation Functions

Function	Range	Non-linear	Use Case
Linear	$(-\infty, \infty)$	No	Regression output
Step	$\{0, 1\}$	Yes	Perceptron
Sigmoid	$(0, 1)$	Yes	Binary classification
tanh	$(-1, 1)$	Yes	Hidden layers
ReLU	$[0, \infty)$	Yes	Deep networks
Leaky ReLU	$(-\infty, \infty)$	Yes	Deep networks
Softmax	$(0, 1)$	Yes	Multi-class output

Perceptron Training Rule

Definition

A learning algorithm used to adjust the weights and bias of a perceptron so that it correctly classifies training examples.

Weight and Bias Update Formula

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

$$b^{\text{new}} = b^{\text{old}} + \eta\,(t - y)$$

Where:

- η : Learning rate, t : Target outpu, y : Predicted output, x_i : Input feature

Perceptron Training Algorithm

- 1. **Initialize** weights w_i and bias b to small random values.
- 2. **For each training example** $(\mathbf{x}, t)$:
  - a.Compute output:
    $$y = f(\mathbf{w} \cdot \mathbf{x} + b)$$
  - b. Compute error:
    $$e = t - y$$
  - c. Update weights:
    $$w_i \leftarrow w_i + \eta\, e\, x_i$$
  - d. Update bias:
    $$b \leftarrow b + \eta\, e$$
- 3. **Repeat** steps 2 until all examples are correctly classified .

Perceptron Training Example: OR Gate

OR Gate Truth Table

x_1	x_2	Target (t)
0	0	0
0	1	1
1	0	1
1	1	1

Perceptron Training Rule

$$w_i^{\text{new}} = w_i^{\text{old}} + \eta(t - y)x_i, \quad b^{\text{new}} = b^{\text{old}} + \eta(t - y)$$

- $y = f(w_1x_1 + w_2x_2 + b)$, step function: $f(z) = 1$ if $z \geq 0$, else 0
- $\eta =$ learning rate (e.g., 0.1)

Perceptron Training: Iterations 1 & 2

Iteration 1: Input (0,0), t=0

$$y = f(0 * 0 + 0 * 0 + 0) = f(0) = 1$$

Error: $e = t - y = 0 - 1 = -1$ Update:

$$w_1 = 0 + 0.1 * (-1 * 0) = 0, \quad w_2 = 0 + 0.1 * (-1 * 0) = 0, \quad b = 0 + 0.1 * (-1) = -0.1$$

Iteration 2: Input (0,1), t=1

$$y = f(0 * 0 + 0 * 1 + (-0.1)) = f(-0.1) = 0$$

Error: $e = 1 - 0 = 1$ Update:

$$w_1 = 0 + 0.1 * 1 * 0 = 0, \quad w_2 = 0 + 0.1 * 1 * 1 = 0.1, \quad b = -0.1 + 0.1 * 1 = 0$$

Perceptron Training: Iterations 3 & 4

Iteration 3: Input (1,0), t=1

$$y = f(0 * 1 + 0.1 * 0 + 0) = f(0) = 1$$

Error: $e = 1 - 1 = 0 \rightarrow$ No update

Iteration 4: Input (1,1), t=1

$$y = f(0 * 1 + 0.1 * 1 + 0) = f(0.1) = 1$$

Error: $e = 1 - 1 = 0 \rightarrow$ No update

Perceptron Training Result: OR Gate

Final Weights and Bias

$$\boxed{w_1 = 0, \quad w_2 = 0.1, \quad b = 0}$$

Conclusion

The perceptron successfully learns the OR gate after 4 iterations. - Input points are correctly classified. - Step function and weight updates ensure proper learning.

Perceptron Training Example: AND Gate

AND Gate Truth Table

x_1	x_2	Target (t)
0	0	0
0	1	0
1	0	0
1	1	1

Perceptron Training Rule

$$w_i^{\text{new}} = w_i^{\text{old}} + \eta(t - y)x_i, \quad b^{\text{new}} = b^{\text{old}} + \eta(t - y)$$

- $y = f(w_1x_1 + w_2x_2 + b)$, step function: $f(z) = 1$ if $z \geq 0$, else 0
- $\eta =$ learning rate (e.g., 0.1)

Perceptron Training: Iterations 1 & 2

Initial values: $w_1 = 0$, $w_2 = 0$, $b = 0$ **Iteration 1: Input (0,0), t=0**

$$y = f(0 * 0 + 0 * 0 + 0) = f(0) = 1$$

Error: $e = t - y = 0 - 1 = -1$ Update:

$$w_1 = 0 + 0.1 * (-1 * 0) = 0, \quad w_2 = 0 + 0.1 * (-1 * 0) = 0, \quad b = 0 + 0.1 * (-1) = -0.1$$

Iteration 2: Input (0,1), t=0

$$y = f(0 * 0 + 0 * 1 + (-0.1)) = f(-0.1) = 0$$

Error: $e = 0 - 0 = 0 \rightarrow$ No update

$$w_1 = 0, \quad w_2 = 0, \quad b = -0.1$$

Perceptron Training: Iterations 3 & 4

Iteration 3: Input (1,0), t=0

$$y = f(0 * 1 + 0 * 0 + (-0.1)) = f(-0.1) = 0$$

Error: $e = 0 - 0 = 0 \rightarrow$ No update

$$w_1 = 0, \quad w_2 = 0, \quad b = -0.1$$

Iteration 4: Input (1,1), t=1

$$y = f(0 * 1 + 0 * 1 + (-0.1)) = f(-0.1) = 0$$

Error: $e = 1 - 0 = 1$ Update:

$$w_1 = 0 + 0.1 * 1 * 1 = 0.1, \quad w_2 = 0 + 0.1 * 1 * 1 = 0.1, \quad b = -0.1 + 0.1 * 1 = 0$$

Perceptron Training Result: AND Gate

Final Weights and Bias

$$\boxed{w_1 = 0.1, \quad w_2 = 0.1, \quad b = 0}$$

Conclusion

The perceptron successfully learns the AND gate after 4 iterations. - Input points are correctly classified. - Step function and weight updates ensure proper learning.

Conclusion

- Activation functions add **non-linearity** to neural networks.
- Different functions suit different **tasks and layers**.
- ReLU and its variants are preferred in modern deep learning.

Key Idea: Choose an activation function that balances learning speed and performance.

Introduction

What is Gradient Descent?

- Gradient Descent is an **optimization algorithm** used to minimize a cost (or loss) function.
- It helps find the values of parameters (weights and bias) that reduce the prediction error.
- It is the foundation of training in machine learning and deep learning models.

Concept of Gradient Descent

- The cost function $J(w)$ measures how far the model's predictions are from the true values.
- The **gradient** $\frac{\partial J(w)}{\partial w}$ gives the direction of steepest increase of $J(w)$.
- To minimize the cost, we move in the opposite direction of the gradient.

$$w_{\text{new}} = w_{\text{old}} - \eta \frac{\partial J(w)}{\partial w}$$

Where:

- $\eta =$ learning rate (step size)
- $\frac{\partial J(w)}{\partial w} =$ gradient of cost function

How Gradient Descent Works

Algorithm Steps:

- Initialize parameters w and b (randomly or to zero).
- Compute predictions $\hat{y}$ using current parameters.
- Calculate cost $J(w, b)$.
- Compute gradients $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$.
- Update parameters:

$$w := w - \eta \frac{\partial J}{\partial w}, \quad b := b - \eta \frac{\partial J}{\partial b}$$

- Repeat until $J(w, b)$ converges (minimal change or threshold reached).

Mathematical Formulation

Example: Linear Regression Cost Function

$$J(w, b) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$$

where:

$$\hat{y}_i = wx_i + b$$

Compute Gradients:

$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)x_i$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)$$

Parameter Update Rules:

$$w := w - \eta\frac{\partial J}{\partial w}, \quad b := b - \eta\frac{\partial J}{\partial b}$$

Types of Gradient Descent

- **Batch Gradient Descent:** Uses all training samples to compute the gradient at once — stable but slow.
- **Stochastic Gradient Descent (SGD):** Updates parameters for each training example — fast but noisy.
- **Mini-Batch Gradient Descent:** Uses a small subset (batch) of data per update — best trade-off between speed and accuracy.

Visualization and Intuition

- Imagine a ball rolling down a curved surface — the goal is to reach the lowest point (minimum).
- The slope (gradient) tells how steep the surface is.
- The learning rate η decides step size:
  - Too small $\rightarrow$ slow convergence.
  - Too large $\rightarrow$ may overshoot the minimum.

$$w_{\text{new}} = w_{\text{old}} - \eta \nabla J(w)$$

Goal: Minimize $J(w)$ efficiently by iterative learning.

Example Calculation

Suppose:

$$h(x) = 4, \quad y = 6, \quad x = 3, \quad \eta = 0.1$$

Then:

$$\Delta w = 0.1 \times (6 - 4) \times 3 = 0.6$$

So:

$$w := w + 0.6$$

Interpretation: The weight increases to reduce the next prediction error.

Motivation: Stochastic Gradient

- **Goal:** Minimize the error (cost) function by adjusting model weights.
- **Problem:** Batch Gradient Descent is slow for large datasets.
- **Solution:** Update weights after each example — **Stochastic Gradient Descent (SGD)**.

Error Function

Given training examples (x_i, y_i):

$$E(\mathbf{w}) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - h_{\mathbf{w}}(x_i))^2$$

where:

$$h_{\mathbf{w}}(x_i) = \sum_{j} w_j x_{ij}$$

Batch Gradient Descent

$$\frac{\partial E}{\partial w_j} = -\frac{1}{m}\sum_{i=1}^{m}(y_i - h_{\mathbf{w}}(x_i))x_{ij}$$

$$w_j := w_j + \eta\sum_{i=1}^{m}(y_i - h_{\mathbf{w}}(x_i))x_{ij}$$

Problem: Needs the entire dataset for one update.

Stochastic Gradient Descent (SGD)

Instead of waiting for all data, update after each example:

$$\boxed{w_j := w_j + \eta(y_i - h_{\mathbf{w}}(x_i))x_{ij}}$$

- η – learning rate
- $(y_i - h_{\mathbf{w}}(x_i))$ – error for example i
- x_{ij} – input value for feature j

Algorithm (SGD)

- Initialize weights randomly.
- Repeat until convergence:
  - For each training example (x_i, y_i):

$$w_j := w_j + \eta(y_i - h_{\mathbf{w}}(x_i))x_{ij}$$

Example

Given:

$$h(x) = 4,\ y = 6,\ x = 3,\ \eta = 0.1$$

Then:

$$\Delta w = 0.1 \times (6 - 4) \times 3 = 0.6$$

$$w := w + 0.6$$

Each example slightly adjusts the weight toward the correct value.

Comparison: Batch vs Stochastic

Type	Update Rule	Remarks
Batch GD	$w_j := w_j + \eta \sum_i (y_i - h(x_i))x_{ij}$	Uses all examples per step
Stochastic GD	$w_j := w_j + \eta (y_i - h(x_i))x_{ij}$	One example per update

Advantages and Limitations

Advantages:

- Faster updates for large datasets
- Can escape shallow local minima

Limitations:

- Updates are noisy
- May oscillate near the minimum