Introduction

Artificial Neural Networks (ANNs) are computational models inspired by the human brain. They replicate human brain functions to process and analyze complex data, making them a fundamental tool in artificial intelligence and machine learning.

Key Features:

- **Self-learning ability**: ANNs improve performance through training.
- **Non-linearity**: Capable of solving complex patterns and relationships.
- **Parallel Processing**: Can handle multiple computations simultaneously.
- **Generalization**: Learns patterns from data and applies them to unseen inputs.

2. History of Neural Networks

Evolution Timeline

1. **1943**: McCulloch and Pitts proposed the first computational model of a neuron.
2. **1957**: Frank Rosenblatt introduced the Perceptron model, the first neural network.
3. **1960s**: Development of per-layered perceptron (MLP) began.
4. **1974**: Paul Werbos introduced the backpropagation algorithm.
5. **1980s**: Rummelhart and McClelland advanced backpropagation techniques.
6. **1982**: Hopfield introduced a new type of neural network.
7. **1990s-Present**: Evolution of Deep Learning with CNNs, RNNs, and transformers.

Artificial Neural Network Architecture

An **Artificial Neural Network (ANN)** is a computing system inspired by the structure and functioning of the human brain. It consists of interconnected processing units (neurons) that work together to process information and generate outputs. The architecture of an ANN typically consists of three fundamental layers:

1. **Input Layer**
2. **Hidden Layer(s)**
3. **Output Layer**

Each of these layers plays a crucial role in data processing and pattern recognition.

1. Input Layer

- **Purpose:** The input layer is responsible for receiving raw data from external sources and passing it to the next layer without any computation.
- **Structure:** This layer contains **neurons (nodes)**, where each neuron corresponds to a feature in the input data.
- **Function:** Each neuron takes in an input, assigns a weight, and passes the weighted input to the next layer.

Example:
For an ANN classifying handwritten digits (0-9), the input layer would receive pixel values from an image.

2. Hidden Layers

- **Purpose:** The hidden layers perform complex computations and extract meaningful features from input data.
- **Structure:** A neural network can have **one or more hidden layers**, with each neuron in these layers receiving inputs from previous layers and passing outputs to the next.
- **Functions:** Each neuron in a hidden layer performs the following operations:
  - **Receives weighted inputs** from the previous layer.
  - **Applies a bias** to shift the activation threshold.
  - **Processes the result through an activation function** to introduce non-linearity.

Common Activation Functions:

- **ReLU (Rectified Linear Unit):** Allows only positive values, setting negatives to zero.
- **Sigmoid:** Maps input between 0 and 1, useful for probability-based classification.
- **Tanh:** Maps input between -1 and 1, zero-centered for better optimization.

More hidden layers allow the network to learn deeper representations of the input data.

3. Output Layer

- **Purpose:** The output layer generates the final result or decision of the neural network.
- **Structure:** The number of neurons in this layer corresponds to the number of possible output categories.
- **Function:** The output is calculated based on the activations from the previous hidden layers.

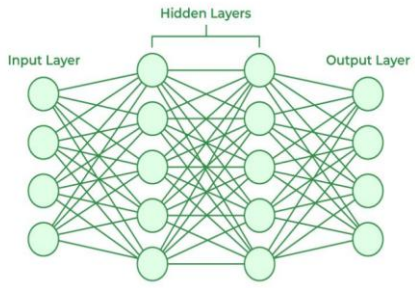
Example:

- In a **binary classification task**, the output layer has **one neuron** with a sigmoid activation function, returning values between 0 and 1.
- In a **multi-class classification task**, the output layer has **multiple neurons**, each representing a class, and uses the **softmax activation function** to produce probability values for each class.

Artificial Neural Network Flow

1. **Data is fed into the Input Layer.**
2. **Hidden Layers process the data** using weights, biases, and activation functions.

3. **The Output Layer generates a final prediction** based on learned pattern.



ANN Architecture

Advantages of Neural Networks

Neural networks offer several advantages due to their **adaptive learning** and **generalization capabilities**:

1. **Adaptive Learning:** Neural networks can learn from data **without requiring prior assumptions** about the underlying patterns.
2. **Universal Function Approximation:** Models like **feedforward multilayer perceptron (MLP)** and **radial basis function (RBF) networks** have been proven to be **universal function approximators**.
3. **Non-Linearity and Generalization:** Neural networks are **non-linear models** capable of learning complex relationships and **generalizing well** to unseen data.

Applications of Neural Networks

Neural network applications can be categorized into the following areas:

1. **Clustering:**
   - Neural networks identify **similarities** between data patterns and group them into **clusters**.
   - Common applications include **data compression** and **data mining**.
2. **Classification / Pattern Recognition:**
   - Neural networks assign input patterns (e.g., **handwritten symbols**) to predefined **classes**.
   - Used in applications like **image recognition** and **speech processing**.
3. **Function Approximation:**
   - Neural networks estimate **unknown functions** based on noisy input data.
   - Widely used in **engineering, physics, and scientific modeling**.
4. **Prediction / Dynamical Systems:**
   - Neural networks **forecast future values** based on time-series data.
   - Essential for **decision support systems** in **finance, weather forecasting, and stock market prediction**.

Key Features of Artificial Neural Networks

1. **Adaptive Learning** – ANNs can learn from data without explicit programming.
2. **Self-Organization** – Can reorganize connections based on learning patterns.
3. **Real-Time Operation** – Performs fast computations for real-time decision-making.
4. **Fault Tolerance** – Can handle noisy and incomplete data effectively.
5. **Parallel Processing** – Processes multiple data inputs simultaneously.

Characteristics of Artificial Neural Networks

Neural networks have specific characteristics that differentiate them from traditional algorithms:

1. **Composed of Neurons:** Mimics biological neurons to process information.
2. **Layered Structure:** Contains an **input layer, hidden layers, and an output layer**.
3. **Weight-Based Learning:** Connections between neurons have **weights** that adjust during training.
4. **Activation Functions:** Uses **mathematical functions** (e.g., ReLU, Sigmoid) to introduce non-linearity.
5. **Data-Driven:** Requires **large datasets** for training and improving accuracy.
6. **Error Minimization:** Uses learning algorithms (e.g., **Backpropagation**) to reduce errors.
7. **Pattern Recognition:** Excels in recognizing **patterns and trends** in data.

3. Types of Artificial Neural Networks

ANNs come in various types, each suited for different tasks:

1. Feedforward Neural Network (FNN)

- The **simplest** type of ANN.
- Data moves in **one direction**: **Input → Hidden Layer(s) → Output**.
- Used for **classification and regression** tasks.
- **Example:** Image recognition, speech processing.

2. Convolutional Neural Network (CNN)

- Specially designed for **image processing**.
- Uses **convolutional layers** to extract features like edges, textures, and shapes.
- **Example:** Facial recognition, medical imaging.

3. Recurrent Neural Network (RNN)

- Designed for **sequential data processing** (e.g., time-series data, language processing).
- Has **memory cells** to retain information from previous inputs.
- **Example:** Chatbots, stock market prediction.

4. Radial Basis Function (RBF) Network

- Uses **radial basis functions** as activation functions.
- Good for **function approximation and pattern recognition**.
- **Example:** Fraud detection, medical diagnosis.

5. Self-Organizing Map (SOM)

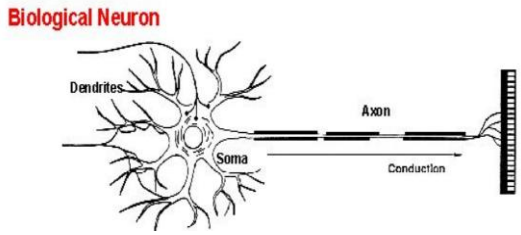
- An **unsupervised learning** network used for **clustering** and **data visualization**.
- **Example:** Market segmentation, anomaly detection.

6. Hopfield Network

- A **fully connected** network where each neuron connects to every other neuron.
- Used for **associative memory and optimization**.
- **Example:** Error correction, content retrieval.

Structure and Working of Biological Neural Network (BNN)

A **Biological Neural Network (BNN)** is a network of neurons in the human brain and nervous system that processes information through electrochemical signals. These networks serve as the inspiration for **Artificial Neural Networks (ANNs)** in artificial intelligence.



1. Structure of a Biological Neural Network

The biological neural network consists of **neurons**, which are the basic functional units of the nervous system. A neuron is responsible for transmitting and processing information through electrical and chemical signals.

Components of a Neuron

A biological neuron consists of the following main components:

1.1 Dendrites

- Dendrites are **branch-like structures** attached to the neuron's **cell body** (soma).
- They **receive electrical impulses** (signals) from other neurons and pass them to the soma for processing.
- The number of dendrites varies across different types of neurons.

1.2 Soma (Cell Body)

- The soma contains the **nucleus**, which is responsible for maintaining the neuron's functions.
- It processes the incoming signals received from dendrites and determines whether the signal should be transmitted further.

1.3 Axon

- The axon is a **long, tube-like extension** that transmits signals away from the soma toward other neurons or muscles.
- Axons can be **short or long**, with some extending up to a meter in length (e.g., neurons in the spinal cord).
- The axon is covered by a **myelin sheath**, which helps in faster transmission of signals.

1.4 Synapse

- The synapse is the **junction between two neurons** where communication occurs.
- The signal is transmitted from one neuron to another through **neurotransmitters**, which are chemical messengers.
- The transmitting neuron releases neurotransmitters into the **synaptic cleft**, which binds to receptors on the receiving neuron, initiating an electrical response.

1.5 Myelin Sheath

- Some neurons have a **fatty layer** called the myelin sheath, which **insulates the axon** and helps in **fast signal transmission**.
- Myelinated neurons transmit signals **10 times faster** than non-myelinated neurons.

2. Working of a Biological Neural Network

The **functioning of a biological neural network** is based on **electrical and chemical signaling**. It involves **three main processes**:

2.1 Signal Reception (Input Stage)

- Signals from **external stimuli** (light, sound, touch) or **other neurons** are received by the dendrites.
- The strength of the signal determines if it will be processed further.

2.2 Signal Processing (Computation Stage)

- The neuron's soma (cell body) **integrates and processes** all incoming signals.
- If the combined signal strength exceeds a certain threshold, the neuron **fires an action potential** (electrical impulse).
- If the threshold is not reached, the signal **dies out** and is not transmitted further.

2.3 Signal Transmission (Output Stage)

- When the neuron fires, the action potential travels down the axon to the **axon terminals**.
- At the axon terminals, **neurotransmitters** are released into the synapse.
- These neurotransmitters carry the signal to the next neuron, where the process repeats.

Feature	Biological Neural Network (BNN)	Artificial Neural Network (ANN)
Structure	Made of neurons, dendrites, axons, and synapses.	Made of artificial neurons (nodes) with weighted links.
Processing Speed	Slow (chemical signals).	Fast (digital computation).
Signal Transmission	Electrochemical signals.	Mathematical functions (weights, biases, activation).
Learning Mechanism	Learns autonomously.	Requires training (supervised, unsupervise reinforcement).
Fault Tolerance	Self-repairing, robust.	Less robust, affected by errors.
Energy Consumption	Low energy use (bioelectric signals).	High energy demand (computational pow
Memory Storage	Distributed (synaptic plasticity).	Stored in units (weight updates).
Flexibility & Adaptability	Continuous learning.	Needs retraining for new tasks.
Parallel Processing	Highly parallel (billions of neurons).	Limited parallelism.
Scalability	Naturally scales with growth.	Needs more resources for scaling.
Decision Making	Uses reasoning, emotions, experience.	Based on pattern recognition.
Accuracy	Prone to fatigue, distractions.	High accuracy, data-dependent.
Lifespan	Functions for a lifetime.	Limited by hardware/software lifespan.
Application Areas	Found in living beings.	Used in AI, robotics, automation.

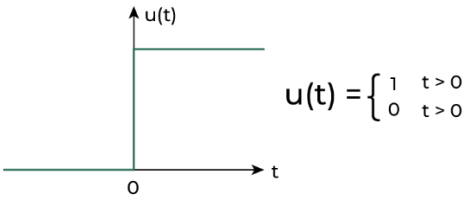
1. Activation Functions

What are activation functions?
Activation functions decide whether a neuron should be activated based on its input. They introduce **non-linearity**, allowing the neural network to learn complex patterns.

2. Types of Activation Functions 🚀

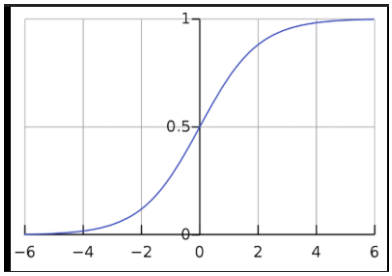
🔹 Step Function

- Outputs **1** if input is above a threshold, else **0**.
- *Limitation:* Not differentiable, making it hard to train deep networks.
- *Use Case:* Early perceptron models.



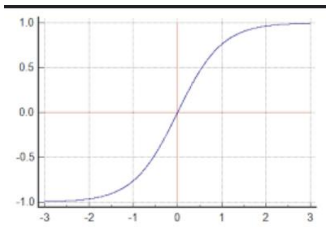
🔹 Sigmoid Function

- Outputs values between **0 and 1**.used for binary classification.
- Formula: f(x) = 1 / (1 + e^(-x))
- *Limitation:* Causes **vanishing gradients**, making deep learning difficult.
- *Use Case:* Binary classification (e.g., Spam detection).



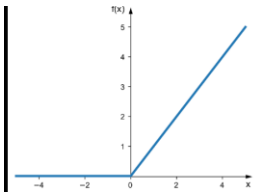
🔹 Tanh (Hyperbolic Tangent) Function

- Similar to Sigmoid but outputs between **-1 and 1**.
- Formula: f(x) = (e^x - e^(-x)) / (e^x + e^(-x))
- *Advantage:* Zero-centered output helps optimization.
- *Limitation:* Still suffers from vanishing gradients.



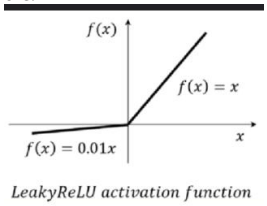
🔹 ReLU (Rectified Linear Unit)

- Outputs **x** if x > 0, else **0**.
- Formula: f(x) = max(0, x)
- *Advantage:* Simple, efficient & prevents vanishing gradient issues.
- *Limitation:* Some neurons can become inactive (Dying ReLU problem).
- *Use Case:* Almost every deep learning model (e.g., image processing, NLP).



🔹 Leaky ReLU

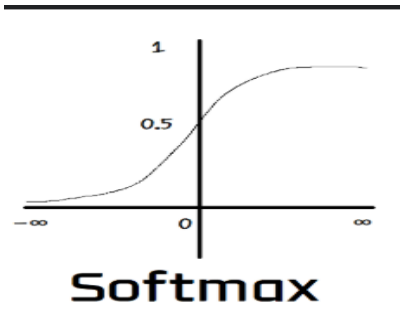
- Similar to ReLU but allows small slope for negative values.
- Formula: f(x) = x (if x > 0), αx (if x < 0) (α is a small value like 0.01).
- *Advantage:* Solves Dying ReLU issue.
- *Use Case:* Deep learning tasks where ReLU neurons die.



- *LeakyReLU activation function*

🔹 *Softmax Function*

- Converts values into probabilities summing to **1**.
- Formula: f(x_i) = e^(x_i) / Σ e^(x_j)
- *Use Case:* Used in **multi-class classification** problems (e.g., image recognition).



Function	Range	Use Case	Limitation
Step	{0,1}	Perceptron models	Not differentiable
Sigmoid	(0,1)	Binary classification	Vanishing gradients
Tanh	(-1,1)	Hidden layers	Vanishing gradients
ReLU	[0,∞)	Deep learning	Dying ReLU problem
Leaky ReLU	(-∞,∞)	Solves dying ReLU	Needs fine-tuning
Softmax	(0,1)	Multi-class classification	Computationally expensive

Why ReLU is the Most Commonly Used Activation Function in Neural Networks?

The **Rectified Linear Unit (ReLU)** activation function is widely used in deep learning because it overcomes limitations of earlier activation functions like **sigmoid** and **tanh**. Its simple yet effective behavior makes it the default choice for hidden layers in neural networks.

Mathematical Definition of ReLU

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

- If x>, then f(x)=x (linear behavior).
- If x≤0x then f(x)=0 (non-linearity).

Reasons Why ReLU is Preferred

1️⃣ *Solves the Vanishing Gradient Problem*

- **Sigmoid and Tanh** suffer from **vanishing gradients**, where gradients become very small in deep networks, slowing learning.
- **ReLU does not saturate for positive values**, allowing gradients to remain large and improving learning efficiency.

2️⃣ *Computationally Efficient*

- Unlike sigmoid/tanh, **ReLU does not require exponential calculations**, making it faster to compute.
- Only requires a simple **max(0, x)** operation.

3️⃣ *Introduces Non-linearity*

- Neural networks require **non-linearity** to learn complex patterns.
- ReLU achieves this while keeping **positive values linear**, which helps in efficient gradient flow.

4️⃣ *Sparse Activation (Promotes Efficient Learning)*

- ReLU outputs **zero for negative inputs**, reducing the number of active neurons.
- This leads to a **sparse network**, reducing computation and improving generalization.

5️⃣ *Works Well in Deep Networks*

- Deep networks trained with **ReLU converge faster** compared to sigmoid or tanh.
- It prevents the **vanishing gradient** issue in deep architectures.

Models of Neurons in Artificial Neural Networks

Neural network models are mathematical representations of biological neurons. The most fundamental models are:

1. **McCulloch & Pitts Model**
2. **Perceptron Model**
3. **Adaline (Adaptive Linear Neuron) Model**

Each of these models plays a crucial role in the evolution of artificial neural networks.

1. McCulloch & Pitts Model (MCP Model)

Introduction

The **McCulloch-Pitts (MCP) model**, proposed in 1943 by Warren McCulloch and Walter Pitts, is the **simplest model of an artificial neuron**. It is a **binary threshold model**, meaning the neuron either **fires (1) or does not fire (0)** based on a threshold value.

Structure of MCP Neuron

- Takes **multiple inputs (x_1, x_2, ..., xn)**.
- Each input has an **associated weight (w_1, w_2, ..., wn)**.
- Computes a **weighted sum** of inputs.
- Applies a **threshold function** (Step Function).

Mathematical Representation

$$y = f\left(\sum w_i x_i\right)$$

Where:

- x_i = Inputs
- w_i = Weights
- $\sum w_i x_i$ = Weighted sum
- y = Output (0 or 1)

The **activation function** is a **step function**:

$$f(x) = \begin{cases} 1, & \text{if } \sum w_i x_i \geq \theta \\ 0, & \text{otherwise} \end{cases}$$

where θ is the threshold value.

Example

Consider an MCP neuron with:

- Inputs: $x_1 = 1, x_2 = 1$
- Weights: $w_1 = 0.6, w_2 = 0.5$
- Threshold $\theta = 1$

$$\sum w_i x_i = (0.6 \times 1) + (0.5 \times 1) = 1.1$$

Since $1.1 \geq \theta$, the output **fires (1)**.

Limitations

- **Cannot learn from data** (fixed weights).
- **Only handles binary outputs.**
- **Cannot solve non-linearly separable problems** (e.g., XOR problem).

Perceptron Model

The **Perceptron Model** is a fundamental neural network model introduced by **Frank Rosenblatt in 1958**. It is the **simplest type of artificial neural network** and forms the basis of more advanced neural architectures.

The perceptron is used for **binary classification**, meaning it can classify input data into one of two categories (e.g., **yes/no, spam/non-spam, 0/1**).

2. Structure of a Perceptron

A perceptron consists of the following key components:

1. **Inputs (x1,x2,...,xn)** – The features or attributes of the input data.
2. **Weights (w1,w2,...,wn)** – Each input is assigned a weight that determines its importance.
3. **Summation Function (∑wixi+b)** – Computes the weighted sum of inputs and adds a bias (b).
4. **Activation Function** – Decides whether the perceptron should **fire (1)** or **not fire (0)** based on the threshold.

3. Working of a Perceptron

The perceptron processes input data in the following steps:

Step 1: Calculate the Weighted Sum

$$\text{Net Input} = \sum w_i x_i + b$$

where:

- x_i = Inputs
- w_i = Weights
- b = Bias

Step 2: Apply the Activation Function

The perceptron uses a **Step Function (Threshold Function)** as its activation function:

$$f(\text{Net Input}) = \begin{cases} 1, & \text{if } \sum w_i x_i + b \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

This function **determines the output** based on whether the net input crosses the threshold.

5. Example of Perceptron for an AND Gate

Let's build a perceptron to **simulate the AND gate**:

x_1	x_2	AND Output
0	0	0
0	1	0
1	0	0
1	1	1

Step 1: Assign Initial Weights and Bias

Let's assume:

- $w_1 = 0.5, w_2 = 0.5$
- Bias (b) = -0.7

Step 2: Compute Net Input for Each Case

For $x_1 = 1, x_2 = 1$:

$$(0.5 \times 1) + (0.5 \times 1) - 0.7 = 0.3$$

Since **0.3 ≥ 0**, the perceptron **fires (1)**, which is correct for the AND gate.

For $x_1 = 0, x_2 = 1$:

$$(0.5 \times 0) + (0.5 \times 1) - 0.7 = -0.2$$

Since **-0.2 < 0**, the perceptron **does not fire (0)**, which is also correct.

Limitations of Perceptron

- **Cannot solve non-linearly separable problems** (e.g., XOR problem).
- **Uses a step function**, which **does not allow smooth weight updates**.
- **Works only for binary classification**, not for multi-class problems.

Learning in Perceptron

The **Perceptron Learning Algorithm** enables a perceptron to **adjust its weights** so that it correctly classifies input data. The algorithm works iteratively by comparing the predicted output with the actual target output and updating the weights accordingly

Steps in Perceptron Learning Algorithm

The perceptron learning process follows these steps:

Step 1: Initialize Weights and Bias

- Assign **random small values** (e.g., 0.1 or -0.2) to each weight (wiw_iwi).
- Set an initial bias (bbb).

Step 2: Compute the Net Input

For a given input vector $X = (x_1, x_2, \ldots, x_n)$, compute the weighted sum:

$$\text{Net Input} = \sum w_i x_i + b$$

Step 3: Apply the Activation Function

The perceptron uses a **step function**:

$$f(\text{Net Input}) = \begin{cases} 1, & \text{if } \sum w_i x_i + b \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

This determines whether the perceptron **fires (1)** or **does not fire (0)**.

Step 4: Compute Error

The **error** is calculated as the difference between the target output (ttt) and the predicted output (yyy):

$$E = t - y$$

If E=0 the prediction is correct → **No weight update** needed.
If E≠0, update the weights using the **Perceptron Learning Rule**.

Step 5: Update Weights and Bias

Adjust the weights using the **Perceptron Learning Rule**:

$$w_i = w_i + \eta(t - y)x_i$$

where:

- η = Learning rate (small positive value, e.g., 0.01)
- t = Actual output (target)
- y = Predicted output

The **bias** is also updated:

$$b = b + \eta(t - y)$$

Step 6: Repeat Until Convergence

- Repeat the process for all training samples.
- Stop when the perceptron correctly classifies all training examples or when a maximum number of iterations is reached.

Introduction to Adaline Model

The **Adaline (Adaptive Linear Neuron) model**, developed by Bernard Widrow & Marcian Hoff (1960), is an improvement over the **Perceptron model**. Unlike the perceptron, which uses a **step function**, Adaline uses a **linear activation function** and learns by minimizing the **Mean Squared Error (MSE)**.

Structure of the Adaline Model

Adaline consists of the following key components:

1. **Inputs (x1,x2,...,xn)** – Features of the input data.
2. **Weights (w1,w2,...,wn)** – Each input is assigned a weight.
3. **Summation Function** – Computes the weighted sum of inputs.
4. **Linear Activation Function** – Unlike perceptron, it does **not use a step function**.
5. **Learning Rule (Gradient Descent)** – Adjusts weights using the Mean Squared Error (MSE).

3. Mathematical Representation of Adaline

Step 1: Compute Net Input (Weighted Sum)

$$y_{net} = \sum w_i x_i + b$$

where:

- x_i = Inputs
- w_i = Weights
- b = Bias

Step 2: Apply Activation Function

Unlike the perceptron, Adaline does not use a step function. Instead, the activation function is identity (linear function):

$$f(y_{net}) = y_{net}$$

This means the output is continuous, not just 0 or 1.

Step 3: Compute Error

The error is calculated as:

$$E = t - y_{net}$$

where t is the actual target value.

Step 4: Update Weights Using Gradient Descent

Adaline minimizes the Mean Squared Error (MSE):

$$MSE = \frac{1}{2}\sum(t - y_{net})^2$$

Weights are updated using Gradient Descent:

$$w_i = w_i + \eta(t - y_{net})x_i$$

$$b = b + \eta(t - y_{net})$$

where:

- η = Learning rate (small positive value, e.g., 0.01)
- t = Actual target value
- y_{net} = Net input

5. Example: Training an Adaline for an AND Gate

Step 1: Define Inputs and Target Outputs

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

Step 2: Initialize Weights and Bias

Let's assume:

- $w_1 = 0.2, w_2 = -0.1, b = 0.1$
- Learning rate $\eta = 0.1$

Step 3: Compute Net Input and Output for Each Case

For $x_1 = 1, x_2 = 1, t = 1$:

$$y_{net} = (1 \times 0.2) + (1 \times -0.1) + 0.1 = 0.2$$

Error:

$$E = t - y_{net} = 1 - 0.2 = 0.8$$

Step 4: Update Weights and Bias

$$w_1 = 0.2 + (0.1 \times 0.8 \times 1) = 0.28$$

$$w_2 = -0.1 + (0.1 \times 0.8 \times 1) = -0.02$$

$$b = 0.1 + (0.1 \times 0.8) = 0.18$$

Repeat for all training examples until the error is minimized.

6. Advantages of Adaline

Can learn continuous outputs (better for regression tasks).
Minimizes Mean Squared Error (MSE) → More stable learning.
Uses Gradient Descent, leading to **smoother convergence**.

7. Limitations of Adaline

Still cannot solve XOR problem (requires multi-layer networks).
Sensitive to learning rate (too large → oscillations, too small → slow convergence).
Needs normalized inputs for better performance.

Feature	McCulloch-Pitts (MCP) Model	Perceptron Model	Adaptive Linear Neuron (Adaline) Model
Developed By	McCulloch & Pitts (1943)	Frank Rosenblatt (1958)	Bernard Widrow & Marcian Hoff (1960)
Type	Early artificial neuron model	Binary classifier	Linear classifier with learning
Activation Function	Step function (Threshold-based)	Step function (Threshold-based)	Linear activation function before applying a threshold
Learning Rule	No learning mechanism	Perceptron Learning Rule (Weight update only when misclassification occurs)	Least Mean Squares (LMS) rule (Minimizes error before thresholding)
Weights Update	Fixed, no learning	Updated after each misclassified sample	Updated based on error before applying activation
Error Measurement	Not applicable (Fixed rules)	Misclassification error	Mean Squared Error (MSE)
Output	Binary (0 or 1)	Binary (0 or 1)	Continuous output (before applying threshold)
Limitation	Cannot learn from data, only theoretical	Only works for **linearly separable** problems	More robust than Perceptron but still limited to linear problems
Usage	Early theoretical studies on neural networks	Simple binary classification (e.g., AND, OR logic gates)	Regression and classification tasks (e.g., stock price prediction)

Basic Learning Laws in Neural Networks

Neural networks learn by adjusting weights using different learning laws. These laws define **how** a neural network modifies its connections to improve performance.

1. Hebbian Learning Rule

📌 **Concept:** "Neurons that fire together, wire together."
📌 **Type: Unsupervised Learning**
📌 **Introduced by: Donald Hebb (1949)**

✅ **Weight Strengthening:** If **both input and output neurons are active simultaneously**, their connection is strengthened.
✅ **Biologically Inspired:** Mimics how neurons in the brain form **stronger synaptic connections** with frequent activation.
✅ **No Supervision Required:** The network **learns by itself** based on data patterns.
✅ **Best for Associative Memory:** Used in networks where recalling patterns is important.

Mathematical Formula:

$$w_{ij} = w_{ij} + \eta x_i y_j$$

where:

- w_{ij} = Weight between neuron i and j
- η = Learning rate
- x_i = Input value
- y_j = Output value

Example Applications:

✅ **Face recognition**
✅ **Self-organizing maps**
✅ **Pattern association (e.g., linking an image to a name)**

Limitations:

❌ **Cannot handle negative weight updates well**
❌ **Works best only with strongly correlated inputs**

2. Perceptron Learning Rule

📌 **Concept:** Adjusts weights based on classification error.
📌 **Type: Supervised Learning**
📌 **Introduced by: Frank Rosenblatt (1958)**

Key Points:

✅ **Starts with Random Weights:** Initial weights are set to small random values.
✅ **Step Activation Function:** Uses a **threshold function** to determine the output (0 or 1).
✅ **Binary Classification:** Works well for **linearly separable** data (e.g., AND, OR logic gates).
✅ **Adjusts Weights Based on Errors:** If the **predicted output is wrong**, the weights are updated.

Mathematical Formula:

$$w_i = w_i + \eta(t - y)x_i$$

where:

- t = Target output
- y = Predicted output
- x_i = Input
- η = Learning rate

Example Applications:

✓ **Spam detection (spam vs non-spam email)**
✓ **Image classification (cat vs dog)**

Limitations:

✗ **Cannot solve non-linearly separable problems (e.g., XOR problem)**
✗ **Uses only a step function (not smooth learning like sigmoid or ReLU)**

3. Delta Learning Rule (Widrow-Hoff Rule)

📌 **Concept:** Minimizes error using **Mean Squared Error (MSE)**.
📌 **Type: Supervised Learning**
📌 **Introduced by: Bernard Widrow & Ted Hoff (1960)**

Key Points:

✅ **Uses Gradient Descent:** Adjusts weights using **MSE minimization**.
✅ **Works with Continuous Outputs:** Unlike perceptron, which uses a **step function**, Adaline uses a **linear activation function**.
✅ **Better Stability:** Weight updates are **gradual**, leading to more stable learning.

Mathematical Representation:

$$w_i = w_i + \eta(t - y_{net})x_i$$

where:

- y_{net} = Weighted sum of inputs
- $E = \frac{1}{2}\sum(t - y_{net})^2$ (Mean Squared Error)

Example:

- Used in **Adaline (Adaptive Linear Neuron)** networks for **pattern recognition and regression**.

Limitation:

- Requires **normalized inputs** for efficient learning.

Example Applications:

✓ **Regression tasks (predicting stock prices, temperature forecasting)**
✓ **Pattern recognition**
✓ **Adaline networks**

Limitations:

✗ **Slow learning if the learning rate (η\etaη) is too small**
✗ **Requires data normalization for better performance**

4. Correlation Learning Rule

📌 **Concept:** Strengthens weights if **input and output are correlated**.
📌 **Type: Supervised Learning**

Key Points:

✅ **Directly Links Input & Output:** Increases weight if input and target output **match**.
✅ **No Need for Error Calculation:** Unlike Perceptron and Delta rules, which calculate **error**, correlation rule only focuses on **similarity** between input and output.
✅ **Good for Pattern Matching:** Works well in networks where **similar patterns should be strengthened**.

Mathematical Representation:

$$w_i = w_i + \eta x_i t$$

Example:

- Used in **image recognition** and **speech processing**.

Limitation:

- Cannot handle **complex, non-linear relationships**.

Example Applications:

✓ **Speech recognition (matching spoken words to text)**
✓ **Image recognition (detecting familiar objects in pictures)**

Limitations:

✗ **Not effective for complex, non-linear relationships**
✗ **Requires clearly labeled data**

5. Outstar Learning Rule

📌 **Concept:** Used for **self-organizing networks**, where neurons adapt to **external inputs**.
📌 **Type: Unsupervised Learning**

Key Points:

✅ **Best for Layered Networks:** Works **only if neurons are arranged in a layer**.
✅ **Helps Self-Organization:** Adjusts weights **without requiring labeled data**.
✅ **Used in Cognitive Learning Systems:** Helps in **unsupervised learning models**.

Mathematical Representation:

$$w_{ij} = w_{ij} + \eta(x_i - w_{ij})$$

Example:

- Used in **biological neural models** and **cognitive learning systems**.

Limitation:

- Works best **only when neurons are arranged in layers**.

Example Applications:

✓ **Cognitive models (brain-inspired learning systems)**
✓ **Neural networks for language processing**

Limitations:

✗ **Only works for layered architectures**
✗ **Difficult to tune learning rate (η\etaη) properly**

Comparison of Learning Rules

Learning Rule	Type	Mechanism	Example Application
Hebbian Learning	Unsupervised	Strengthens frequently used connections	**Memory association, pattern recognition**
Perceptron Learning	Supervised	Updates weights based on classification errors	**Binary classification (AND, OR gates)**
Delta Rule	Supervised	Minimizes Mean Squared Error (MSE)	**Regression, pattern recognition**
Correlation Rule	Supervised	Strengthens correlated input-output pairs	**Speech and image recognition**
Outstar Rule	Unsupervised	Self-organizing learning	**Cognitive models, layered networks**

UNIT 2

Learning and Memory in Neural Networks

1. Understanding Learning and Memory in Neural Networks

Neural networks learn from data and store patterns to make predictions. Learning involves adjusting weights and biases, while memory refers to the ability of a network to retain learned information. A well-trained network generalizes knowledge to make accurate predictions on new data. For example, when a neural network is trained to recognize handwritten digits, it remembers patterns from past examples and applies that knowledge when seeing a new digit.

2. Types of Learning in Neural Networks

Supervised Learning

Supervised learning involves training a neural network using labeled data, meaning each input has a corresponding correct output. The network learns by comparing its predictions to the correct labels and adjusting its parameters to minimize errors. The process involves making predictions, calculating error using a loss function, and updating weights using algorithms like gradient descent and backpropagation.

Supervised learning is widely used in applications like image classification, spam email detection, and speech recognition. Algorithms such as backpropagation, convolutional neural networks (CNNs), and recurrent neural networks (RNNs) are commonly used in this paradigm.

Unsupervised Learning

In unsupervised learning, the neural network does not receive labeled data. Instead, it identifies patterns and structures by grouping similar data points together. The network clusters similar data points without explicit labels, making it useful for tasks like customer segmentation, anomaly detection, and data compression.

Common algorithms include K-Means clustering, self-organizing maps (SOMs), and autoencoders. These methods allow the network to find hidden patterns in data without human supervision.

Semi-Supervised Learning

Semi-supervised learning combines supervised and unsupervised learning. It is useful when only a small portion of the data is labeled while the rest remains unlabeled. The network first learns from labeled data and then applies its knowledge to predict labels for the remaining unlabeled data.

This approach is commonly used in medical diagnosis, where there may be limited labeled medical images, and in fraud detection, where only a few fraudulent cases are identified. Algorithms such as generative adversarial networks (GANs) and graph-based neural networks help train models effectively with limited labeled data.

Reinforcement Learning

Reinforcement learning is a trial-and-error learning method where an agent interacts with an environment and learns by receiving rewards or penalties for actions taken. The agent takes an action, receives feedback (reward or penalty), and updates its strategy to maximize long-term rewards.

This method is widely used in artificial intelligence applications such as AlphaGo, which defeated world champions in the game of Go, and in self-driving cars that optimize routes and driving decisions. Algorithms like Q-learning, deep Q networks (DQN), and policy gradient methods are commonly used in reinforcement learning.

Self-Supervised Learning

Self-supervised learning is an advanced technique where the network generates its own labels instead of relying on manually labeled data. The model learns by predicting missing parts of data or solving tasks without human supervision.

This approach is used in cutting-edge AI models like BERT, which enhances Google Search by predicting missing words in sentences, and in Facebook's AI for automatic image tagging. Algorithms like contrastive learning and transformers (BERT, GPT) enable self-supervised learning to be highly effective in modern AI applications.

3. Summary: Comparison of Learning Types

Learning Type	Definition	Example Applications	Common Algorithms
Supervised Learning	Learns from labeled data	Image classification, speech recognition	Backpropagation, CNNs, RNNs
Unsupervised Learning	Finds patterns in unlabeled data	Customer segmentation, anomaly detection	K-Means Clustering, Autoencoders

Learning Type	Definition	Example Applications	Common Algorithms
Semi-Supervised Learning	Uses small labeled data with large unlabeled data	Medical diagnosis, fraud detection	GANs, Graph Neural Networks
Reinforcement Learning	Learns by trial and error using rewards	Chess-playing AI, self-driving cars	Q-Learning, Deep Q Networks
Self-Supervised Learning	Learns from data without human labels	Google Search, Facebook image recognition	Transformers (BERT, GPT), Contrastive Learning

Role of Hidden Layer in ANN
The **hidden layer** in a neural network plays a crucial role in learning and extracting patterns from input data. It acts as an intermediary between the input and output layers and is responsible for performing complex transformations. Here are the key roles of the hidden layer:

1. Feature Extraction

- The hidden layer learns to identify relevant features from raw input data by applying weighted transformations and activation functions.
- It helps in recognizing patterns such as edges in images, speech phonemes in audio, or dependencies in tabular data.

2. Non-Linearity and Representation Learning

- The activation functions (e.g., ReLU, Sigmoid, Tanh) in hidden layers introduce non-linearity, allowing the network to model complex relationships that cannot be captured by a simple linear model.

3. Hierarchical Learning

- In deep neural networks (DNNs), multiple hidden layers allow for hierarchical feature learning. Lower layers capture basic patterns, while deeper layers learn more abstract features.

4. Dimensionality Reduction

- Hidden layers help in reducing the complexity of the input data by transforming it into a more meaningful representation, often in a lower-dimensional space.

5. Weight Learning and Optimization

- Hidden layers adjust their weights using backpropagation and gradient descent to minimize the error between predicted and actual outputs.

6. Decision Boundary Formation

- The transformation in the hidden layers helps in shaping decision boundaries, making it possible to classify complex datasets effectively.

7. Encoding and Data Transformation

- In autoencoders, hidden layers compress the input into a smaller representation (encoding) and then reconstruct it (decoding), making them useful for feature learning and anomaly detection.

Multilayered Neural Network (MLNN) Architecture – A Detailed Explanation

A **Multilayered Neural Network (MLNN)** is an advanced type of artificial neural network (ANN) that consists of **multiple layers** of neurons. This structure enables the network to learn complex patterns and make accurate predictions.

1 Components of MLNN

A typical **Multilayer Neural Network** consists of the following layers:

1. Input Layer

- The **first layer** in the network.
- Receives raw input features (e.g., pixel values of an image, numerical data, text embeddings).
- Each neuron in this layer represents one input feature.

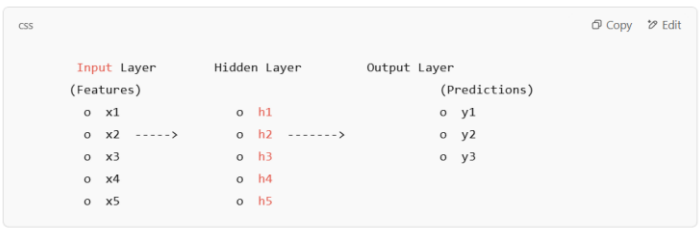
2. Hidden Layer(s)

- One or more layers between the input and output layers.
- Neurons in hidden layers apply **activation functions** (like ReLU, Sigmoid) to introduce non-linearity.
- The **more hidden layers**, the **deeper** the network, leading to better feature extraction.

3. Output Layer

- The **final layer** in the network that provides predictions.
- The number of neurons in this layer depends on the type of task:\n
  - **For Classification**: One neuron per class with a **Softmax activation function**.
  - **For Regression**: A single neuron with a **linear activation function**.

Example: A 3-layer Neural Network



👉 **Each layer is fully connected**, meaning each neuron in one layer connects to every neuron in the next layer.

1 Forward Propagation

✅ **Step 1:** Inputs pass through each layer, undergoing weighted summation and activation

$$h_j = f\left(\sum_i w_{ij} x_i + b_j\right)$$

where:

- w_{ij} = weight of connection between neuron i and neuron j.
- x_i = input feature.
- b_j = bias term.
- f = activation function (ReLU, Sigmoid, etc.).

✅ **Step 2:** The final layer outputs predictions.

2 Backpropagation (Learning Phase)

1. The network computes the **error** (difference between predicted and actual output).
2. The error is **propagated backward** to update weights using **Gradient Descent**:

$$w_{new} = w_{old} - \eta \frac{\partial E}{\partial w}$$

where:

- η = learning rate (controls how fast weights update).
- E = error function.

Error Correction in Neural Networks & How to Minimize Error

In artificial neural networks (ANNs), **error correction** is the process of adjusting weights to reduce the difference between the predicted output and the actual output. This ensures that the model learns from mistakes and improves accuracy over time.

1 What is Error in Neural Networks?

Error is the **difference between the actual output and the predicted output**.

Mathematically, error E is given by:

$$E = Y_{\text{actual}} - Y_{\text{predicted}}$$

where:

- Y_{actual} = True output.
- $Y_{\text{predicted}}$ = Output given by the model.

2 Types of Errors

1. **Mean Squared Error (MSE)**

$$MSE = \frac{1}{N}\sum (Y_{\text{actual}} - Y_{\text{predicted}})^2$$

✅ Used for regression problems.
✅ Penalizes large errors more.

2. **Cross-Entropy Loss (Log Loss)**

$$E = -\sum [Y_{\text{actual}} \log(Y_{\text{predicted}})]$$

✅ Used for classification problems.
✅ Works well with softmax activation.

3. **Absolute Error (L1 Loss)**

$$E = \sum |Y_{\text{actual}} - Y_{\text{predicted}}|$$

✅ Less sensitive to outliers than MSE.

1. Gradient Descent (Optimization Algorithm)

✅ Adjusts weights based on error to reduce loss.
✅ Formula for weight update:

$$w_{\text{new}} = w_{\text{old}} - \eta \frac{\partial E}{\partial w}$$

where:

- η = learning rate (step size).
- $\frac{\partial E}{\partial w}$ = gradient of error w.r.t weight.
- This process **repeats until the error is minimized**.

2. Learning Rate Adjustment

✅ A **small learning rate** leads to slow learning.
✅ A **large learning rate** can cause oscillations.
✅ **Adaptive learning rate** (Adam, RMSprop) balances speed & stability.

3. Increasing Training Data

✅ More data improves generalization and reduces overfitting.

4. Using Activation Functions

✅ **ReLU**, **Sigmoid**, and **Softmax** help in better learning and preventing vanishing gradients.

5. Adding Regularization (L1 & L2)

✅ Prevents overfitting by penalizing large weights.
✅ L2 Regularization (Ridge Regression):

$$E = MSE + \lambda \sum w^2$$

✅ L1 Regularization (Lasso Regression):

$$E = MSE + \lambda \sum |w|$$

6. Early Stopping

✅ Stops training when validation loss starts increasing (prevents overfitting).

7. Using Dropout

✅ Randomly removes neurons during training to avoid over-reliance on certain features.

4 Important Rules of Gradient Descent (GD)

1 **Move in the Direction of the Negative Gradient**

- Always update weights in the opposite direction of the gradient to reduce loss: Wnew=Wold−η∂L/∂W
- This ensures the model finds the optimal weights for minimizing the error.

2 **Choose an Appropriate Learning Rate η\etaη**

- **Too Small** → Slow convergence.
- **Too Large** → May diverge or oscillate.
- **Solution:** Use adaptive methods like Adam or try different learning rates.

3 **Use Mini-Batch Gradient Descent for Efficiency**

- **Batch GD** is slow for large datasets.
- **Stochastic GD** is noisy.
- **Mini-Batch GD (e.g., batch size = 32 or 64)** balances stability and speed.

4 **Normalize Data for Better Convergence**

- If input features have different scales, gradient updates can be unstable.
- **Solution:** Use feature scaling techniques like **Standardization (Z-score)** or **Min-Max scaling**.

What is Gradient Descent?

Gradient Descent is an **optimization algorithm** used to **minimize a loss function** by iteratively updating model parameters (weights and biases) in the direction of the **negative gradient**.

📌 **Goal:** Find the optimal values of parameters that minimize the error in machine learning models.

---

1. How Does Gradient Descent Work?

1 **Compute the Loss**

- The model predicts an output, and the difference from the actual value is measured using a **loss function** (e.g., Mean Squared Error).

2 **Calculate the Gradient (Derivative)**

- Compute the partial derivative of the loss function w.r.t. each parameter (weight & bias).

3 **Update Weights Using the Gradient**

- Adjust the parameters in the opposite direction of the gradient:

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W}$$

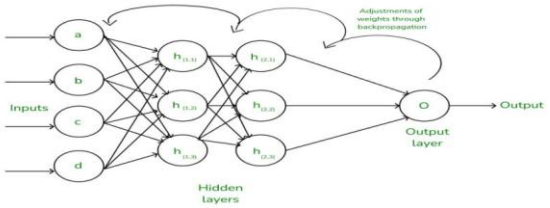
  - η = Learning rate (step size).
  - $\frac{\partial L}{\partial W}$ = Gradient of loss function.

4 **Repeat Until Convergence**

- Continue updating weights until the loss stops decreasing (model converges).

Backpropagation

Backpropagation (short for "Backward Propagation of Errors") is a method used to train artificial neural networks. Its goal is to reduce the difference between the model's predicted output and the actual output by adjusting the weights and biases in the network



Steps of Backpropagation

Step 1: Initialize Weights and Biases

- Start with small random values for **weights** (w) and **biases** (b).

Step 2: Forward Propagation (Making Predictions)

1. **Compute Weighted Sum for Each Neuron**

$$Z = (W \times X) + b$$

   - X = Input values
   - W = Weights
   - b = Bias
2. **Apply Activation Function** (to introduce non-linearity)

$$A = f(Z)$$

   - Common activation functions:
     - Sigmoid: $f(Z) = \frac{1}{1+e^{-Z}}$
     - ReLU: $f(Z) = \max(0, Z)$
3. **Repeat Until Output Layer is Reached.**

Step 3: Compute Error (Loss Function)

Measure the difference between the **predicted output** A_{pred} and the **actual output** Y.

Common Loss Functions:

- Mean Squared Error (MSE) (for Regression):

$$L = \frac{1}{n} \sum (Y - A_{pred})^2$$

Step 4: Update Weights and Biases Using Gradient Descent

- Weights and biases are updated using the gradient descent formula:

$$W_{new} = W_{old} - \eta \times \frac{\partial L}{\partial W}$$

$$b_{new} = b_{old} - \eta \times \frac{\partial L}{\partial b}$$

- η (Learning Rate) controls the step size for updates.
- The process repeats to **reduce error** in future predictions.

Step 5: Repeat Until Training is Complete

- **Repeat steps 2 to 5** until the error is **very small** or after a set number of training cycles (**epochs**).

Advantages & Disadvantages

✅ **Advantages:**
✓ Efficient and fast.
✓ Works well for deep neural networks.
✓ Can handle complex problems.

❌ **Disadvantages:**
✗ Can get stuck in local minima.
✗ May require a lot of data.
✗ Learning rate tuning is critical.

Feedforward Neural Networks (FNNs) - Data flows in one direction.

Definition

- A **feedforward neural network** is a type of artificial neural network where information moves **only in one direction** – from input to output.
- There are **no loops or cycles** in the connections between neurons.
- It is commonly used for **classification and regression** tasks.

Architecture

A feedforward network consists of:

1. **Input Layer** – Receives the input data.
2. **Hidden Layers** – Perform computations and transformations.
3. **Output Layer** – Produces the final result.

Example of Feedforward Network

- A simple **3-layer neural network** for digit recognition:

- **Input Layer:** Pixels of an image (e.g., handwritten digit).
- **Hidden Layers:** Process the data and extract patterns.
- **Output Layer:** Predicts a digit (0-9).

Mathematical Representation

For a single neuron:

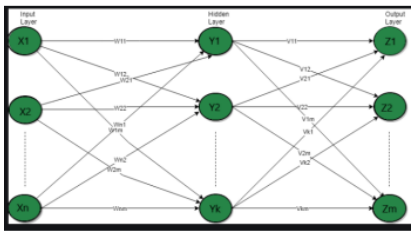
$Z=W\times X+b$

Where:

- XXX = Input
- WWW = Weights
- bbb = Bias
- f(Z)= Activation function (e.g., ReLU, Sigmoid)

Applications of Feedforward Neural Networks

✓ **Image Recognition** – Classifying objects in images.
✓ **Speech Recognition** – Converting voice to text.
✓ **Medical Diagnosis** – Detecting diseases from medical scans.
✓ **Stock Market Prediction** – Forecasting trends.
✓ **Spam Detection** – Identifying spam emails.



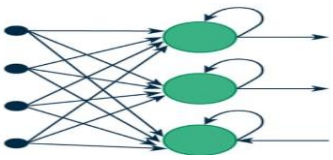
Feedback (Recurrent) Neural Networks (RNNs) - Data loops back in the network.

A **feedback neural network**, also known as a **recurrent neural network (RNN)**, has connections that loop back to previous neurons.

- Unlike feedforward networks, **it remembers past data**, making it suitable for **sequential tasks**.
- These networks use **memory** to retain information from previous inputs.

Architecture

1. **Input Layer** – Takes in a sequence of data.
2. **Hidden Layers** – Process data while keeping track of past information.
3. **Output Layer** – Produces the final result.
4. **Feedback Loops** – Send the output of a layer back into previous layers.



Example of Feedback Network

- A **language model for text prediction**:
  - **Input:** "I love to eat..."
  - **Hidden Layer:** Stores previous words and predicts the next word.
  - **Output:** "pizza" or "ice cream" based on past patterns.

Applications of Feedback Neural Networks

✓ **Time Series Prediction** – Stock prices, weather forecasting.
✓ **Speech Recognition** – Understanding spoken words.
✓ **Text Generation** – AI chatbots like ChatGPT.
✓ **Anomaly Detection** – Detecting fraud in banking.
✓ **Robotics** – Controlling movements based on past actions.

Feature	Feedforward Neural Network (FNN)	Feedback Neural Network (RNN/FNN with Loops)
1. Data Flow	Moves in **one direction** (input → hidden → output)	Information flows **backward and forward** (has loops)
2. Memory	No memory of past inputs	Stores past information (good for sequential data)
3. Structure	No recurrent connections	Has recurrent connections (feedback loops)
4. Complexity	Simpler and easier to train	More complex and harder to train
5. Usage	Best for **static problems** (e.g., classification)	Best for **time-dependent problems** (e.g., speech recognition)
6. Training Time	Requires fewer computations, so training is faster	Requires more computations, so training takes longer
7. Backpropagation	Standard **backpropagation** is used for weight updates	Uses **Backpropagation Through Time (BPTT)** to handle sequences
8. Suitability	Works well for **structured data**	Works well for **sequential data**
9. Gradient Issues	Less prone to vanishing gradient problem	More prone to vanishing gradient problem
10. Applications	Image recognition, fraud detection, object detection	Time series prediction, natural language processing, speech recognition