Associative Memory Nets

Associative Memory Networks are a type of neural network designed to store and recall patterns or data based on associations. They are inspired by the way the human brain associates different pieces of information. The primary goal of AMNs is to retrieve a complete pattern when presented with a partial or noisy version of that pattern.

Key Characteristics:

1. **Pattern Storage and Recall**: AMNs can store multiple patterns and retrieve them based on partial or noisy inputs.
2. **Content-Addressable Memory**: Unlike traditional memory systems that use addresses to retrieve data, AMNs use the content itself to retrieve stored information.
3. **Robustness to Noise**: AMNs are designed to be robust to noisy or incomplete inputs, making them useful in scenarios where data may be corrupted or incomplete.

Use Cases of Associative Memory Networks:

1. **Pattern Recognition**:
   - **Image Recognition**: AMNs can be used to recognize images even when they are partially obscured or corrupted. For example, in facial recognition systems, AMNs can help identify a person even if part of their face is hidden or the image is blurry.
   - **Speech Recognition**: In speech recognition systems, AMNs can help in recognizing words or phrases even when the audio is noisy or contains gaps.
2. **Data Retrieval**:
   - **Database Querying**: AMNs can be used to retrieve information from large databases based on partial or fuzzy queries. This is particularly useful in scenarios where the exact query is not known, but a close approximation is available.
   - **Information Retrieval**: In search engines, AMNs can help retrieve relevant documents or web pages based on partial or noisy search terms.
3. **Error Correction**:
   - **Error-Correcting Codes**: AMNs can be used in communication systems to correct errors in transmitted data. By storing valid codewords, the network can retrieve the correct data even if the received signal is corrupted.
   - **Noise Reduction**: In signal processing, AMNs can be used to filter out noise from signals, such as in audio or image processing, by recalling the original, clean signal from a noisy input.
4. **Cognitive Modeling**:

- **Simulating Human Memory**: AMNs are used in cognitive science to model how humans store and recall information. They help in understanding the mechanisms of human memory and how associations are formed and retrieved.
- **Neurological Research**: AMNs are used to study neurological disorders related to memory, such as Alzheimer's disease, by simulating how memory functions and fails in the brain.

5. **Robotics**:
   - **Autonomous Navigation**: In robotics, AMNs can be used for navigation tasks where the robot needs to recognize its environment based on partial or noisy sensor data. The network can recall the correct path or action based on the current sensory input.
   - **Object Recognition**: Robots can use AMNs to recognize objects in their environment, even if the objects are partially obscured or viewed from different angles.
6. **Biometric Systems**:
   - **Fingerprint Recognition**: AMNs can be used in fingerprint recognition systems to match partial or noisy fingerprints to stored templates.
   - **Iris Recognition**: In iris recognition systems, AMNs can help in identifying individuals even when the iris image is partially obscured or of low quality.
7. **Medical Diagnosis**:
   - **Disease Diagnosis**: AMNs can be used in medical diagnosis systems to recall patterns associated with specific diseases based on partial or noisy patient data, such as symptoms or test results.
   - **Medical Imaging**: In medical imaging, AMNs can help in identifying abnormalities in images (e.g., X-rays, MRIs) even when the images are noisy or incomplete.

Types of Associative Memory Networks

1. **Autoassociative Memory**:
   - Stores and retrieves patterns based on partial or noisy input.
   - The input and output patterns are the same.
   - Used for tasks like noise reduction and pattern completion.
2. **Heteroassociative Memory**:
   - Stores and retrieves patterns where the input and output patterns are different.
   - Used for tasks like mapping input patterns to output patterns (e.g., image-to-text mapping).

Autoassociative Networks

Autoassociative networks are a type of associative memory network where the input and output patterns are the same. These networks are trained to store a set of patterns and can recall the complete pattern when presented with a partial or noisy version of it.

Architecture of Autoassociative Networks

1. **Input Layer**:
   - Receives the input pattern (e.g., a vector of features).
2. **Output Layer**:
   - Produces the output pattern, which is the same as the input pattern.
3. **Weights**:
   - The weights between the input and output layers are adjusted during training to store the patterns.

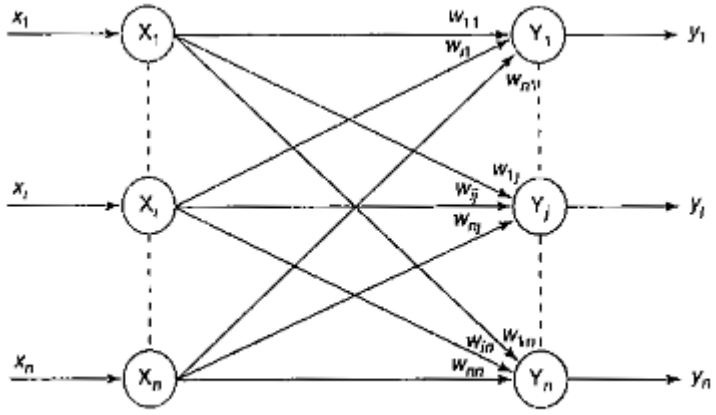


Figure 4-3 Architecture of autoassociative net.

Training Algorithm for Autoassociative Networks

The training process involves adjusting the weights to store the patterns in the network. The most common training algorithm for autoassociative networks is the **Hebbian Learning Rule**.

Hebbian Learning Rule:

- The Hebbian learning rule is based on the principle that "neurons that fire together, wire together."
- The weight update rule is:

  $$w_{ij} = w_{ij} + \eta \cdot x_i \cdot x_j$$

  where:
  - (w_{ij}) is the weight between neuron (i) and neuron (j),

- (\eta) is the learning rate,
- (x_i) and (x_j) are the activations of neurons (i) and (j), respectively.

Steps in Training:

1. **Initialize Weights**:
   - Initialize the weights to zero or small random values.
2. **Present Training Patterns**:
   - For each training pattern (x), update the weights using the Hebbian learning rule:

$$w_{ij} = w_{ij} + \eta \cdot x_i \cdot x_j$$

3. **Repeat**:
   - Repeat the process for all training patterns until the weights converge.

Recall Process in Autoassociative Networks

Once the network is trained, it can recall stored patterns when presented with a partial or noisy input.

Steps in Recall:

1. **Present Input Pattern**:
   - Present a partial or noisy version of the stored pattern to the input layer.
2. **Compute Output**:
   - Compute the output of the network using the current weights:

$$y_i = \sum_{j=1}^{n} w_{ij} \cdot x_j$$

3. **Iterate**:
   - Repeat the computation until the output stabilizes (i.e., the network converges to the stored pattern).

Use Cases of Autoassociative Networks

1. **Pattern Completion**:
   - Autoassociative networks can complete partial patterns. For example, if a part of an image is missing, the network can reconstruct the missing parts based on the stored patterns.
2. **Noise Reduction**:

- These networks can remove noise from input patterns. For example, if an image is corrupted with noise, the network can retrieve the original image by filtering out the noise.

3. **Data Retrieval**:
   - Autoassociative networks can retrieve stored data based on partial or noisy input. This is useful in applications like database retrieval and content-addressable memory.
4. **Image and Speech Recognition**:
   - These networks are used in image and speech recognition systems to recognize patterns even when the input is distorted or incomplete.

Advantages of Autoassociative Networks

1. **Robustness**:
   - Autoassociative networks are robust to noise and can retrieve stored patterns even from noisy or partial inputs.
2. **Simplicity**:
   - The architecture and training algorithm are simple and easy to implement.
3. **Efficiency**:
   - The recall process is fast and efficient, making these networks suitable for real-time applications.

Limitations of Autoassociative Networks

1. **Capacity**:
   - The number of patterns that can be stored in an autoassociative network is limited by the number of neurons and the correlation between patterns.
2. **Interference**:
   - If the stored patterns are too similar, the network may have difficulty distinguishing between them, leading to interference.
3. **Scalability**:
   - Autoassociative networks may not scale well to very large datasets or high-dimensional input spaces.

Summary

- Autoassociative networks are a type of associative memory network where the input and output patterns are the same.
- They are trained using the Hebbian learning rule, which adjusts the weights based on the correlation between input patterns.
- These networks are used for tasks like pattern completion, noise reduction, and data retrieval.
- While they are robust and efficient, their capacity and scalability are limited, especially for large and complex datasets.

Heteroassociative Memory Network

Heteroassociative memory networks are a type of associative memory network where the input and output patterns are different. These networks are used to map input patterns to output patterns, such as mapping images to text or words to meanings. Unlike autoassociative networks, heteroassociative networks do not require the input and output patterns to be the same.

Architecture of Heteroassociative Networks

The architecture of a heteroassociative memory network consists of two layers:

1. **Input Layer**:
   - Receives the input pattern (e.g., a vector of features).
2. **Output Layer**:
   - Produces the output pattern, which is different from the input pattern.
3. **Weights**:
   - The weights between the input and output layers are adjusted during training to store the mapping between input and output patterns.

Training Algorithm for Heteroassociative Networks

The training process involves adjusting the weights to store the mapping between input and output patterns. The most common training algorithm for heteroassociative networks is the **Hebbian Learning Rule**, similar to autoassociative networks.

Hebbian Learning Rule:

- The Hebbian learning rule is based on the principle that "neurons that fire together, wire together."
- The weight update rule is:

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

where:
  - (w_{ij}) is the weight between input neuron (i) and output neuron (j),
  - (\eta) is the learning rate,
  - (x_i) is the activation of input neuron (i),
  - (y_j) is the activation of output neuron (j).

Steps in Training:

1. **Initialize Weights**:
   - Initialize the weights to zero or small random values.
2. **Present Training Patterns**:
   - For each pair of input-output patterns ((x, y)), update the weights using the Hebbian learning rule:

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

3. **Repeat**:
   - Repeat the process for all training pairs until the weights converge.

Recall Process in Heteroassociative Networks

Once the network is trained, it can recall the output pattern when presented with an input pattern.

Steps in Recall:

1. **Present Input Pattern**:
   - Present an input pattern to the input layer.
2. **Compute Output**:
   - Compute the output of the network using the current weights:

$$y_j = \sum_{i=1}^{n} w_{ij} \cdot x_i$$

3. **Iterate**:
   - Repeat the computation until the output stabilizes (i.e., the network converges to the stored output pattern).

Use Cases of Heteroassociative Networks

1. **Pattern Mapping**:
   - Heteroassociative networks can map input patterns to output patterns. For example, mapping images to labels or words to meanings.
2. **Data Retrieval**:
   - These networks can retrieve output data based on input patterns. This is useful in applications like database retrieval and content-addressable memory.
3. **Image and Speech Recognition**:
   - Heteroassociative networks are used in image and speech recognition systems to recognize patterns and map them to corresponding outputs.
4. **Control Systems**:
   - These networks are used in control systems to map sensor inputs to control outputs.

Advantages of Heteroassociative Networks

1. **Flexibility**:
   - Heteroassociative networks can map different types of input and output patterns, making them flexible for various applications.
2. **Simplicity**:
   - The architecture and training algorithm are simple and easy to implement.
3. **Efficiency**:
   - The recall process is fast and efficient, making these networks suitable for real-time applications.

Limitations of Heteroassociative Networks

1. **Capacity**:
   - The number of patterns that can be stored in a heteroassociative network is limited by the number of neurons and the correlation between patterns.
2. **Interference**:
   - If the stored patterns are too similar, the network may have difficulty distinguishing between them, leading to interference.
3. **Scalability**:
   - Heteroassociative networks may not scale well to very large datasets or high-dimensional input spaces.

Summary

- Heteroassociative networks are a type of associative memory network where the input and output patterns are different.
- They are trained using the Hebbian learning rule, which adjusts the weights based on the correlation between input and output patterns.
- These networks are used for tasks like pattern mapping, data retrieval, and control systems.
- While they are flexible and efficient, their capacity and scalability are limited, especially for large and complex datasets.

Problems:

Example: Training an Autoassociative Network

Let’s consider a simple example where we train an autoassociative network to store two binary patterns:

1. Pattern 1: (x_1 = [1, -1, 1, -1])
2. Pattern 2: (x_2 = [-1, 1, -1, 1])

Step 1: Initialize Weights

- Initialize the weight matrix (W) to zeros:

$$W = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Step 2: Train with Pattern 1

- Update weights using Hebbian learning:

$$W = W + \eta \cdot x_1^T \cdot x_1$$

  Assuming $(\eta = 1)$:

$$W = \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

Step 3: Train with Pattern 2

- Update weights using Hebbian learning:

$$W = W + \eta \cdot x_2^T \cdot x_2$$

  Resulting weight matrix:

$$W = \begin{bmatrix} 2 & -2 & 2 & -2 \\ -2 & 2 & -2 & 2 \\ 2 & -2 & 2 & -2 \\ -2 & 2 & -2 & 2 \end{bmatrix}$$

Step 4: Recall

- To recall a pattern, present a partial or noisy input to the network and compute the output using the trained weights.

This example demonstrates how an autoassociative network can store and retrieve patterns using the Hebbian learning rule.

Example: Training a Heteroassociative Network

Let's consider a simple example where we train a heteroassociative network to store two pairs of binary patterns:

1. Pair 1:
   - Input Pattern: (x_1 = [1, -1, 1, -1])
   - Output Pattern: (y_1 = [1, -1])
2. Pair 2:
   - Input Pattern: (x_2 = [-1, 1, -1, 1])
   - Output Pattern: (y_2 = [-1, 1])

Step 1: Initialize Weights

- Initialize the weight matrix (W) to zeros:

$$W = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Step 2: Train with Pair 1

- Update weights using Hebbian learning:

$$W = W + \eta \cdot x_1^T \cdot y_1$$

Assuming $(\eta = 1)$:

$$W = \begin{bmatrix} 1 & -1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \end{bmatrix}$$

Step 3: Train with Pair 2

- Update weights using Hebbian learning:

$$W = W + \eta \cdot x_2^T \cdot y_2$$

Resulting weight matrix:

$$W = \begin{bmatrix} 2 & -2 \\ -2 & 2 \\ 2 & -2 \\ -2 & 2 \end{bmatrix}$$

Step 4: Recall

- To recall an output pattern, present an input pattern to the network and compute the output using the trained weights.

This example demonstrates how a heteroassociative network can store and retrieve patterns using the Hebbian learning rule.

Bidirectional Associative Memory (BAM)

Bidirectional Associative Memory (BAM) is a type of neural network designed by Bart Kosko in 1988. It is a hetero-associative memory model that can store pairs of patterns and recall them bidirectionally. BAM is capable of associating patterns from one set to another and vice versa, making it unique compared to other associative memory models.

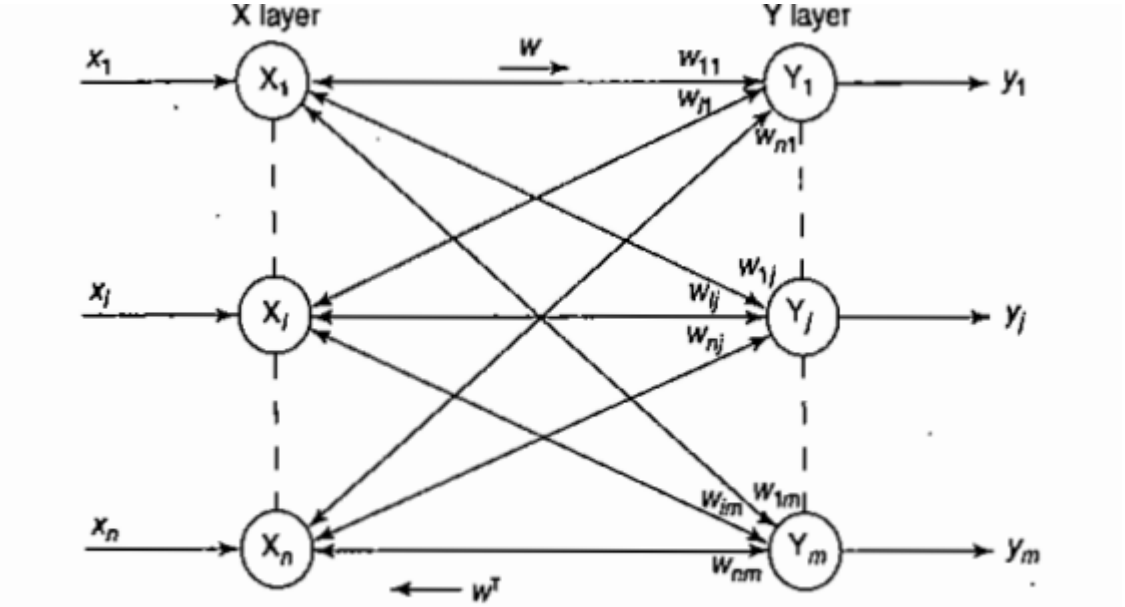


Figure 4-6 Bidirectional associative memory net.

Architecture of BAM

1. **Two Layers**:
   - BAM consists of two layers of neurons: **Layer X** (input layer) and **Layer Y** (output layer).
   - Neurons in both layers are fully connected, meaning every neuron in Layer X is connected to every neuron in Layer Y and vice versa.
2. **Weights**:
   - The connections between the layers are represented by a weight matrix W.
   - The weight matrix is designed to store associations between input and output patterns.
3. **Bidirectional Activation**:
   - BAM operates in both directions:
     - From X to Y: Given an input in X, BAM retrieves the associated output in Y.
     - From Y to X: Given an input in Y, BAM retrieves the associated output in X.
4. **Activation Function**:
   - Typically, a bipolar (or binary) activation function is used.

$$f(x) = \begin{cases} 1 & \text{if } x > \theta \\ 0 & \text{if } x \leq \theta \end{cases}$$

$$f(x) = \begin{cases} 1 & \text{if } x > \theta \\ -1 & \text{if } x \leq \theta \end{cases}$$

Training Algorithm for BAM

The training algorithm for BAM is based on **Hebbian learning**, which adjusts the weights to store associations between input and output patterns.

1. **Initialization**:
   - Initialize the weight matrix W to zero or small random values.
2. **Training Pairs**:
   - Given a set of training pairs (X^k, Y^k) where k=1,2,…,p and (X^k),and (Y^k) are bipolar vectors, update the weight matrix as follows: $W = W + (X^k)^T \cdot Y^k$
   - This is repeated for all training pairs.
3. **Stability**:
   - After training, the network should reach a stable state where the associations between X and Y are preserved.

Recall Process in BAM

1. **Forward Recall** (from X to Y):
   - Given an input X, compute the output Y as: $Y = f(X \cdot W)$
2. **Backward Recall** (from Y to X):
   - Given an input Y, compute the output X as: $X = f(Y \cdot W^T)$
3. **Iterative Recall**:
   - BAM can iteratively recall patterns until it reaches a stable state:
     - Update Y based on X.
     - Update X based on the new Y.
     - Repeat until no further changes occur.

Differences from Hetero-Associative and Auto-Associative Networks

1. **Hetero-Associative Networks**:
   - Hetero-associative networks map input patterns to different output patterns.
   - They are unidirectional: input X is mapped to output Y, but not vice versa.
   - Examples: Feedforward neural networks with supervised learning.
2. **Auto-Associative Networks**:
   - Auto-associative networks map input patterns to themselves (i.e., X is mapped back to X).
   - They are used for pattern completion or noise reduction.
   - Examples: Hopfield networks.
3. **BAM**:

- BAM is a **bidirectional hetero-associative network**.
- It maps input X to output Y and input Y to output X.
- It is designed for bidirectional recall, making it more versatile than hetero-associative networks.

Summary

- **BAM** is a bidirectional associative memory model that can store and recall pairs of patterns in both directions.
- It uses Hebbian learning to train the weight matrix for storing associations.
- Unlike **hetero-associative networks**, BAM allows bidirectional recall.
- Unlike **auto-associative networks**, BAM associates different patterns rather than mapping patterns to themselves.

BAM is particularly useful in applications where bidirectional associations are required, such as pattern recognition, optimization, and associative memory tasks.

Hopfield Network

The **Hopfield Network** is a type of **recurrent neural network** that serves as a form of **associative memory**. It was introduced by John Hopfield in 1982. The network is designed to store and retrieve patterns, making it useful for tasks like pattern completion, noise reduction, and optimization problems. Unlike feedforward networks, Hopfield networks have feedback connections, allowing them to exhibit dynamic behavior over time.

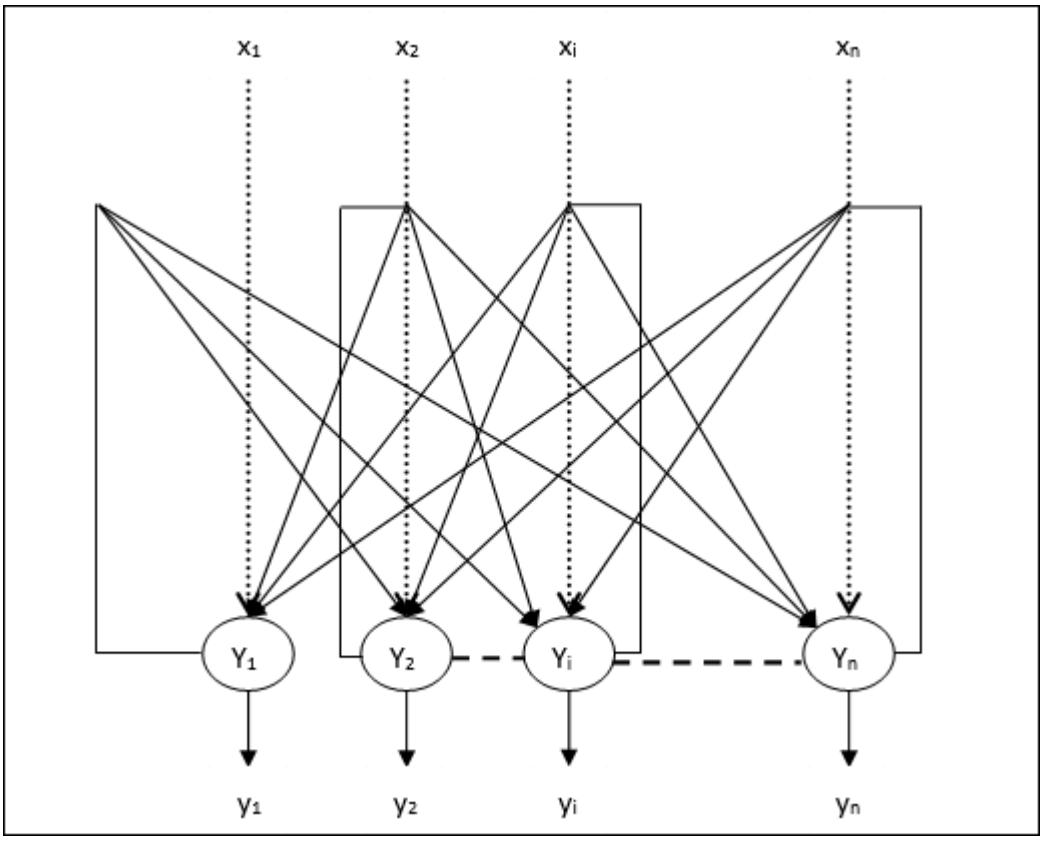
Architecture of Hopfield Network

The architecture of a Hopfield network consists of:

1. **Fully Connected Neurons**:
   - The network is composed of (N) neurons, where each neuron is connected to every other neuron.
   - The connections are bidirectional, meaning the weight from neuron (i) to neuron (j) is the same as the weight from neuron (j) to neuron (i) (i.e., ($w_{ij} = w_{ji}$)).
2. **No Self-Connections**:
   - Neurons do not connect to themselves (i.e., (w_{ii} = 0)).
3. **Binary or Bipolar States**:

- Each neuron has a binary state (e.g., (0) or (1)) or a bipolar state (e.g., (-1) or (1)).

4. **Symmetric Weight Matrix**:
   - The weight matrix (W) is symmetric (($W = W^T$)) and has zeros on the diagonal.
5. **Energy Function**:
   - The network is governed by an energy function, which ensures that the network evolves toward stable states (local minima).



Hopfield Network

The Hopfield Network is a type of **recurrent neural network (RNN)** introduced by John Hopfield in 1982. It is a fully connected, auto-associative memory model that stores patterns and retrieves them from partial or noisy inputs. The network is widely used for **pattern completion**, **noise reduction**, and **optimization tasks**.

Key Features of Hopfield Network

1. **Fully Connected Architecture**:
   - The network consists of a single layer of **fully connected neurons**.
   - Each neuron is connected to every other neuron, and the connections are **symmetric**

$$w_{ij} = w_{ji}$$

2. **Binary or Bipolar Neurons**:
   - Neurons typically operate with binary (0) or(1) or bipolar (-1 or 1) activations.
3. **Energy Function**:
   - The network has a **Lyapunov energy function** that ensures convergence to a stable state:

$$E = -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} x_i x_j$$

   - The network evolves to minimize this energy function.
4. **Recurrent Dynamics**:
   - Neurons update their states iteratively until the network reaches a stable state.

Training Mechanism

The training process in a Hopfield network involves storing patterns in the network by adjusting the weights. The most common training algorithm is based on the **Hebbian Learning Rule**.

Hebbian Learning Rule:

- The Hebbian learning rule is based on the principle that "neurons that fire together, wire together."
- For a set of (P) patterns $\{x^1, x^2, \ldots, x^P\}$, the weight matrix (W) is updated as:

$$w_{ij} = \frac{1}{N} \sum_{p=1}^{P} x_i^p \cdot x_j^p$$

  where:
  - x_i^p is the state of neuron (i) in pattern (p),
  - (N) is the number of neurons,
  - (w_{ij}) is the weight between neuron (i) and neuron (j).

Steps in Training:

1. **Initialize Weights**:
   - Initialize the weight matrix (W) to zeros.
2. **Store Patterns**:
   - For each pattern (x^p), update the weights using the Hebbian learning rule:

$$w_{ij} = w_{ij} + \frac{1}{N} \cdot x_i^p \cdot x_j^p$$

3. **Normalize Weights**:
    - Ensure that the weights are symmetric and have no self-connections ((w_{ii} = 0)).

Recall Process

Once the network is trained, it can retrieve stored patterns when presented with a partial or noisy input. The recall process involves updating the states of the neurons iteratively until the network converges to a stable state.

Steps in Recall:

1. **Present Input Pattern**:
    - Present a partial or noisy version of a stored pattern to the network.
2. **Update Neuron States**:
    - Update the state of each neuron asynchronously (one at a time) using the following rule:

$$s_i(t+1) = \text{sign}\left(\sum_{j=1}^{N} w_{ij} \cdot s_j(t)\right)$$

    where:
    - ($s_i(t)$) is the state of neuron (i) at time (t),
    - ($\text{sign}(\cdot)$) is the sign function, which outputs (+1) or (-1).
3. **Iterate Until Convergence**:
    - Repeat the update process until the network reaches a stable state (i.e., the states of the neurons no longer change).

Stability and Convergence

The Hopfield network is guaranteed to converge to a stable state due to its **energy function**. The energy function is defined as:

$$E = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} \cdot s_i \cdot s_j$$

Properties of the Energy Function:

1. **Boundedness**:

- The energy function is bounded below, meaning it cannot decrease indefinitely.

2. **Monotonicity**:
   - The energy function decreases monotonically as the network evolves over time.
3. **Local Minima**:
   - The network converges to local minima of the energy function, which correspond to stable states.

Stability:

- The network is stable because the energy function ensures that the system evolves toward a stable state.
- The stable states correspond to the stored patterns or their variants.

Use Cases of Hopfield Networks

1. **Pattern Completion**:
   - Hopfield networks can complete partial patterns. For example, if a part of an image is missing, the network can reconstruct the missing parts based on the stored patterns.
2. **Noise Reduction**:
   - These networks can remove noise from input patterns. For example, if an image is corrupted with noise, the network can retrieve the original image by filtering out the noise.
3. **Content-Addressable Memory**:
   - Hopfield networks can retrieve stored data based on partial or noisy input. This is useful in applications like database retrieval and associative memory systems.
4. **Optimization Problems**:
   - Hopfield networks can be used to solve optimization problems, such as the **Traveling Salesman Problem (TSP)**, by mapping the problem to the network's energy function.
5. **Image and Speech Recognition**:
   - These networks are used in image and speech recognition systems to recognize patterns even when the input is distorted or incomplete.

Advantages of Hopfield Networks

1. **Robustness**:

- Hopfield networks are robust to noise and can retrieve stored patterns even from noisy or partial inputs.

2. **Simplicity**:
   - The architecture and training algorithm are simple and easy to implement.
3. **Efficiency**:
   - The recall process is fast and efficient, making these networks suitable for real-time applications.
4. **Theoretical Guarantees**:
   - The network is guaranteed to converge to a stable state due to its energy function.

Limitations of Hopfield Networks

1. **Capacity**:
   - The number of patterns that can be stored in a Hopfield network is limited. The maximum capacity is approximately (0.15N), where (N) is the number of neurons.
2. **Spurious States**:
   - The network may converge to spurious states (local minima that do not correspond to stored patterns).
3. **Interference**:
   - If the stored patterns are too similar, the network may have difficulty distinguishing between them, leading to interference.
4. **Scalability**:
   - Hopfield networks may not scale well to very large datasets or high-dimensional input spaces.

Differences from Other Neural Networks

Feature	Hopfield Network	Other Neural Networks (e.g., Feedforward, BAM, RNN)
Architecture	Single layer, fully connected, recurrent.	Multi-layer, feedforward or recurrent.
Weight Symmetry	Symmetric weights ($w_{ij} = w_{ji}$).	Weights are not necessarily symmetric.
Update Mechanism	Asynchronous (one neuron at a time).	Synchronous (all neurons updated simultaneously).

Feature	Hopfield Network	Other Neural Networks (e.g., Feedforward, BAM, RNN)
Memory Type	Auto-associative (maps input to itself).	Hetero-associative (maps input to output) or auto-associative.
Energy Function	Defined, ensures convergence.	No explicit energy function.
Use Cases	Pattern completion, noise reduction, optimization.	Classification, prediction, sequence modeling.
Training	Hebbian learning (outer product rule).	Backpropagation, gradient descent.

Summary

- Hopfield networks are a type of recurrent neural network used for associative memory and optimization tasks.
- They are trained using the Hebbian learning rule, which adjusts the weights based on the correlation between input patterns.
- The network is governed by an energy function, which ensures stability and convergence to local minima.
- Use cases include pattern completion, noise reduction, content-addressable memory, and optimization problems.
- While they are robust and efficient, their capacity and scalability are limited, especially for large and complex datasets.

Solve this problem:

An ML engineer implemented Perceptron Learning Algorithm (PLA) using the following train data and computed the separating boundary as: x_1 +x_2 - 1 = 0.

When he applied this model on the validation samples, he noticed that the accuracy was low. He debugged the code and found that the separating boundary is not computed accurately.
(i). Determine which points are misclassified in the above samples.
(ii) Suggest appropriate corrections to the parameters of the separating boundary and write the new equation.

x_1	x_2	t
1	2	1
2	3	1
4	4.5	1
0	0	0
0.5	0	0
0.5	1	0