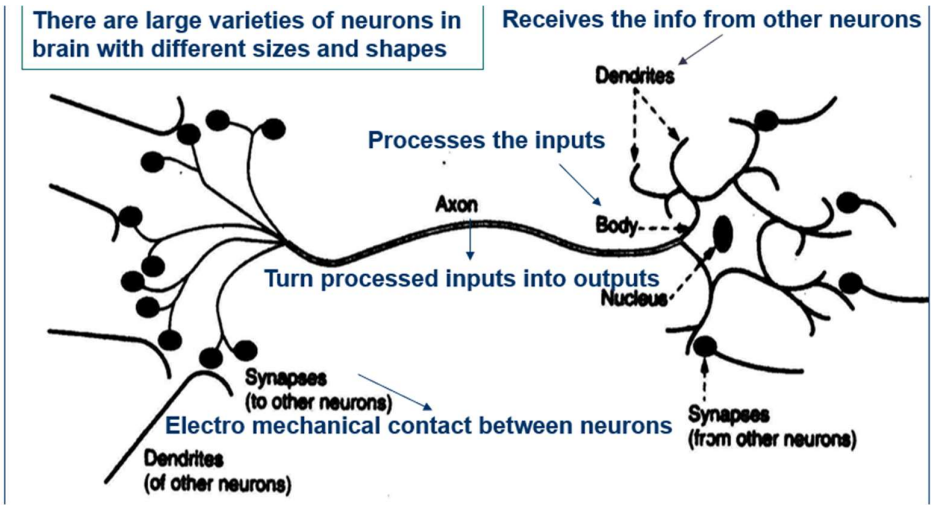
UNIT 1 & 2 NOTES

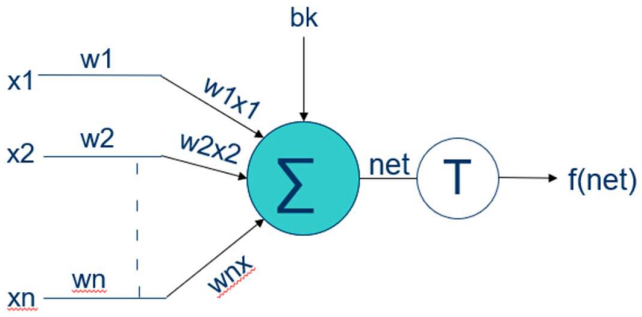


Structure of a Biological Neuron (6-marks answer – concise)

A biological neuron is the basic information-processing unit of the nervous system. In neural networks and encoders, the artificial neuron is modeled based on this biological structure. The main structural components are:

1. **Dendrites** – Branched structures that receive signals from other neurons through synapses. They act as input terminals.
2. **Cell Body (Soma)** – Contains the nucleus and cytoplasm; integrates/sums all incoming signals from dendrites.
3. **Nucleus** – Present inside the soma; controls metabolic activities and functioning of the neuron.
4. **Axon** – A long fiber that carries the processed electrical impulse away from the cell body toward other neurons.
5. **Myelin Sheath** – Fatty insulating layer around the axon (in many neurons) that increases speed of impulse conduction.
6. **Synapse** – Junction between axon terminal of one neuron and dendrite of another; enables electro-chemical signal transmission using neurotransmitters.

Link to neural networks: dendrites = inputs, synaptic strength = weights, soma = summation + activation, axon = output.



Artificial Neuron Model – McCulloch–Pitts (MCP) Neuron (6-marks answer – concise)

The McCulloch–Pitts neuron is a simple mathematical model of a biological neuron used in neural networks. It converts multiple inputs into a single output using weighted summation and thresholding.

1. **Inputs ($x_1, x_2, \ldots, x_n$)**
   External signals are applied to the neuron, analogous to dendrites of biological neurons.
2. **Weights ($w_1, w_2, \ldots, w_n$)**
   Each input is multiplied with a weight representing synaptic strength or importance of that input.
3. **Summation Function (∑)**
   The neuron computes the net input

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

where **b** is bias (or threshold adjustment).

4. **Threshold / Bias (bk)**
   A constant value used to shift the activation level; controls when the neuron should fire.
5. **Activation / Transfer Function T**
   The net value is passed through a threshold activation function producing output:
   - If net ≥ θ → neuron fires (output = 1)
   - If net < θ → neuron does not fire (output = 0)
6. **Output f(net)**
   Produces a binary decision (0/1) or (+1/−1) depending on activation function; used in logic gates and perceptrons.

Linear Separability

A problem is **linearly separable** when a **single straight line (or hyperplane)** can separate two classes completely.

- Classes can be divided by one straight boundary
- Examples: **AND, OR** logic functions
- **Single perceptron/single neuron can solve it**

Nonlinear Separability

A problem is **nonlinearly separable** when **no straight line** can separate classes.

- Requires **curved or complex boundary**
- Example: **XOR problem**
- **Single neuron cannot solve it** — needs multilayer network

---

Key limitation

Single MCP neuron/perceptron → **only solves linearly separable problems**

Multilayer neural networks → **solve nonlinear problems**

Steps of Hebbian learning rule:

1. Initialize all weights to small random values or zero.
2. Apply an input vector (x) to the neuron.
3. Compute the neuron output (y).
4. Update each weight using (\Delta w_i = \eta x_i y).
5. Add weight change to old weight: (w_i = w_i + \Delta w_i).
6. Repeat the above process for all training patterns.

Advantages

1. Simple learning rule; easy to implement
2. Biologically realistic and supports associative learning

Disadvantages

1. Weights may grow indefinitely (no upper limit)
2. Does not include error-correction; poor for complex tasks

Perceptron Learning Rule

Steps

1. Initialize weights and bias
2. Apply input vector and compute output y
3. Compare with target output t
4. If error exists, update

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

5. Repeat for all patterns until error becomes zero

Advantages

1. Simple and easy to implement
2. Guarantees convergence for linearly separable problems

Disadvantages

1. Works **only** for linearly separable data
2. Produces binary output only (hard-limited activation)

Delta Learning Rule (Gradient Descent Rule)

Steps

1. Initialize weights and bias

2. Apply inputs and compute output y
3. Compute error $e = t - y$
4. Update weights

$$\Delta w_i = \eta e x_i$$

5. Repeat for all patterns to minimize mean square error

Advantages

1. Can be used for **continuous-valued outputs**
2. Forms basis of **backpropagation** in multilayer networks

Disadvantages

1. May get stuck in **local minima**
2. Convergence can be slow and depends on learning rate

Widrow–Hoff Learning Rule (LMS Rule / ADALINE)

Steps

1. Initialize weights
2. Apply input and compute ADALINE output (before activation)
3. Calculate error using desired output
4. Update weights using

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

5. Repeat until mean squared error is minimized

Advantages

1. Minimizes **mean square error optimally**
2. More stable and accurate than perceptron rule

Disadvantages

1. Assumes linear activation (works for ADALINE only)
2. Computationally heavier due to error minimization

EBPA (Error Backpropagation Algorithm) is a **supervised learning algorithm** used to train **multilayer feedforward neural networks**. It works by **propagating inputs forward** through the network to produce outputs, comparing them with the **desired outputs**, computing the **error**, and then **back-propagating this error** layer by layer to adjust the connection weights using **gradient descent** so that future outputs become more accurate.

In short:
EBPA adjusts network weights in the direction that **reduces output error**, using **forward pass + backward pass**.

Key points

- used for multilayer perceptrons
- requires continuous activation functions
- minimizes mean square error
- based on gradient descent
- backbone of modern deep learning training

✓ Working of Error Backpropagation Algorithm (EBPA)

1. **Initialize weights**
   Random small values are assigned to all connection weights of the hidden and output layers.
2. **Forward pass – hidden layer**
   Inputs are applied and outputs of hidden layer neurons are computed using the current hidden layer weights.
3. **Forward pass – output layer**
   Using hidden layer outputs and output layer weights, outputs of output neurons are calculated.
4. **Compute error**
   The difference between desired output and actual network output is found.
5. **Backward pass – update output layer weights**
   Output-layer weights are updated first using the **gradient descent rule**.
6. **Update hidden layer weights**
   Hidden-layer weights are then updated using the **backpropagated error** and gradient descent.
7. **Iterate until convergence**
   Steps 2–6 are repeated until the total error becomes **minimum / within tolerance**.

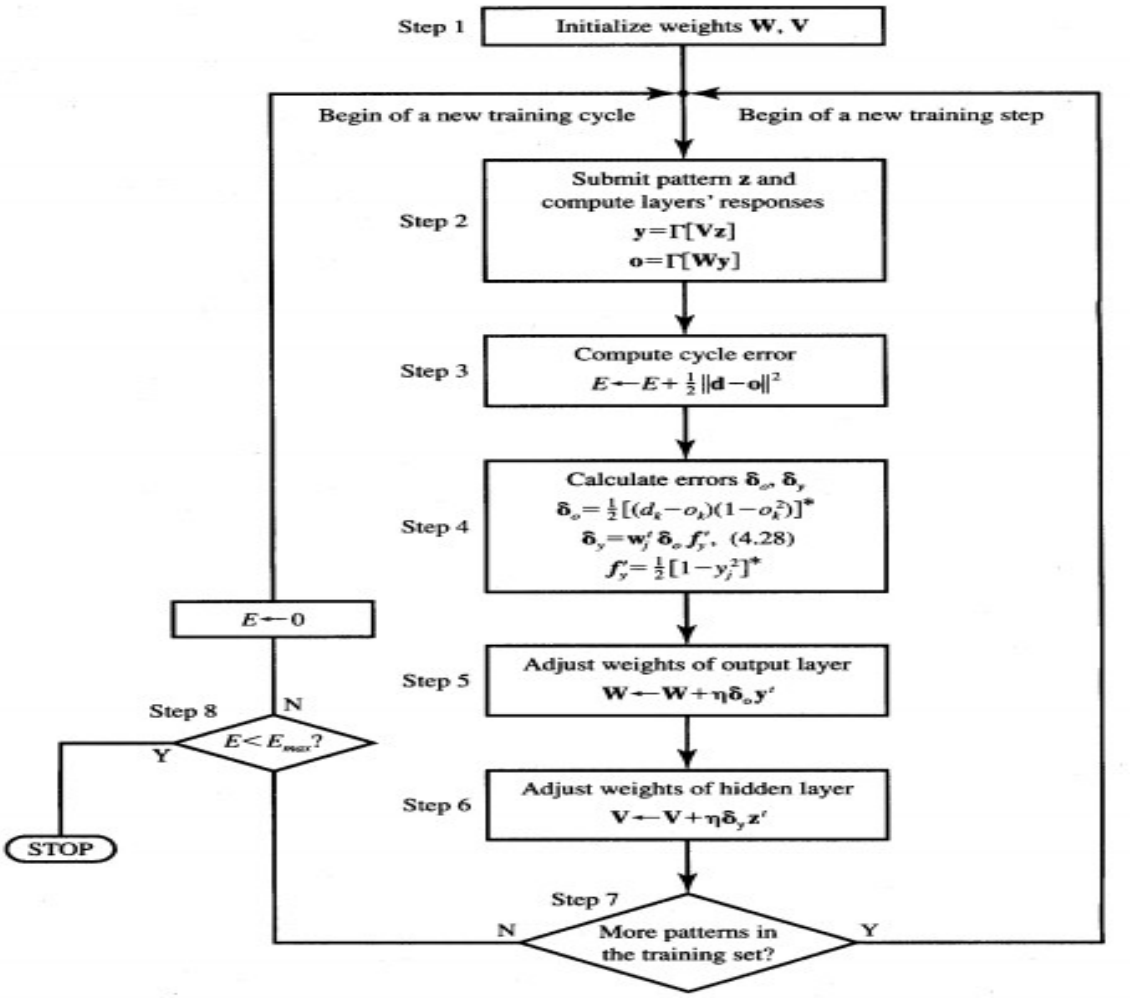
---

the **8 steps from the EBPA flow diagram, in short points**:

1. **Initialize weights** (W, V) with small random values.
2. **Present an input pattern** (z) to the network.
3. **Compute hidden layer outputs** (y = f(Vz)).
4. **Compute output layer outputs** (o = f(Wy)).
5. **Calculate error** for the pattern: (E = \frac{1}{2}\lVert d - o \rVert^2).
6. **Compute error signals** for output and hidden layers (δ terms).
7. **Update weights**
   - Output layer: (W = W + \eta \delta_o y^T)

- Hidden layer: (V = V + \eta \delta_y z^T)

8. **Check stopping condition**
   - if (E < E_{\text{max}}) → stop
   - else → repeat for remaining patterns/cycles



SETTING OF DESIGN CONSIDERATIONS AND PARAMETER VALUES

Initialization of weights

- Weights are initialized to **small random values**
- Common ranges: **(+0.5 to −0.5)** or **(+1 to −1)**
- Very large initial weights → **neuron saturation → slow learning**
- Unequal scaling of inputs may bias learning
- Goal: **similar effective input to each hidden node**

Frequency of weight updates

Per-pattern (online) updating

- Weights updated **after every sample**

- Requires **more computation**
- Suitable for **real-time / streaming applications**

Per-epoch (batch) updating

- Weights updated **after full training set**
- Computationally cheaper
- Not suitable for online systems

Choice of learning rate (η)

- Controls **step size of learning**
- Large η → **fast but oscillatory/unstable**
- Small η → **slow but stable**
- Typical range: **0.1 – 0.9**

Improvement strategies

- Start large then decrease
- Start small then increase when helpful
- Use 2nd derivative of error to adapt step size

Momentum

- Added term in weight update to smooth learning
- Prevents oscillation and helps escape **local minima**
- Typical momentum factor **α ≈ 0.9**
- Effective weight change depends on:
    - current gradient
    - previous weight change

Generalizability

- Network must perform well on **unseen test data**
- Overtraining reduces test accuracy
- Monitor **validation error**
- Stop when:
    - training error ↓ but test error ↑ (overfitting point)

Network size

- Too few nodes → **underfitting / weak learning**
- Too many nodes → **overfitting / high computation**
- Input nodes = number of input features
- Output nodes = number of classes
- Hidden nodes chosen by **trial and error**

Sample size

- More weights require **more training samples**

Thumb rule

$$\text{Samples} = 5\text{–}10 \times \text{number of weights}$$

Necessary condition (Baum–Haussler)

$$P > \frac{|W|}{1-a}$$

Sufficient condition

$$P > \frac{|W|}{a} \log\left(\frac{n}{1-a}\right)$$

Non-numeric inputs

- EBPA requires **numeric inputs**
- Non-numeric data must be **encoded** first:
  - one-hot encoding
  - label encoding
  - embedding representations
- Scaling/normalization improves learning

🧠 **Performance Evaluation of Error Backpropagation Algorithm (EBPA)**

🔹 **1. Processing Time**

The total training time of EBPA depends on:

- number of **weights to be trained**
- value of **learning rate (η)**
- number of **nodes** in the network
- number of **input samples**

- **complexity of activation function** used
- number of **epochs/iterations**

More nodes, more weights, and complex activation functions → longer training time.

🔹 2. Error Measurement

(a) Error per sample / per neuron

$$E = \frac{1}{2}(d_k - o_k)^2$$

where
d_k= desired output
o_k= actual output

(b) Error for all samples and neurons

$$E = \sum_{p=1}^{P} \sum_{k=1}^{K} \frac{1}{2}(d_k - o_k)^2$$

(c) Error based on Euclidean distance

$$E = \frac{1}{PK} \sum_{p=1}^{P} \sum_{k=1}^{K} \frac{1}{2}(d_k - o_k)^2$$

(d) Error based on Hamming distance

$$E = \frac{1}{PK} \sum_{p=1}^{P} \sum_{k=1}^{K} \frac{1}{2} \mid d_k - o_k \mid$$

🔹 3. Implementation

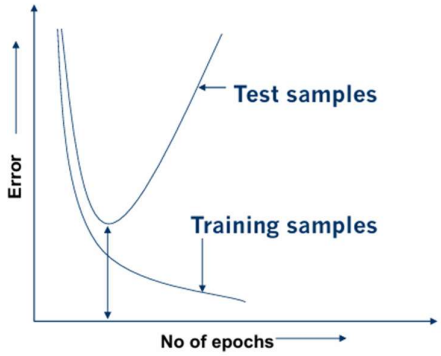
Hardware implementation	**Software implementation**
Very high speed	Slower

Hardware implementation	Software implementation
Cannot support all weight magnitudes	Can support any weight magnitude
Difficult to implement & debug for large networks	Easy to implement & debug
Suitable for small, high-speed systems	Suitable for large networks where speed is less critical

---

🔹 **4. Generalizability (Very Important)**

- Network must perform well on **unseen test data**, not only training data
- Excessive training → **overfitting**
- Overfitting effect:
  - training error ↓ continues decreasing
  - test error ↑ starts increasing



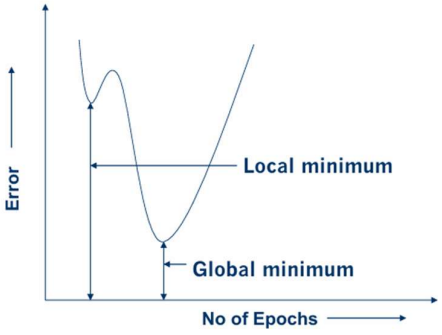
Solution

- continuously monitor test error
- stop training when:

test error increases even if training error decreases

(early stopping principle).

🚩 **Local Minima Problem in EBPA**



- **Ideal error curve** should be **single smooth parabola**
- In practice, the error surface contains **many local minima**
- Training using gradient descent may **get stuck** in one of these
- At a local minimum, **gradient = 0**, so algorithm **stops updating**
- Training is **assumed complete**, but **global minimum is not reached**

Goal:

Reach **global minimum**, not get trapped in local minima.

🧭 **Why it happens?**

- Error surface is **non-convex** in multilayer networks
- Weight space is **high-dimensional**
- Gradient descent follows **steepest local slope only**

✓ **Solution — Momentum Term**

To avoid getting stuck and reduce oscillations:

$$\Delta w(t) = \eta\delta + \alpha\Delta w(t-1)$$

Where

- η= learning rate
- α= **momentum factor** (typically **0.9**)
- $\Delta w(t-1)$= previous weight change

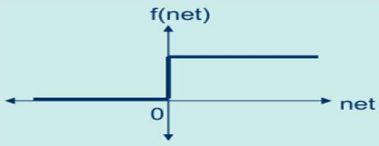
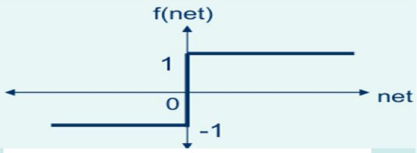
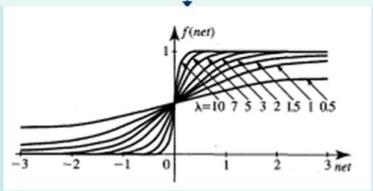
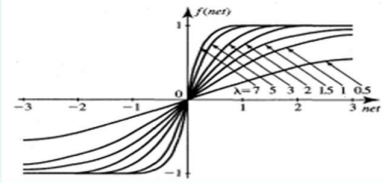
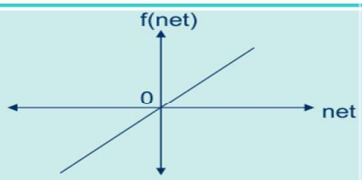
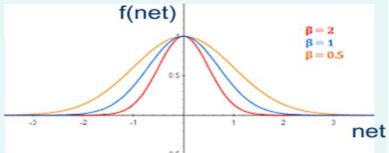
Effect of momentum

- Acts like **averaging** of error curve (without expensive computation)
- Helps system **roll past shallow local minima**
- **Prevents drastic oscillations**
- **Speeds convergence**

Intuition

- If gradient direction stays same → **momentum increases step size**
- If gradient direction flips → **momentum reduces step size**

Activation or Stochastic Functions

Activation Function	Graph	Equation
Discrete Unipolar/ Unipolar Binary/ Hard Limit Unipolar		$f(net) = 1; net \geq 0$ $0; net < 0$
Discrete bipolar/ Bipolar Binary/ Hard Limit Bipolar		$f(net) = 1; net \geq 0$ $-1; net < 0$
Continuous Unipolar/ Unipolar Sigmoid/ Soft limit unipolar		$f(net) = \frac{1}{1+\exp(-\lambda net)}$
Continuous bipolar/ Bipolar Sigmoid/ Soft limit bipolar		$f(net) = \frac{1-\exp(-\lambda net)}{1+\exp(-\lambda net)}$
Linear		$f(net) = net$
Gaussian		$f(net) = \frac{1}{\sqrt{2\Pi}\sigma}\ exp\ (\frac{-1}{2}(\frac{net-u}{\sigma}\ )\text{^}2)$ $\beta = \sigma$

Point of Difference	Supervised Learning	Unsupervised Learning	Reinforcement Learning
Availability of output	Target/desired output is provided	No target output is provided	No target output; reward/penalty given
Presence of teacher	Teacher present	No teacher	Environment acts as teacher via rewards
Data type used	Labeled data	Unlabeled data	Interaction data (state–action–reward)
Main goal	Minimize error between actual and desired output	Discover hidden structure/patterns in data	Maximize cumulative reward
Error signal	Explicit error signal computed	No error signal	Reward signal instead of error
Output type	Classification or regression	Clusters or feature groups	Policy or optimal action strategy
Learning method	Error-correction learning	Self-organization learning	Trial-and-error learning
Example applications	Handwriting recognition, spam detection	Market segmentation, anomaly detection	Game playing, robotics, self-driving cars
Algorithms used	Perceptron, Backpropagation, SVM, k-NN	K-means, PCA, SOM	Q-learning, TD learning, Policy gradients
Feedback type	Direct and explicit	None	Delayed and evaluative feedback

---

Simple Learning Algorithm

Definition

The **simple learning algorithm** is the earliest learning rule used for a single neuron.
Weights are **increased when the output is wrong** and **kept same when the output is correct**.

Explanation

- Inputs are applied to the neuron
- Output is compared with target output
- If output is correct → **no change in weights**
- If output is incorrect → **weights are adjusted** in the direction of error

- Process is repeated for all patterns until correct classification is obtained

---

Steps

1. Initialize weights and bias
2. Apply an input pattern
3. Compute neuron output
4. Compare with desired output
5. If output wrong → modify weights
6. If output correct → keep weights same
7. Repeat for all patterns

Weight update formula

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

✓ Advantages

1. **Simple and easy to implement** (no complex error backpropagation)
2. Learns **clusters automatically** without target output (unsupervised)
3. Requires **fewer computations** since only the winning neuron is updated

---

✕ Disadvantages

1. Sensitive to **initial weights** and may converge to poor clusters
2. Only one neuron wins — may lead to **dead neurons** (some neurons never learn)
3. Cannot handle **overlapping or complex class structures** effectively

---

Learning Vector Quantization (LVQ)

Definition

Learning Vector Quantization is a **supervised competitive learning algorithm** in which **prototype vectors (codebook vectors)** represent classes, and learning occurs by **moving prototypes toward correctly classified samples and away from misclassified samples.**

Explanation

- Each class is represented by one or more **reference/prototype vectors**
- An input vector is compared with all prototypes
- The **nearest prototype** is found (winner neuron)
- If winner's class = input class → move prototype **towards** input

- If winner's class ≠ input class → move prototype **away from** input
- Repeated until classification accuracy improves

Steps

1. Initialize prototype/codebook vectors for each class
2. Present an input sample with known class label
3. Find the prototype vector **closest** to the input (minimum distance)
4. If prototype class = input class → **move prototype towards input**
5. If prototype class ≠ input class → **move prototype away from input**
6. Repeat for all training samples

Update formulas

If correctly classified

$$w_{new} = w_{old} + \eta(x - w_{old})$$

If misclassified

$$w_{new} = w_{old} - \eta(x - w_{old})$$

Where

- w= prototype vector , x= input vector, η= learning rate

✓ Advantages

1. Simple and intuitive; easy to implement
2. Directly works with class labels (supervised competitive learning)
3. Decision boundaries are easy to interpret

✕ Disadvantages

1. Sensitive to **initial prototype selection**
2. May misclassify overlapping classes
3. Performance depends on **learning rate choice**

Self-Organizing Map (SOM)

Definition

Self-Organizing Map is an **unsupervised competitive learning algorithm** that maps **high-dimensional input data to a low-dimensional (usually 2-D) grid**, preserving the topological structure of the data.

Explanation

- No target/teacher is used (unsupervised)

- Each neuron on the grid has an associated **weight (prototype) vector**
- For each input, the **best matching unit (BMU)** (nearest neuron) is found
- BMU and its neighbors **move closer** to the input vector
- Over time, the map organizes itself so that **similar inputs lie close together**

Steps

1. Initialize weights of all neurons randomly
2. Present an input vector to the network
3. Compute distance between input vector and all neuron weight vectors
4. Select the **Best Matching Unit (BMU)** (minimum distance)
5. Update BMU and its neighboring neurons **towards the input**
6. Decrease learning rate and neighborhood size gradually
7. Repeat for all input samples over many iterations

Update formula

$$w_{new} = w_{old} + \eta\, h(t)\, (x - w_{old})$$

Where

w= weight vector, x= input vector, η= learning rate, $h(t)$= neighborhood function (larger at start, shrinks with time)

✓ Advantages

1. Works **without target output** (unsupervised)
2. Preserves **topology** — similar inputs map close together
3. Useful for **visualizing high-dimensional data**

✕ Disadvantages

1. No direct control on number of clusters formed
2. Training time may be high for large data
3. Quality depends on **neighborhood and learning-rate schedule**

Adaptive Resonance Theory (ART)

Definition

Adaptive Resonance Theory (ART) is an **unsupervised learning neural network** that performs **stable clustering** of input patterns while allowing the network to **learn new patterns without forgetting old ones**.

Explanation

- ART groups similar input patterns into **clusters (categories)**

- Each category has a **prototype vector**
- When a new input is presented:
    - it is compared with existing categories
    - if similarity is **above vigilance level**, it is assigned to that category
    - if not, a **new category is created**
- Thus, ART avoids **catastrophic forgetting** while still learning new data

---

Steps

1. Present an input vector to the network
2. Compare input with each category prototype
3. Select the **best matching category**
4. Check **vigilance parameter** (similarity test)
5. If similarity ≥ vigilance → **update that category**
6. If similarity < vigilance → **create new category**
7. Repeat for remaining inputs

Update idea (qualitative)

- For accepted category, weights move **towards input**
- Vigilance parameter controls:
    - **high vigilance** → many small clusters
    - **low vigilance** → few broad clusters

✓ Advantages

1. **Stable learning** — does not forget previously learned patterns
2. Can learn **new patterns dynamically**
3. Vigilance parameter allows **control over cluster granularity**

✕ Disadvantages

1. Performance highly sensitive to **vigilance parameter**
2. May create **too many categories** at high vigilance
3. Implementation is more complex than SOM/LVQ

Hopfield Network

✅ **Definition**

A Hopfield Network is a **recurrent neural network** used as **associative memory**, where stored patterns correspond to **stable states (attractors)** of the network. It is **fully connected**, has **symmetric weights**, and **no self-connections**.

✅ **Explanation**

- Every neuron is connected to every other neuron
- Neuron outputs are **binary or bipolar**
- When an input pattern (possibly noisy/incomplete) is applied, neurons update their states
- With each update, the **energy function decreases**
- Finally, the network settles into a **stable minimum-energy state**
- This stable state corresponds to one of the **stored patterns**

👉 Thus, Hopfield networks can **retrieve complete patterns from partial/noisy inputs** (content-addressable memory).

✅ **Steps (Working)**

1. Initialize symmetric weights; set diagonal weights to zero
2. Store patterns by adjusting weights (e.g., Hebbian rule)
3. Present an input pattern to the network
4. Compute neuron net input using weighted sum
5. Update neuron states **asynchronously or synchronously** using activation rule
6. Recompute energy; network keeps updating
7. Stop when neuron states **stop changing** (stable state reached)

✅ **Advantages**

1. Works as **associative memory** — retrieves stored patterns from noisy inputs
2. Guaranteed convergence to a **stable state** due to energy minimization
3. Simple implementation; binary/bipolar neurons and symmetric weights

---

✅ **Disadvantages**

1. **Low storage capacity** (~0.138N patterns for N neurons)
2. May converge to **spurious states/local minima** instead of correct memory
3. Not suitable for **large network sizes**; sensitive to noise and weight errors

---

Brain-State-in-a-Box (BSB) Network

✅ Definition

The **BSB network** is a **recurrent auto-associative neural network** used for **pattern storage and recall**, similar to Hopfield but with **continuous-valued neuron activations** constrained inside a **box-shaped region** (typically between –1 and +1).

✅ Explanation

- It stores patterns as **stable equilibrium states**
- Neurons are recurrently connected
- Neuron activations are **continuous** (not just binary)
- After each update, the state is **projected back** into a bounded region
- When a noisy or partial input is applied, the network trajectory **moves toward the nearest stored pattern**
- Convergence happens within a **state box** (hence the name)

👉 BSB therefore performs **auto-associative memory with continuous activation values**.

✅ Steps (Working)

1. Initialize weight matrix using stored training patterns
2. Present an input pattern (possibly noisy/incomplete)
3. Compute new state using:

$$x(t+1) = x(t) + \eta(Wx(t) + b - x(t))$$

4. **Clip/limit** each neuron output to lie within bounds (e.g., –1 to +1)
5. Repeat state updates iteratively
6. Stop when states stop changing → **stable pattern recalled**

✅ Advantages

1. Stores and recalls patterns like Hopfield, but with **continuous activations**
2. **Better convergence properties** than binary Hopfield networks
3. Handles **analog/noisy input data** more effectively

✅ Disadvantages

1. Training and weight design are **more complex** than Hopfield
2. Sensitive to **choice of parameters** (learning rate, bounds)
3. Storage capacity is still limited and network may converge to **spurious states**
