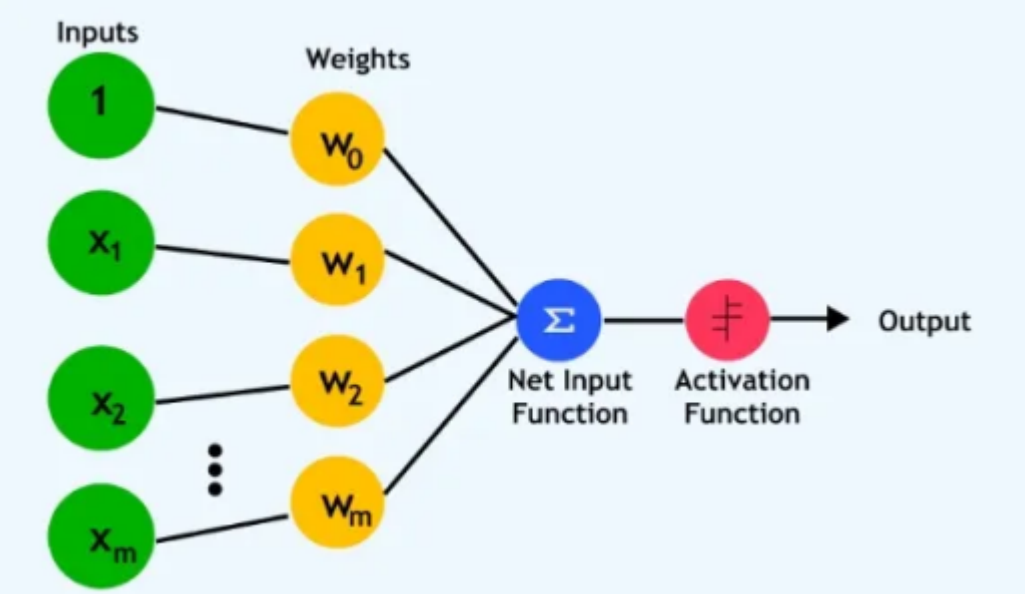
SC U2

23,24. Perceptron Training Algorithm for Single Output

A **Perceptron** is a simple neural network model used for **binary classification**.

It consists of an input layer, weights, bias, and an activation function (usually a step function). The **Perceptron Training Algorithm** helps adjust the weights to correctly classify input data.



Steps of the Perceptron Training Algorithm

1. **Initialize Weights and Bias:**
   - Assign small random values to weights ($W_1, W_2, ...W_n$) and bias (B).
2. **Repeat Until Convergence:**
   For each training sample $X = (x_1, x_2, ..., x_n)$:
   - **Compute Net Input:**

$$Z = W_1x_1 + W_2x_2 + ... + W_nx_n + B$$

   - **Apply Activation Function (Step Function):**

$$Y = \begin{cases} 1, & \text{if } Z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

   - **Update Weights and Bias (if prediction is incorrect):**

$$W_i = W_i + \eta \times (d - Y) \times x_i$$

$$B = B + \eta \times (d - Y)$$

     where:
     - η = Learning rate (small positive value)
     - d = Desired output
     - Y = Predicted output
3. **Repeat Until All Inputs Are Correctly Classified** ↓

Example Calculation

Given:

- Learning rate $\eta = 0.1$
- Initial Weights: $W_1 = 0.2, W_2 = -0.3$
- Bias: $B = 0.1$
- Training Data:

x_1	x_2	Output d
1	0	1
0	1	0

Step 1: Compute Net Input for (1,0)

$$Z = (0.2 \times 1) + (-0.3 \times 0) + 0.1 = 0.3$$

Since $Z \geq 0$, output $Y = 1$ (correct, no update).

Step 2: Compute Net Input for (0,1)

$$Z = (0.2 \times 0) + (-0.3 \times 1) + 0.1 = -0.2$$

Since $Z < 0$, output $Y = 0$ (correct, no update).

✅ If misclassification occurs, weights and bias are ad ted accordingly until convergence.

Key Features of Perceptron Training Algorithm✓ Works for linearly separable problems (e.g., AND, OR gates).

✓ **Simple and efficient** for basic binary classification.

❌ **Fails for non-linearly separable problems** (e.g., XOR gate)

25. Testing Algorithm for Perceptron Network

Once a perceptron is trained, the **testing algorithm** is used to classify new input data based on learned weights. The steps are as follows:

Steps of the Perceptron Testing Algorithm

1. **Input the test sample**
   - Given a test sample $X = (x_1, x_2, ..., x_n)$, where x_i are the feature values.
2. **Compute the Net Input**
   - Calculate the weighted sum using the learned weights W_i and bias B:

$$Z = W_1x_1 + W_2x_2 + ... + W_nx_n + B$$

3. **Apply Activation Function**
   - Use a **step activation function** to determine the output:

$$Y = \begin{cases} 1, & \text{if } Z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

4. **Classify the Output**
   - If $Y = 1$, classify into Class 1.
   - If $Y = 0$, classify into Class 2.

Example

Given:

- Trained **weights**: $W_1 = 0.5, W_2 = -0.4$
- **Bias**: $B = 0.2$
- **Test input**: $X = (1, 1)$

Step 1: Compute Net Input

$$Z = (0.5 \times 1) + (-0.4 \times 1) + 0.2$$

$$Z = 0.5 - 0.4 + 0.2 = 0.3$$

Step 2: Apply Activation Function

Since $Z \geq 0$, output $Y = 1$ (Class 1).

26. Adaline (Adaptive Linear Neuron) Network

Adaline (Adaptive Linear Neuron) is a type of **single-layer artificial neural network** developed by **Bernard Widrow and Marcian Hoff in 1960**.

It is an improvement over the **Perceptron** because it uses a **linear activation function** and **gradient descent** for weight updates.

Key Features of Adaline Network

✅ Uses **weighted sum of inputs** (like Perceptron).

✅ Employs a **linear activation function**.

✅ Updates weights based on **Mean Squared Error (MSE)**.

✅ Uses **Gradient Descent** to minimize error.

Structure of Adaline Network

1. Input Layer

- Takes multiple input features **($X_1, X_2, \ldots, Xn$)**.
- Each input has an **associated weight ($W_1, W_2, \ldots, Wn$)**.
- A **bias term (b)** is added for flexibility.

2. Summation Function

- Computes the **net input (Z)**:

  $Z=W1X1+W2X2+...+WnXn+bZ$

3. Activation Function (Linear Function)

- Unlike Perceptron, Adaline does **not** use a step function.
- Uses an **identity function**:

  $Y=Z$

4. Weight Update (Using LMS Rule)

- **Error Calculation**:

  $E = d - Y$

  where $ddd$is the desired output.
- **Weight Update Formula (Gradient Descent Rule)**:

  $Wi = Wi + \eta \times Xi \times E$

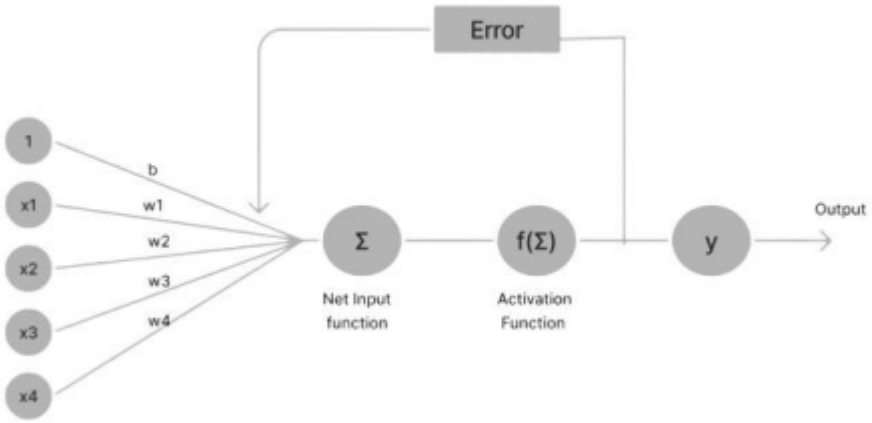
  where η\eta$\eta$is the**learning rate**.

Step-by-Step Working of Adaline

1. **Initialize weights** (Wi) and bias (b) to small random values.
2. **Compute net input**:

   $Z = W1X1 + W2X2 + ... + WnXn + bZ$
3. **Calculate output**: $Y = Z$(linear activation).
4. **Compute error**: $E = d - Y$.
5. **Update weights** using **LMS Rule**.
6. **Repeat** steps **2-5** until error is minimized.

Diagram of Adaline Network



Key Differences Between Adaline and Perceptron

Feature	Adaline	Perceptron
Activation Function	**Linear (Y = Z)**	**Step Function**
Learning Rule	**LMS (Least Mean Square)**	**Perceptron Rule**
Error Calculation	Uses **continuous error**	Uses **discrete misclassification**
Convergence	**More stable**	Can oscillate
Application	Regression & Classification	Only Classification

Advantages of Adaline

✅ Works **better** than Perceptron for **linearly separable** problems.

✅ Can be extended to **multi-layer networks (MADALINE)**.

✅ Uses **Mean Squared Error (MSE)** for smoother learning.

❌ **Limitation**: Cannot solve **non-linear** problems like XOR.

27, 28. Testing Algorithm for Adaline (Adaptive Linear Neuron) Network

Once the **Adaline network is trained**, we can use the following **testing algorithm** to classify new inputs.

Step-by-Step Testing Algorithm

1. **Input the test data**
   - Provide new inputs $X = (X_1, X_2, ..., X_n)$ to the trained Adaline network.
2. **Compute the Net Input**
   - Use the trained weights $W = (W_1, W_2, ..., W_n)$ and bias b to calculate:

$$Z = W_1X_1 + W_2X_2 + ... + W_nX_n + b$$

3. **Apply Activation Function**
   - Since Adaline uses a **linear activation function**:

$$Y = Z$$

   - However, for classification tasks, a **threshold function** is applied:

$$Y = \begin{cases} +1, & \text{if } Z \geq 0 \\ -1, & \text{if } Z < 0 \end{cases}$$

4. **Output the Predicted Class**
   - If $Y = 1$, classify as **positive class**.
   - If $Y = -1$, classify as **negative class**.
5. **Repeat for all test samples.**

Example Calculation

Given:

- Trained weights: $W_1 = 0.5, W_2 = -0.3$, bias $b = 0.2$.
- Test input: $X = (1, 2)$.

Step 1: Compute Net Input

$$Z = (0.5 \times 1) + (-0.3 \times 2) + 0.2$$

$$Z = 0.5 - 0.6 + 0.2 = 0.1$$

Step 2: Apply Activation Function

Since $Z = 0.1 \geq 0$, output $Y = 1$ (Positive class).

Key Takeaways

✅ Adaline **tests new inputs** using learned weights.
✅ Uses **linear summation** and a **threshold function** for classification.
✅ More stable than the **Perceptron** as it minimizes error continuously.

29. Backpropagation Network (BPN) – Explanation with Diagram

Backpropagation (Backprop) is a supervised learning algorithm used to train **multi-layer neural networks**. It adjusts weights using the **gradient descent** method to minimize errors.

Structure of Backpropagation Network (BPN)

A typical **Backpropagation Neural Network (BPN)** consists of:

1. **Input Layer:** Accepts feature values.
2. **Hidden Layer(s):** Processes inputs using weighted connections and activation functions.
3. **Output Layer:** Produces the final prediction.
4. **Weights & Bias:** Adjusted to minimize prediction errors.

Working of Backpropagation Algorithm

1. Forward Propagation:

- Input is passed through the network.
- Each neuron computes:

  $Z=WX+B$

- Activation function is applied to compute output YY.

2. Compute Error (Loss Function):

- Error is calculated as the difference between actual (dd) and predicted (YY) values:

  Error=d−Y
- Common loss functions: **Mean Squared Error (MSE), Cross-Entropy Loss**.

3. Backward Propagation (Weight Adjustment):

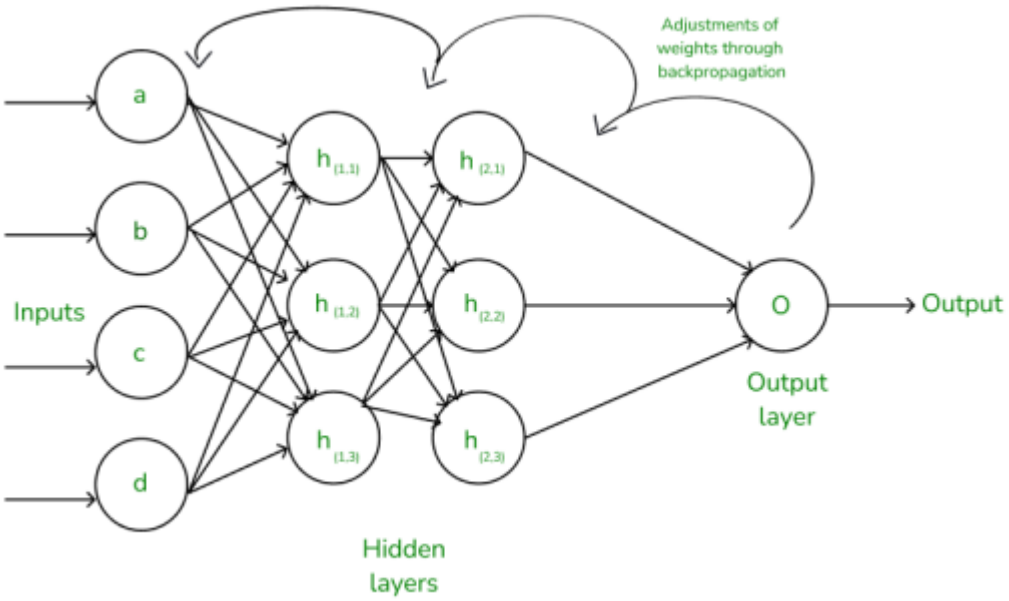
- Error is propagated backward using the **chain rule of differentiation**.
- Partial derivatives of the loss function are computed to update weights:

  W new = W old− η × ∂E/∂W

  where **η\eta**= learning rate.

4. Repeat Until Convergence:

- Steps 1-3 are repeated until the error is minimized.



Key Features of Backpropagation:

✓ **Efficient Learning**: Improves accuracy by minimizing errors.

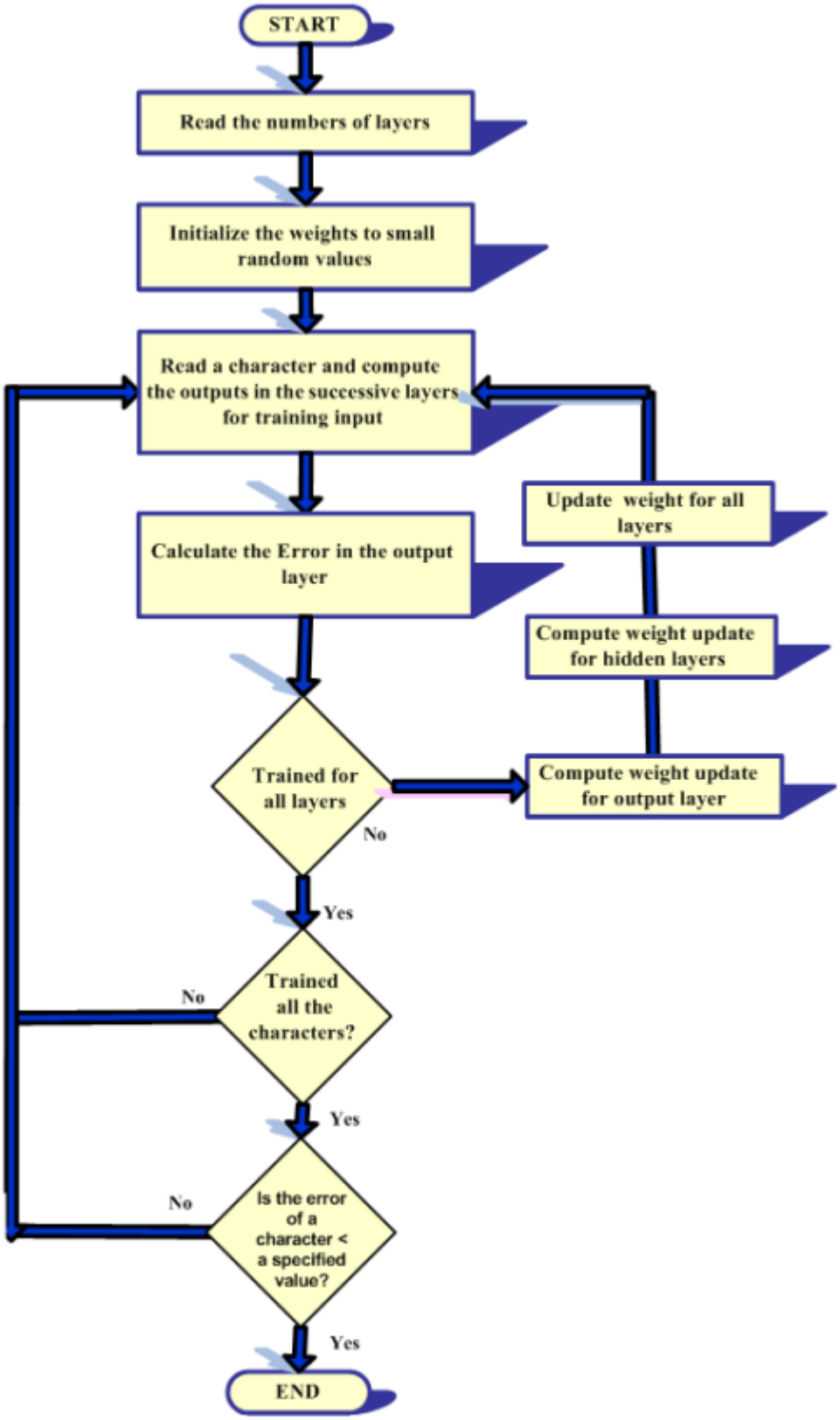
✓ **Works for Deep Networks**: Essential for training multi-layer perceptrons (MLP).

✓ **Gradient Descent Optimization**: Ensures optimal weight adjustments.

❌ **Slow for Large Datasets**: May require techniques like **batch normalization**.

Would you like a numerical example for better understanding? 😊

30 . flowchart of back prop



31. Importance of Training in Neural Networks

Training is a crucial process in neural networks as it enables the model to learn from data and make accurate predictions. The **goal of training** is to adjust weights and biases to minimize errors and improve performance.

Why is Training Important?

1. **Pattern Recognition & Learning**
    - Helps the neural network learn patterns, relationships, and trends from data.
    - Example: A neural network trained on handwriting data can recognize handwritten digits.
2. **Error Minimization**
    - Uses techniques like **gradient descent** and **backpropagation** to minimize prediction errors.
    - The loss function (e.g., **Mean Squared Error (MSE)**) is used to measure how well the model is performing.
3. **Generalization to New Data**
    - A well-trained model can accurately classify unseen data, improving **real-world usability**.
    - Prevents **overfitting**, where the model memorizes training data instead of learning patterns.
4. **Weight Optimization**
    - Training adjusts **weights and biases** so the network can correctly classify inputs.
    - Uses optimization algorithms like **SGD (Stochastic Gradient Descent)** and **Adam** to update weights.
5. **Improves Decision-Making**
    - Training allows the network to make better, more informed decisions by understanding complex data structures.
    - Example: A trained **medical diagnosis network** can assist doctors in identifying diseases.

Training Process in Neural Networks

1. **Initialize Weights & Biases** (Random values).
2. **Forward Propagation** (Compute outputs using input data).
3. **Calculate Error** (Compare predictions with actual values).
4. **Backpropagation** (Adjust weights using gradient descent).
5. **Repeat Until Convergence** (Until the error is minimized).

Conclusion

- ✅ **Essential for Model Performance** – Without training, a neural network cannot learn.
- ✅ **Reduces Error & Increases Accuracy** – Training refines the model for better predictions.
- ✅ **Improves Generalization** – Ensures the model works well on unseen data.

32. Difference Between Perceptron Learning and Backpropagation

Perceptron Learning and Backpropagation are two different learning algorithms used to train neural networks.

The main differences lie in their **network structure, learning approach, and applicability**.

Comparison Table: Perceptron Learning vs. Backpropagation

Feature	Perceptron Learning	Backpropagation
Applicable Network	Single-layer perceptron (SLP)	Multi-layer perceptron (MLP)
Learning Type	Supervised learning	Supervised learning
Error Calculation	Uses **perceptron rule**, which updates weights only for misclassified examples.	Uses **gradient descent** to minimize error continuously.
Weight Update	$W=W+\eta\times(d-Y)\times X$ W = W + \eta \times (d - Y) \times X $W=W+\eta\times(d-Y)\times X$	$W=W-\eta\times\partial E\partial WW = W - \eta \times \frac{\partial E}{\partial W}$ $W=W-\eta\times\partial W\partial E$
Activation Function	Step function (Binary classification)	Sigmoid, ReLU, Tanh (supports non-linearity)
Capability	Works **only for linearly separable problems** (e.g., AND, OR).	Works for **both linear and non-linear problems** (e.g., XOR).
Error Minimization	Stops adjusting weights once classification is correct.	Uses iterative optimization (backpropagation) to minimize error.
Training Speed	Fast and simple	Slower but more effective for complex problems
Network Depth	**Single-layer only**	**Multiple hidden layers supported** (Deep Learning)

35. Difference Between ADALINE and Perceptron

ADALINE (Adaptive Linear Neuron) and the **Perceptron** are both single-layer neural networks used for classification, but they differ in their **learning rule, activation function, and error calculation**.

Key Differences Between ADALINE and Perceptron

Feature	ADALINE (Adaptive Linear Neuron)	Perceptron
Activation Function	**Linear Function** (Identity function)	**Step Function** (Threshold-based)
Learning Rule	Uses **Least Mean Square (LMS) Rule** (Gradient Descent)	Uses **Perceptron Learning Rule**
Error Calculation	Based on **continuous error** (before activation)	Based on **misclassified samples** (after activation)
Weight Update	Updates weights using **Mean Squared Error (MSE)**	Updates only for misclassified samples
Convergence	More stable; **gradient-based learning**	May oscillate or fail for certain cases
Classification	Can approximate **real-valued functions**	Only works for **linearly separable** problems
Thresholding	Decision is made **after training** using thresholding	Decision is made **within the training process**
Application	Used in **Regression & Classification**	Used only in **Classification**

---

38. Types of Learning Rules in Neural Networks

A **learning rule** (or **learning algorithm**) is a method used by neural networks to **update weights** based on input data and expected output. The main types of learning rules are:

1. **Hebbian Learning Rule**
2. **Perceptron Learning Rule**
3. **Delta Learning Rule**
4. **Competitive Learning Rule**

5. **Outstar Learning Rule**
6. **Boltzmann Learning Rule**
7. **Error-Correction Learning Rule**

Explanation of Specific Learning Rules

1. Competitive Learning Rule

- **Concept:** Neurons **compete** with each other to be activated. The **winning neuron** strengthens its weights, while others weaken.
- **Mechanism:**
  - All neurons receive input.
  - The neuron with the highest response (winner) updates its weights.
  - Other neurons remain unchanged or weaken.
- **Used In: Self-organizing maps (SOMs), clustering algorithms.**
- **Formula:**

---

2. Delta Learning Rule (Widrow-Hoff Rule)

- **Concept:** Adjusts weights to minimize the difference between actual output and desired output using **Gradient Descent**.
- **Mechanism:**
  - Calculates **error**:

    E=d−YE = d - YE=d−Y
  - Updates weight using:
- **Used In: Adaline networks, regression problems.**
- **Advantage:** Ensures smooth weight updates, unlike Perceptron.

3. Outstar Learning Rule

- **Concept:** Strengthens connections **from a single neuron to multiple neurons**.
- **Mechanism:**
  - A central neuron influences **all** connected neurons.
  - Weights are updated using:
- **Used In: Pattern association, Kohonen networks.**

Key Takeaways

✅ **Competitive Learning** → Neurons compete to learn.
✅ **Delta Rule** → Uses **Gradient Descent** to minimize error.
✅ **Outstar Learning** → Strengthens connections **from one neuron to many neurons**.

---

39. Illustration of Learning Rules in Neural Networks

Here's a breakdown of the three learning rules with diagrams and explanations:

---

1. Hebbian Learning Rule

📌 **Concept**:

- "Neurons that fire together, wire together."
- If two neurons activate simultaneously, the **connection strengthens**.
- Weight update formula:

  $W_{ij} = W_{ij} + \eta X_i Y_j$

  where X_i is the input, Y_j is the output, and η the learning rate.

📌 **Diagram:**

- **Weights increase** when input & output are both positive.

📌 **Used In**: Associative memory, unsupervised learning.

---

2. Perceptron Learning Rule

📌 **Concept**:

- Weights are updated only for **misclassified samples**.
- Uses a **step activation function** for binary classification.
- Weight update formula:

  Wi=Wi+η(d−Y)Xi

  where ddis the desired output,YYis the actual output.

📌 **Diagram:**

- **If output matches target → no change.**
- **If misclassified → update weights.**

📌 **Used In**: Simple classification (AND, OR functions).

---

3. Correlation Learning Rule

📌 **Concept**:

- Weights are updated based on **correlation between input and output**.
- If XiX_iandYjY_jare correlated, their weight increases.
- Weight update formula:

  Wij=ηXiYjW_{ij} = \eta X_i Y_j

  where η\etais the learning rate.

📌 **Diagram:**

- **Weights increase** when input-output values correlate.

📌 **Used In**: Associative memory, pattern recognition.

Key Differences

Learning Rule	Weight Update Formula	Key Idea	Used In
Hebbian	$Wij=Wij+\eta XiYjW_{ij} = W_{ij} + \eta X_i Y_j$	Strengthens frequently activated connections	Associative memory
Perceptron	$Wi=Wi+\eta(d-Y)XiW_{i} = W_{i} + \eta (d - Y) X_i$	Updates only misclassified weights	Simple classifiers
Correlation	$Wij=\eta XiYjW_{ij} = \eta X_i Y_j$	Increases weights based on correlation	Pattern recognition

Would you like a **Python example** for any of these? 😊