Adaline

Adaline Neural Network (Adaptive Linear Neuron)

The **Adaline (Adaptive Linear Neuron)** is a single-layer neural network introduced by Bernard Widrow and Ted Hoff in 1960. It is an improvement over the **Perceptron** and is primarily used for binary classification tasks. Adaline uses a linear activation function and employs the **Least Mean Squares (LMS)** algorithm for training, which minimizes the mean squared error between the predicted and actual outputs.

Key Features of Adaline

1. **Linear Activation Function**:
   - Adaline uses a **linear activation function** instead of a step function (as in the Perceptron).
   - The output is continuous and not binary, making it suitable for regression tasks as well.
2. **Error Minimization**:
   - Adaline minimizes the **mean squared error (MSE)** between the predicted output and the target output during training.
3. **Weight Update Rule**:
   - Weights are updated using the **gradient descent** algorithm, which adjusts weights in the direction of the negative gradient of the error function.
4. **Bias Term**:
   - Adaline includes a bias term to account for the intercept in the linear model.

Architecture of Adaline

- **Input Layer**: Receives input features $(x_1, x_2, \ldots, x_n)$.
- **Weights**: Each input is associated with a weight $(w_1, w_2, \ldots, w_n)$.
- **Bias**: A bias term (b) is added to the weighted sum.
- **Output**: The output (y) is computed as:
  $[y = \sum_{i=1}^{n} w_i x_i + b]$
- **Activation Function**: The output is passed through a linear activation function (identity function).

Training Process of Adaline

The training process involves adjusting the weights and bias to minimize the mean squared error (MSE) between the predicted output and the target output.

Steps in Training:

1. **Initialize Weights and Bias**:
   - Initialize weights $(w_1, w_2, \ldots, w_n)$ and bias (b) to small random values or zeros.
2. **Compute the Net Input**:
   - For each training example ((x, t)), compute the net input (z):
     [$z = \sum_{i=1}^{n} w_i x_i + b$]
3. **Compute the Predicted Output**:
   - The predicted output (y) is the same as the net input (z) (since the activation function is linear):
     [$y = z$]
4. **Compute the Error**:
   - Calculate the error (e) as the difference between the target output (t) and the predicted output (y):
     [$e = t - y$]
5. **Update Weights and Bias**:
   - Update the weights and bias using the **gradient descent** rule:
     [$w_i \leftarrow w_i + \eta \cdot e \cdot x_i$]
     [$b \leftarrow b + \eta \cdot e$]
     where (η) is the **learning rate**.
6. **Repeat**:
   - Repeat steps 2–5 for all training examples in the dataset.
   - Continue iterating until the error converges to a minimum or a predefined number of epochs is reached.

Mathematical Formulation

The goal of Adaline is to minimize the **mean squared error (MSE)**:
[$\text{MSE} = \frac{1}{2} \sum_{i=1}^{m} (t_i - y_i)^2$]]
where:

- (m) is the number of training examples,
- (t_i) is the target output for the (i)-th example,
- (y_i) is the predicted output for the (i)-th example.

The weight update rule is derived from the gradient of the MSE with respect to the weights and bias:
$[\Delta w_i = -\eta \frac{\partial \mathrm{MSE}}{\partial w_i} = \eta \cdot e \cdot x_i]$
$[\Delta b = -\eta \frac{\partial \mathrm{MSE}}{\partial b} = \eta \cdot e]$

Advantages of Adaline

1. **Continuous Output**:
   - The linear activation function provides continuous output, making Adaline suitable for regression tasks.
2. **Error Minimization**:
   - Adaline minimizes the mean squared error, leading to better performance compared to the Perceptron.
3. **Simple and Efficient**:
   - The training process is straightforward and computationally efficient.

Limitations of Adaline

1. **Linear Separability**:
   - Like the Perceptron, Adaline can only solve linearly separable problems.
2. **Sensitivity to Learning Rate**:
   - The choice of learning rate (η) is crucial; a high learning rate may cause oscillations, while a low learning rate may slow down convergence.

Comparison with Perceptron

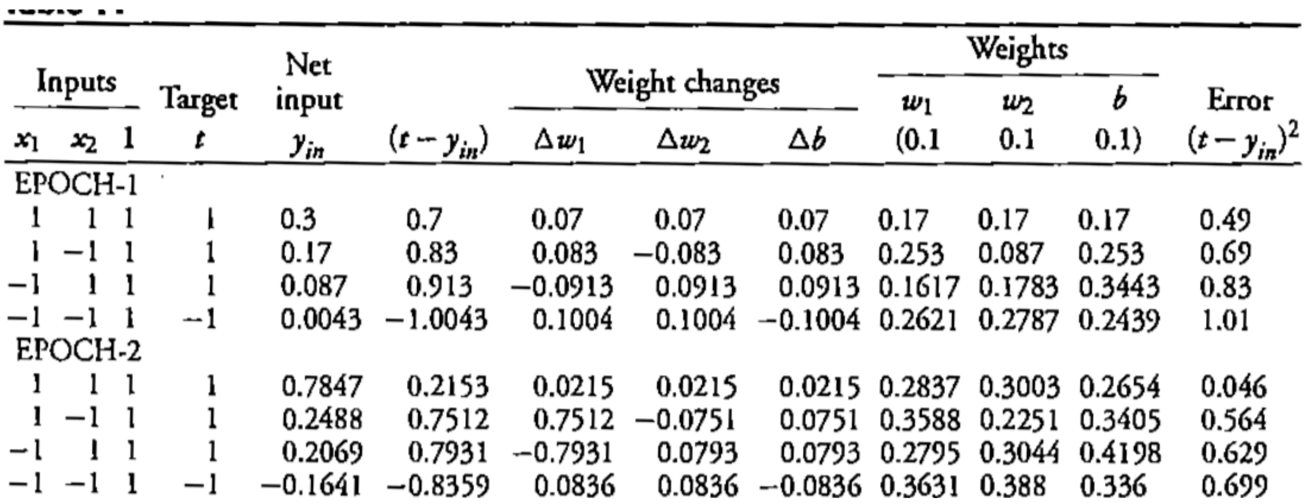
Feature	Adaline	Perceptron
Activation Function	Linear	Step Function
Error Minimization	Mean Squared Error (MSE)	Binary Classification Error
Output	Continuous	Binary (0 or 1)
Training Algorithm	Least Mean Squares (LMS)	Perceptron Learning Rule

---

Summary (Adaline)

- Adaline is a single-layer neural network that uses a linear activation function and minimizes the mean squared error during training.
- It is trained using the **Least Mean Squares (LMS)** algorithm, which updates weights and bias based on the gradient of the error function.
- Adaline is suitable for both classification and regression tasks but is limited to linearly separable problems.
- Its continuous output and error minimization make it more powerful than the Perceptron for certain applications.

Problem: Implement OR function with Bipolar input and targets using Adaline Network.
Assume initial weights and bias to be 0.1 and learning rate of 0.1



Inputs			Target	Net input		Weight changes			Weights			Error
x_1	x_2	1	t	y_{in}	$(t - y_{in})$	Δw_1	Δw_2	Δb	w_1 (0.1	w_2 0.1	b 0.1)	$(t - y_{in})^2$
EPOCH-1												
1	1	1	1	0.3	0.7	0.07	0.07	0.07	0.17	0.17	0.17	0.49
1	−1	1	1	0.17	0.83	0.083	−0.083	0.083	0.253	0.087	0.253	0.69
−1	1	1	1	0.087	0.913	−0.0913	0.0913	0.0913	0.1617	0.1783	0.3443	0.83
−1	−1	1	−1	0.0043	−1.0043	0.1004	0.1004	−0.1004	0.2621	0.2787	0.2439	1.01
EPOCH-2												
1	1	1	1	0.7847	0.2153	0.0215	0.0215	0.0215	0.2837	0.3003	0.2654	0.046
1	−1	1	1	0.2488	0.7512	0.7512	−0.0751	0.0751	0.3588	0.2251	0.3405	0.564
−1	1	1	1	0.2069	0.7931	−0.7931	0.0793	0.0793	0.2795	0.3044	0.4198	0.629
−1	−1	1	−1	−0.1641	−0.8359	0.0836	0.0836	−0.0836	0.3631	0.388	0.336	0.699

Backpropagation Network

The **Backpropagation Network** is a type of artificial neural network (ANN) that uses the **backpropagation algorithm** for training. It is widely used for supervised learning tasks, such as classification and regression. The backpropagation algorithm adjusts the weights of the network by propagating the error backward from the output layer to the input layer, minimizing the error between the predicted and actual outputs.

Key Concepts of Backpropagation

1. **Feedforward Process**:

    - Input data is passed through the network layer by layer, from the input layer to the output layer.
    - Each neuron computes a weighted sum of its inputs, applies an activation function, and passes the result to the next layer.
2. **Error Calculation**:
    - The error is calculated as the difference between the predicted output and the target output.
3. **Backward Propagation**:
    - The error is propagated backward through the network, and the weights are updated using gradient descent to minimize the error.
4. **Activation Functions**:
    - Common activation functions include **sigmoid**, **tanh**, and **ReLU**.

Architectures of Backpropagation Networks

Backpropagation networks can have different architectures depending on the number of layers and the connections between neurons. Some common architectures include:

1. Single-Layer Perceptron (SLP):

- Consists of only one layer of neurons (output layer).
- Suitable for linearly separable problems.
- Limited in its ability to solve complex tasks.

2. Multilayer Perceptron (MLP):

- Consists of an input layer, one or more hidden layers, and an output layer.
- Capable of solving non-linearly separable problems.
- The most common architecture for backpropagation networks.

3. Deep Neural Networks (DNN):

- A type of MLP with multiple hidden layers.
- Used for complex tasks such as image recognition, natural language processing, etc.
- Requires large amounts of data and computational resources.

4. Recurrent Neural Networks (RNN):

- Designed for sequential data (e.g., time series, text).
- Neurons have connections that form directed cycles, allowing information to persist over time.

- Variants include **LSTM** (Long Short-Term Memory) and **GRU** (Gated Recurrent Unit).

5. **Convolutional Neural Networks (CNN)**:

- Designed for grid-like data (e.g., images).
- Uses convolutional layers to extract spatial features.
- Commonly used in computer vision tasks.

Learning Algorithm: Backpropagation

The backpropagation algorithm is used to train the network by minimizing the error between the predicted and actual outputs. The steps are as follows:

1. **Initialize Weights**:

- Initialize the weights and biases to small random values.

2. **Forward Pass**:

- For each training example, compute the output of the network:
  - Compute the weighted sum of inputs for each neuron.
  - Apply the activation function to get the neuron's output.
  - Pass the output to the next layer.

3. **Compute Error**:

- Calculate the error at the output layer using a loss function (e.g., mean squared error, cross-entropy).

4. **Backward Pass**:

- Compute the gradient of the error with respect to each weight using the chain rule.
- Propagate the error backward through the network to update the weights.

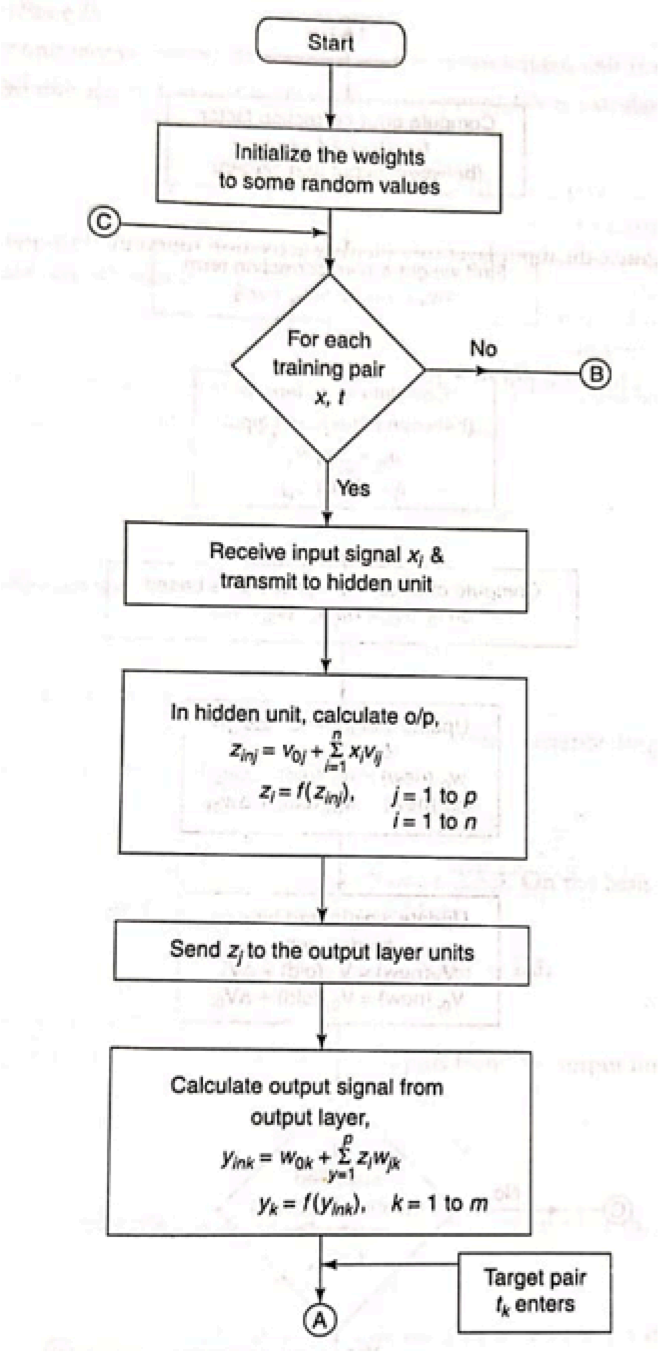
5. **Update Weights**:

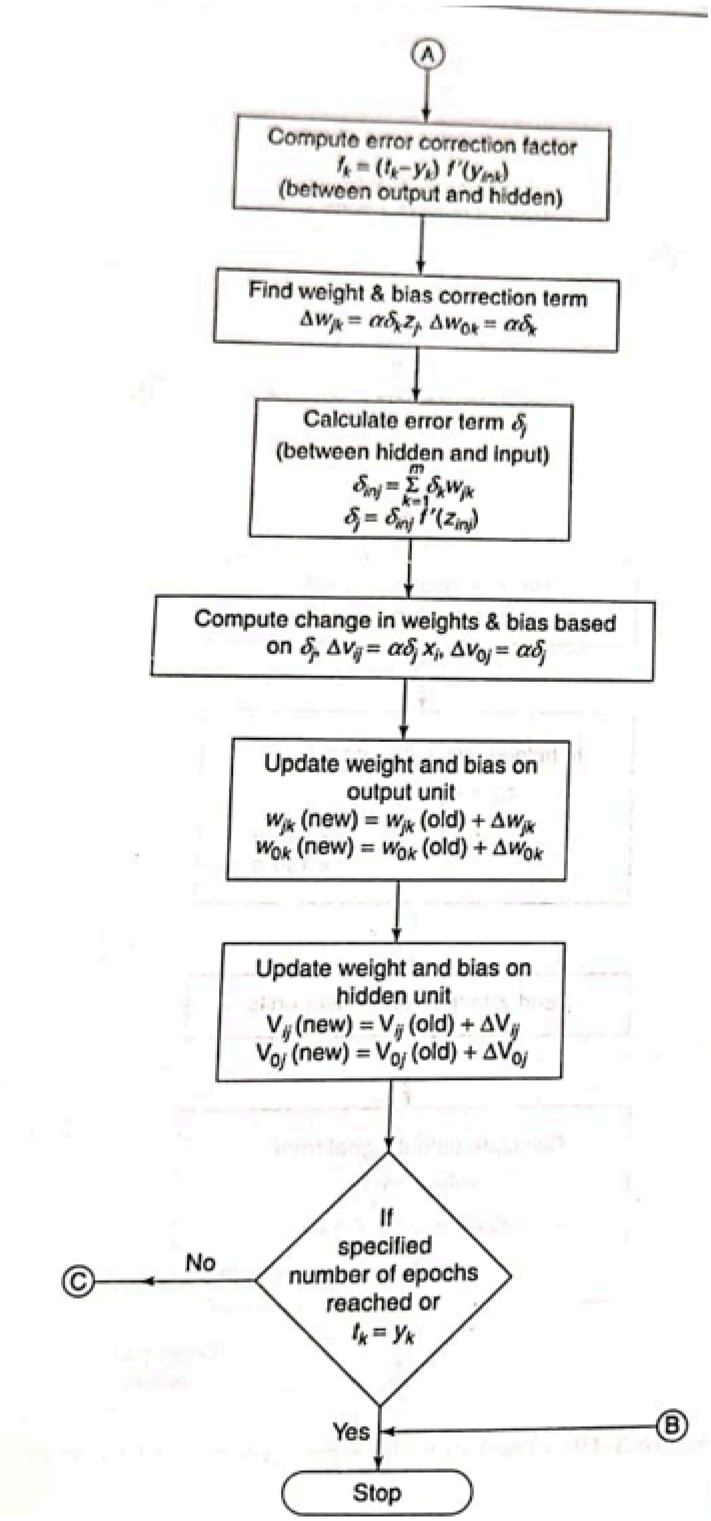
- Adjust the weights and biases using gradient descent:
  $[w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}]$
  where (η) is the learning rate and ($\frac{\partial E}{\partial w_{ij}}$) is the gradient of the error with respect to the weight.

6. **Repeat**:

- Repeat the forward pass, error computation, backward pass, and weight update for all training examples.
- Continue iterating until the error converges to a minimum or a predefined number of epochs is reached.

Training Process


Start
Initialize the weights to some random values
C
For each training pair x, t
No
B
Yes
Receive input signal x_i & transmit to hidden unit
In hidden unit, calculate o/p,
$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$
$z_j = f(z_{inj})$, $j = 1$ to p
$i = 1$ to n
Send z_j to the output layer units
Calculate output signal from output layer,
$y_{ink} = w_{0k} + \sum_{y=1}^{p} z_j w_{jk}$
$y_k = f(y_{ink})$, $k = 1$ to m
Target pair t_k enters
A


A
Compute error correction factor
$f_k = (t_k - y_k)\, f'(y_{ink})$
(between output and hidden)
Find weight & bias correction term
$\Delta w_{jk} = \alpha \delta_k z_j,\ \Delta w_{0k} = \alpha \delta_k$
Calculate error term δ_j
(between hidden and input)
$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$
$\delta_j = \delta_{inj} f'(z_{inj})$
Compute change in weights & bias based on δ_j, $\Delta v_{ij} = \alpha \delta_j x_i$, $\Delta v_{0j} = \alpha \delta_j$
Update weight and bias on output unit
w_{jk} (new) = w_{jk} (old) + Δw_{jk}
w_{0k} (new) = w_{0k} (old) + Δw_{0k}
Update weight and bias on hidden unit
V_{ij} (new) = V_{ij} (old) + ΔV_{ij}
V_{0j} (new) = V_{0j} (old) + ΔV_{0j}
If specified number of epochs reached or $t_k = y_k$
No
C
Yes
B
Stop

Variants of Backpropagation

1. **Stochastic Gradient Descent (SGD)**:
   - Updates weights after each training example.
   - Faster convergence but more noisy updates.
2. **Batch Gradient Descent**:
   - Updates weights after processing the entire training dataset.
   - Slower convergence but more stable updates.
3. **Mini-Batch Gradient Descent**:
   - Updates weights after processing a small batch of training examples.

- Balances the speed of SGD and the stability of batch gradient descent.

4. **Momentum**:
   - Adds a momentum term to the weight update to accelerate convergence and reduce oscillations.
5. **Adaptive Learning Rate Methods**:
   - Methods like **AdaGrad**, **RMSProp**, and **Adam** adapt the learning rate during training to improve convergence.

Advantages of Backpropagation Networks

1. **Universal Approximation**:
   - MLPs with one hidden layer can approximate any continuous function given sufficient neurons.
2. **Flexibility**:
   - Can be applied to a wide range of tasks, including classification, regression, and pattern recognition.
3. **Scalability**:
   - Can be scaled to handle large datasets and complex architectures (e.g., deep learning).

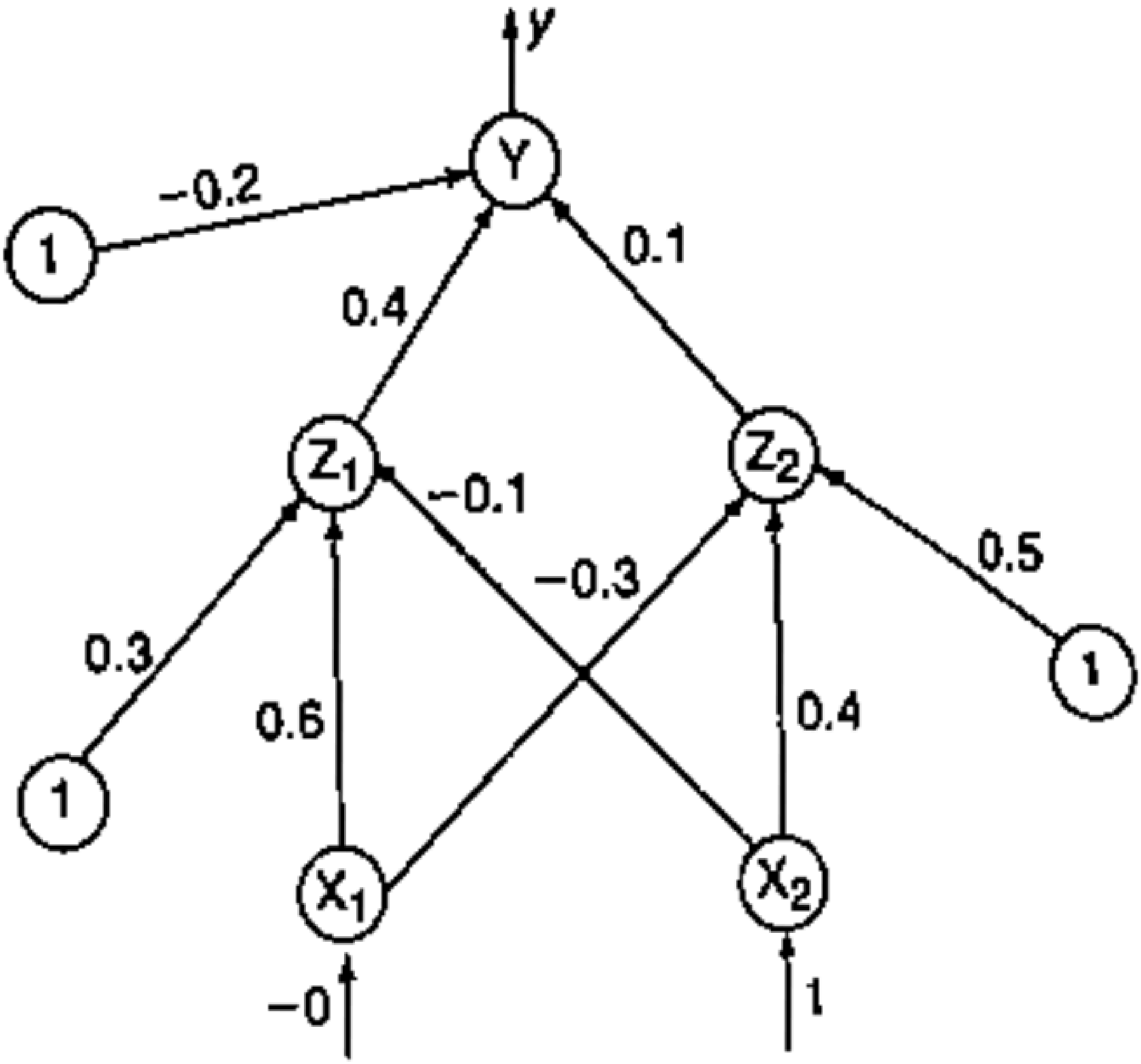
Limitations of Backpropagation Networks

1. **Computational Complexity**:
   - Training deep networks requires significant computational resources and time.
2. **Overfitting**:
   - Complex networks may overfit the training data, especially with limited data.
3. **Vanishing/Exploding Gradients**:
   - In deep networks, gradients may become very small (vanishing) or very large (exploding), hindering training.
4. **Local Minima**:
   - The error surface may have many local minima, causing the algorithm to converge to suboptimal solutions.

Summary

- Backpropagation networks are a powerful class of neural networks trained using the backpropagation algorithm.
- They can have various architectures, including single-layer perceptrons, multilayer perceptrons, deep neural networks, recurrent neural networks, and convolutional neural networks.
- The backpropagation algorithm minimizes the error by propagating it backward through the network and updating the weights using gradient descent.
- While backpropagation networks are versatile and scalable, they face challenges such as computational complexity, overfitting, and vanishing/exploding gradients.

Problem: Using back-propagation network , find the new weights for the net shown below. It is presented with the input pattern [0,1] and the target output is [1]. Use a learning rate of 0.25 and binary sigmoidal activation function.



Solution Set:

$[v_{12}\ v_{22}\ v_{02}] = [-0.3\ 0.4\ 0.5]$ and $[w_1\ w_2\ w_0] = [0.4\ 0.1\ -0.2]$, and the learning rate is $\alpha = 0.25$. Activation function used is binary sigmoidal activation function and is given by

$$f(x) = \frac{1}{1+e^{-x}}$$

Given the output sample $[x_1, x_2] = [0, 1]$ and target $t = 1$,

- Calculate the net input: For z_1 layer

$$\begin{aligned} z_{in1} &= v_{01} + x_1 v_{11} + x_2 v_{21} \\ &= 0.3 + 0 \times 0.6 + 1 \times -0.1 = 0.2 \end{aligned}$$

For z_2 layer

$$\begin{aligned} z_{in2} &= v_{02} + x_1 v_{12} + x_2 v_{22} \\ &= 0.5 + 0 \times -0.3 + 1 \times 0.4 = 0.9 \end{aligned}$$

Applying activation to calculate the output, we obtain

$$z_1 = f(z_{in1}) = \frac{1}{1+e^{-z_{in1}}} = \frac{1}{1+e^{-0.2}} = 0.5498$$

$$z_2 = f(z_{in2}) = \frac{1}{1+e^{-z_{in2}}} = \frac{1}{1+e^{-0.9}} = 0.7109$$

- Calculate the net input entering the output layer. For y layer

$$\begin{aligned} y_{in} &= w_0 + z_1 w_1 + z_2 w_2 \\ &= -0.2 + 0.5498 \times 0.4 + 0.7109 \times 0.1 \\ &= 0.09101 \end{aligned}$$

Applying activations to calculate the output, we obtain

$$y = f(y_{in}) = \frac{1}{1+e^{-y_{in}}} = \frac{1}{1+e^{-0.09101}} = 0.5227$$

- Compute the error portion δ_k:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

Now

$$f'(y_{in}) = f(y_{in})[1 - f(y_{in})] = 0.5227[1 - 0.5227]$$
$$f'(y_{in}) = 0.2495$$

This implies

$$\delta_1 = (1 - 0.5227)\ (0.2495) = 0.1191$$

Find the changes in weights between hidden and output layer:

$$\begin{aligned} \Delta w_1 &= \alpha\delta_1 z_1 = 0.25 \times 0.1191 \times 0.5498 \\ &= 0.0164 \\ \Delta w_2 &= \alpha\delta_1 z_2 = 0.25 \times 0.1191 \times 0.7109 \\ &= 0.02117 \\ \Delta w_0 &= \alpha\delta_1 = 0.25 \times 0.1191 = 0.02978 \end{aligned}$$

- Compute the error portion δ_j between input and hidden layer ($j = 1$ to 2):

$$\boxed{\delta_j = \delta_{inj} f'(z_{inj})}$$

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

$$\delta_{inj} = \delta_1 w_{j1} \quad [\because \text{only one output neuron}]$$

$$\Rightarrow \delta_{in1} = \delta_1 w_{11} = 0.1191 \times 0.4 = 0.04764$$

$$\Rightarrow \delta_{in2} = \delta_1 w_{21} = 0.1191 \times 0.1 = 0.01191$$

Error, $\delta_1 = \delta_{in1} f'(z_{in1})$

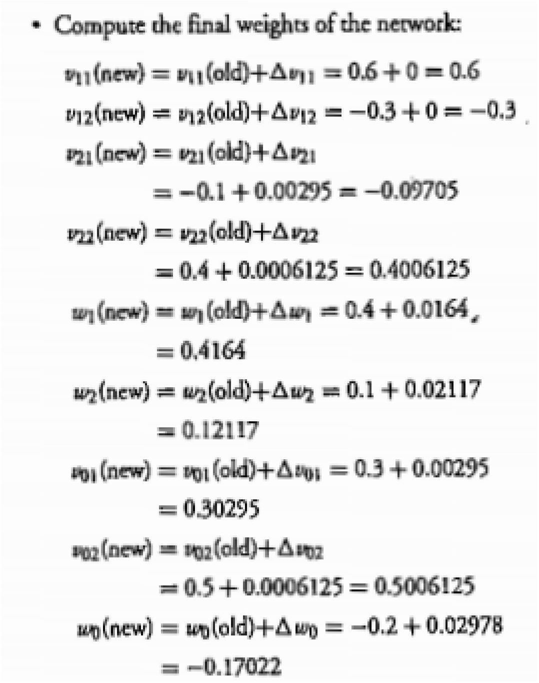
$$\begin{aligned} f'(z_{in1}) &= f(z_{in1})\,[1 - f(z_{in1})] \\ &= 0.5498[1 - 0.5498] = 0.2475 \\ \delta_1 &= \delta_{in1} f'(z_{in1}) \\ &= 0.04764 \times 0.2475 = 0.0118 \end{aligned}$$

Error, $\delta_2 = \delta_{in2} f'(z_{in2})$

$$\begin{aligned} f'(z_{in2}) &= f(z_{in2})\,[1 - f(z_{in2})] \\ &= 0.7109[1 - 0.7109] = 0.2055 \\ \delta_2 &= \delta_{in2} f'(z_{in2}) \\ &= 0.01191 \times 0.2055 = 0.00245 \end{aligned}$$

Now find the changes in weights between input and hidden layer:

$$\begin{aligned} \Delta v_{11} &= \alpha\delta_1 x_1 = 0.25 \times 0.0118 \times 0 = 0 \\ \Delta v_{21} &= \alpha\delta_1 x_2 = 0.25 \times 0.0118 \times 1 = 0.00295 \\ \Delta v_{01} &= \alpha\delta_1 = 0.25 \times 0.0118 = 0.00295 \\ \Delta v_{12} &= \alpha\delta_2 x_1 = 0.25 \times 0.00245 \times 0 = 0 \\ \Delta v_{22} &= \alpha\delta_2 x_2 = 0.25 \times 0.00245 \times 1 = 0.0006125 \\ \Delta v_{02} &= \alpha\delta_2 = 0.25 \times 0.00245 = 0.0006125 \end{aligned}$$



- Compute the final weights of the network:

$$v_{11}(\text{new}) = v_{11}(\text{old}) + \Delta v_{11} = 0.6 + 0 = 0.6$$
$$v_{12}(\text{new}) = v_{12}(\text{old}) + \Delta v_{12} = -0.3 + 0 = -0.3$$
$$v_{21}(\text{new}) = v_{21}(\text{old}) + \Delta v_{21}$$
$$= -0.1 + 0.00295 = -0.09705$$
$$v_{22}(\text{new}) = v_{22}(\text{old}) + \Delta v_{22}$$
$$= 0.4 + 0.0006125 = 0.4006125$$
$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0.4 + 0.0164,$$
$$= 0.4164$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0.1 + 0.02117$$
$$= 0.12117$$
$$v_{01}(\text{new}) = v_{01}(\text{old}) + \Delta v_{01} = 0.3 + 0.00295$$
$$= 0.30295$$
$$v_{02}(\text{new}) = v_{02}(\text{old}) + \Delta v_{02}$$
$$= 0.5 + 0.0006125 = 0.5006125$$
$$w_0(\text{new}) = w_0(\text{old}) + \Delta w_0 = -0.2 + 0.02978$$
$$= -0.17022$$

A **Radial Basis Function Network** (RBF network) is a type of artificial neural network that is widely used for regression, classification, and function approximation. Its architecture and operation method differ from traditional feedforward neural networks like the **Multilayer Perceptron (MLP)**. Let's explore the RBF network and compare it with an MLP to highlight the differences.

---

1. Radial Basis Function Network (RBF) Architecture:

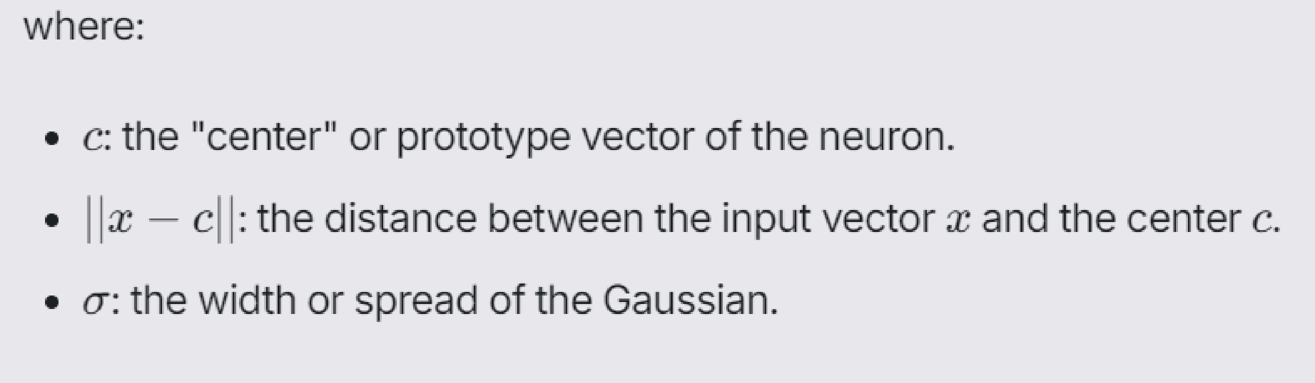
The RBF network typically has **three layers**:

1. **Input layer:**
   - Consists of input neurons, one neuron for each feature.
2. **Hidden layer:**
   - This layer applies radial basis functions (e.g., Gaussian function) to transform the input space into a higher-dimensional space.
   - Each hidden neuron calculates the distance between the input vector and a "center" vector associated with the neuron, then applies the radial basis function (e.g., Gaussian) to this distance.
3. **Output layer:**
   - The output layer typically performs a linear combination of the radial basis function outputs from the hidden layer.
   - This gives the final predicted output.

Activation Function in Hidden Layer:

- Radial Basis Functions are commonly chosen as Gaussian functions, defined as:

$$\phi(||x-c||) = \exp\left(-\frac{||x-c||^2}{2\sigma^2}\right)$$



where:

- c: the "center" or prototype vector of the neuron.
- $||x-c||$: the distance between the input vector x and the center c.
- σ: the width or spread of the Gaussian.

Simple Architecture Diagram:

Input -> Hidden Layer with RBF -> Weighted Sum -> Output

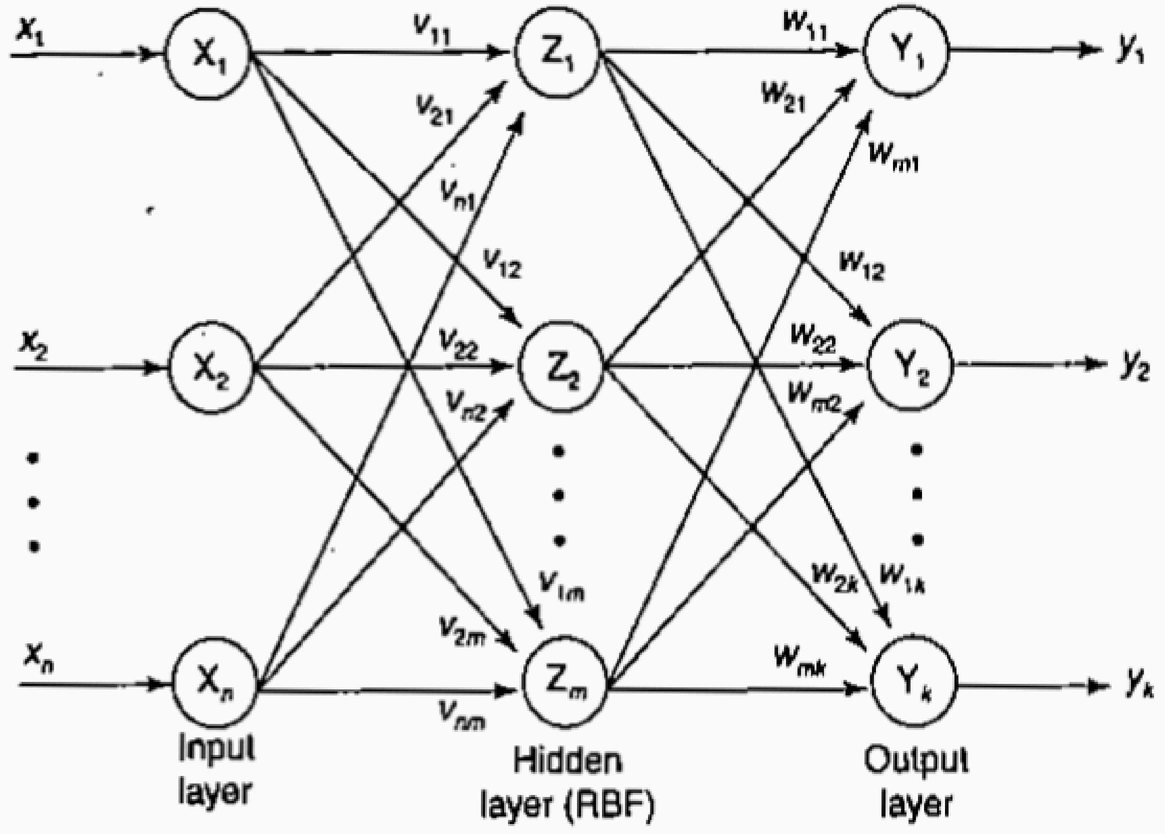


Figure 3-12 Architecture of RBF.

2. Training Algorithm:

The training of an RBF network generally involves the following steps:

1. **Determine the RBF centers**:

   - The hidden layer centers c are typically initialized using clustering techniques like **k-means** or extracted from the data in some other way.

2. **Set the spread parameter

$$\sigma$$

   - The spreads (or widths) of the radial basis functions can either be set manually or learned during training.
   - A commonly used heuristic is to set

$$\sigma$$

   based on the distance between the chosen RBF centers.

3. **Train the output layer weights**:
   - Once the centers and spreads are fixed, the output layer is responsible for mapping the outputs of the RBF layer to actual predicted values.
   - The output layer weights are typically optimized using **linear least squares** or other regression-based methods.

---

3. Multilayer Perceptron (MLP) vs. RBF Network:

Aspect	Multilayer Perceptron (MLP)	Radial Basis Function (RBF) Network
Layers	Typically multiple fully connected layers with non-linear activation functions.	Three layers: input, one RBF hidden layer, and output layer.
Hidden Layer Activation	Uses non-linear activation functions like Sigmoid, ReLU, etc.	Uses radial basis functions like Gaussian.
Feature Transformation	Non-linear transformations are learned through backpropagation across layers.	Explicitly transforms input into a higher-dimensional space using RBF.
Training Algorithm	Backpropagation with gradient descent to update all weights.	RBF centers are determined first (e.g., k-means), output weights are trained separately.
Training Speed	Slower, especially with many layers and parameters.	Faster, as the output layer can often be solved using linear regression.
Output Interpretation	Non-linear decision boundaries are learned through multilayer transformations.	Generally linear output but based on the RBF-transformed space.
Best for…	General-purpose tasks with more complex, non-linear	Function approximation, interpolation, and simpler

Aspect	Multilayer Perceptron (MLP)	Radial Basis Function (RBF) Network
	relationships.	decision problems.

Key Difference in Functioning:

- The **MLP** learns hyperplanes and non-linear transformations end-to-end across layers via backpropagation.
- The **RBF network** explicitly transforms inputs into a higher-dimensional space using non-linear radial basis functions in one step (hidden layer), and focuses on linear mapping from that space to the output.

Advantages/Disadvantages of RBF:

Advantages:

- Faster training due to closed-form solution for output layer training.
- Simpler architecture compared to MLPs.
- Better at solving interpolation or smooth function approximation problems.

Disadvantages:

- RBF networks often require many hidden layer neurons to model complex problems.
- Choosing an appropriate number of centers, spreads

  $$\sigma$$

  and their tuning can be tricky.

Conclusion:

While both RBF networks and MLPs are neural architectures, their design philosophy and training methodologies differ significantly. MLPs excel at deep feature learning, while RBF networks work well for interpolation and function approximation in cases where the data is not excessively complex.