Unit - II - Neural Networks: Introduction

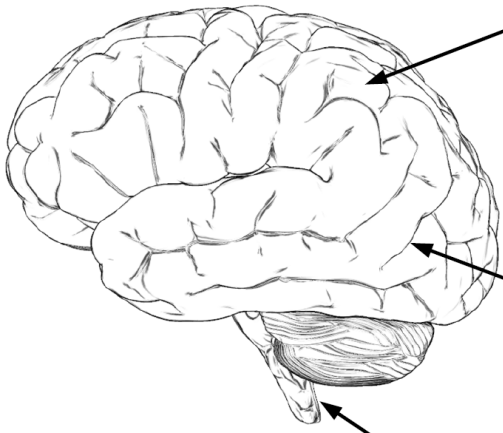
Sumanto Dutta

23.08.2024

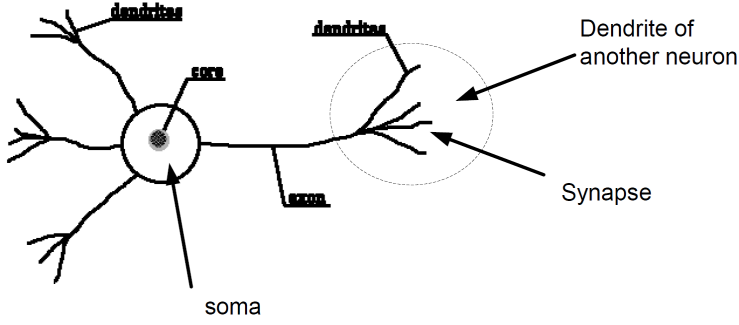
Biological Nervous System

- The biological nervous system is central to many living things, especially humans.
- The human brain is the center of the nervous system.
- A biological nervous system consists of a large number of interconnected processing units called neurons.
- Each neuron is approximately 10μm long and operates in parallel.
- A human brain contains about 10^{11} neurons that communicate using electrical impulses.

Brain: Center of the Nervous System



Neuron: Basic Unit of Nervous System



- Neuron structure includes dendrite, soma, axon, and synapse.
- Dendrite: A bush of very thin fibers.
- Axon: A long cylindrical fiber.
- Soma: Also known as the cell body, similar to the nucleus of a cell.
- Synapse: Junction where the axon connects with the dendrites of neighboring neurons.

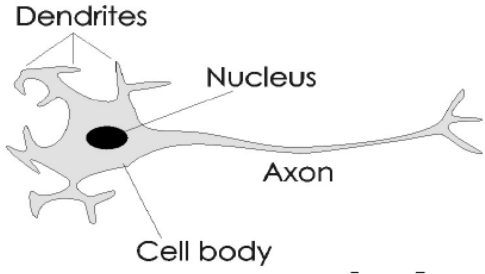
Neuron and Its Working

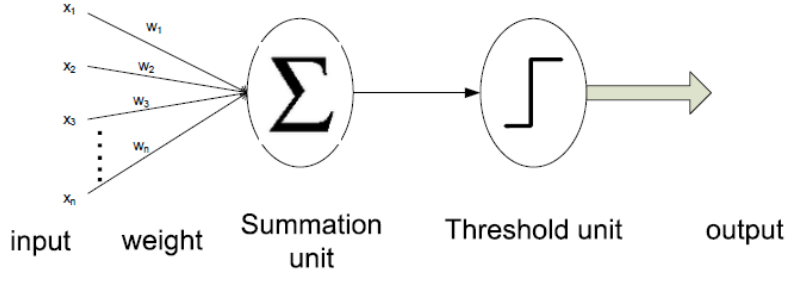
- Neurons contain a chemical called neurotransmitter, responsible for transmitting signals.
- Inputs from other neurons arrive via dendrites and accumulate at the synapse.
- A signal may produce an electrical impulse lasting about a millisecond.
- Signals need to cross a threshold value to produce a pulse in the axon.

Artificial Neural Network (ANN)

- The human brain is viewed as a complex, interconnected network of simple processing elements called neurons.
- Artificial Neural Networks (ANNs) are simplified models of biological nervous systems.
- ANNs are motivated by the computing performed by the human brain.
- The behavior of a biological neural network can be captured by a simple model called an artificial neural network.

Analogy Between BNN and ANN

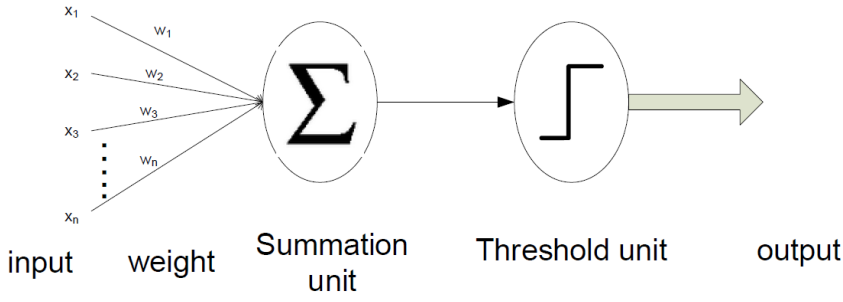




Artificial Neural Network (ANN) (cont.)

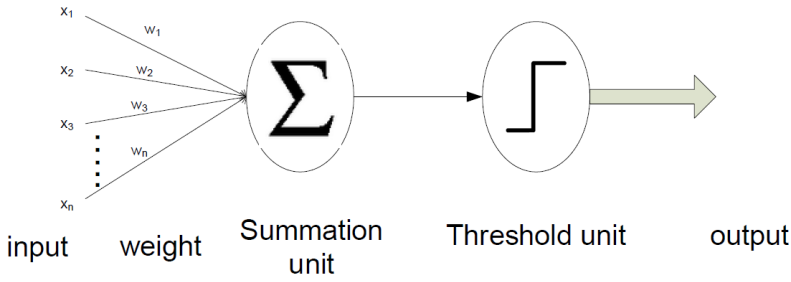
- A neuron is part of an interconnected network in the nervous system.
- Functions include computing input signals, transporting signals, storing information, perception, automatic training, and learning.
- Every component of the artificial neuron model bears direct analogy to a biological neuron.

Artificial Neural Network (ANN) (cont.)



- Inputs $x_1, x_2, \ldots, x_n$ are the inputs to the artificial neuron.
- Weights $w_1, w_2, \ldots, w_n$ are attached to the input links.
- Input signals are passed to the cell body through the synapse.
- Synapses may accelerate or retard an arriving signal, modeled by the weights.

Artificial Neural Network (ANN) (cont.)



- The total input I received by the soma of the artificial neuron is:

$$I = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \sum_{i=1}^{n} w_i x_i$$

- To generate the final output y, the sum is passed to a transfer function φ, which releases the output:

$$y = \varphi(I)$$

Thresholding Function in ANN

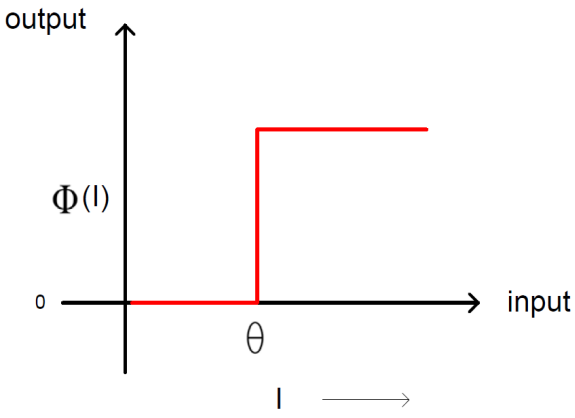
- A common transfer function is the thresholding function.
- Sum I is compared with a threshold value θ.
- If $I > \theta$, the output is 1; otherwise, it is 0.
- This function is also known as a step function or Heaviside function.

$$y = \varphi\left(\sum_{i=1}^{n} w_i x_i - \theta\right)$$

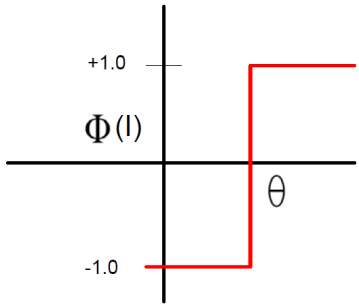
- Where:

$$\varphi(I) = \begin{cases} 1, & \text{if } I > \theta \\ 0, & \text{if } I \leq \theta \end{cases}$$

Thresholding Function Example



(a) Hard-limit transfer function



(b) Signum transfer function

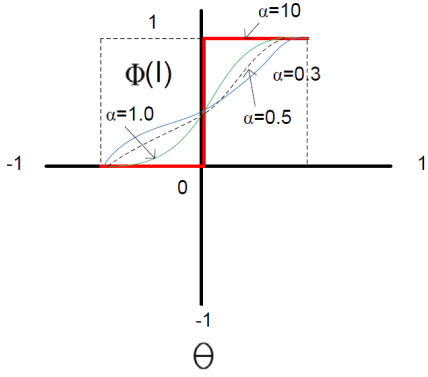
Transformation Functions

- **Hard-limit transfer function**: Compares sum I with a threshold value θ.
- **Linear transfer function**: The output is equal to the input, normalized within a range of $[-1.0, +1.0]$.
- Also known as the Signum or Quantizer function.

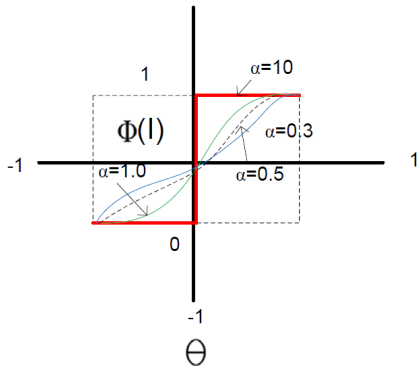
Other Transformation Functions

- **Sigmoid transfer function**: A continuous function that varies gradually between asymptotic values.
- $\varphi(I) = \frac{1}{1+e^{-\alpha I}}$ [Log-Sigmoid]
- $\varphi(I) = \tanh(I) = \frac{e^{\alpha I}-e^{-\alpha I}}{e^{\alpha I}+e^{-\alpha I}}$ [Tan-Sigmoid]
- α is the coefficient of the transfer function.

Transfer Functions in ANN



(a) Log-Sigmoid transfer function



(b) Tan-Sigmoid transfer function

Advantages of ANN

- ANNs exhibit mapping capabilities, allowing them to map input patterns to their associated output patterns.
- ANNs learn by examples, enabling training with known examples before testing on unknown instances.
- ANNs possess the capability to generalize, applying knowledge where exact mathematical models are not possible.
- ANNs are robust and fault-tolerant, capable of recalling full patterns from incomplete or noisy patterns.
- ANNs process information in parallel, at high speed, and in a distributed manner, enabling massively parallel distributed processing systems.

Introduction : Perceptron

- The Perceptron is a fundamental algorithm for binary classification.
- It models a linear decision boundary between two classes.
- We will cover the mathematical formulation of the Perceptron and work through an example.

Perceptron as a Linear Model

The Perceptron models a linear decision boundary of the form:

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{1}$$

- $\mathbf{x}$ is the input vector: $\mathbf{x} = (x_1, x_2, \ldots, x_n)$
- $\mathbf{w}$ is the weight vector: $\mathbf{w} = (w_1, w_2, \ldots, w_n)$
- b is the bias term.

The decision rule is based on the sign of $\mathbf{w} \cdot \mathbf{x} + b$:

$$y = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) \tag{2}$$

where $y = 1$ if $\mathbf{w} \cdot \mathbf{x} + b > 0$, and $y = -1$ otherwise.

Mathematical Formulation

The learning rule adjusts weights and bias to minimize classification errors.
Given a training example $\mathbf{x_i}$ with true label y_i, the prediction is:

$$\hat{y}_i = \text{sign}(\mathbf{w} \cdot \mathbf{x_i} + b) \tag{3}$$

If the prediction is correct ($y_i = \hat{y}_i$), no update is needed. If $y_i \neq \hat{y}_i$, update the weights and bias:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha y_i \mathbf{x_i} \tag{4}$$

$$b \leftarrow b + \alpha y_i \tag{5}$$

where α is the learning rate (a small positive constant).

Geometric Interpretation

The perceptron algorithm seeks to find a hyperplane separating two classes by iteratively adjusting the weight vector.

- When the perceptron makes a mistake, it adjusts $\mathbf{w}$ to shift the hyperplane toward the misclassified point.
- This adjustment brings the hyperplane closer to the true boundary.
- The weight update shifts the decision boundary in the direction of $\mathbf{x_i}$.

The goal is to minimize the misclassifications by moving the boundary iteratively until it perfectly separates the two classes (if linearly separable).

Perceptron Algorithm (Single Layer)

Step 0: Initialize the weights and the bias (for easy calculation, they can be set to zero). Also initialize the learning rate $\alpha(0, \alpha, 1)$ for simplicity, α is set to 1.

Step 1: Perform Step 2 to 6 until the final stopping condition is false.

Step 2: Perform Step 3 to 5 for each training pair indicated by $s : t$.

Step 3: The input layer containing input units is applied with identity activation function: $x_i = s_i$.

Step 4: Calculate the output of the network. To do so, first obtain the net input:

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

where n is the number of input neurons in the input layer. Then, apply activation (signum) over the net input calculated to obtain the output:

$$signum(y_{\text{in}}) = \begin{cases} 1 & \text{if } y_{\text{in}} > 0 \\ -1 & \text{if } y_{\text{in}} \leq 0 \end{cases}$$

Step 5: Weight and bias adjustment: compare the value of the actual (calculated) output (y) and the desired (target) output (t).

if $y \neq t$, then

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha t$$

else we have

$$w_i(\text{new}) = w_i(\text{old})$$
$$b(\text{new}) = b(\text{old})$$

Step 6: Train the network until there is no weight change. This is the stopping condition for the network. If this condition is not met, then start again from Step 2.

Example: Step-by-Step Solution

Consider the following dataset with two features:

x_1	x_2	t
1	1	1
1	−1	−1
−1	1	−1
−1	−1	−1

with initial weights: $\mathbf{w} = (0, 0)$, bias $b = 0$, and learning rate $\alpha = 1$.
Step 1: Process the first point $\mathbf{P_1} = (1, 1)$ with true label $t = 1$:

$$\hat{y} = \text{signum}(0 \cdot 1 + 0 \cdot 1 + 0) = 0 \quad \text{(wrong prediction)}$$

Update weights:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$

$$w_1(\text{new}) = 0 + 1 \cdot 1 \cdot 1 = 1$$

$$w_2(\text{new}) = 0 + 1 \cdot 1 \cdot 1 = 1$$

Update bias:

$$b(\text{new}) = b(\text{old}) + \alpha t = 0 + 1 \cdot 1 = 1$$

Step 2: Process the second point $\mathbf{P_2} = (1, -1)$ with true label $t = -1$:

$$\hat{y} = \text{signum}(1 \cdot 1 + 1 \cdot -1 + 1) = 1 \quad \text{(wrong prediction)}$$

Update weights:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$

$$w_1(\text{new}) = 1 + 1 \cdot -1 \cdot 1 = 0$$

$$w_2(\text{new}) = 1 + 1 \cdot -1 \cdot -1 = 2$$

Update bias:

$$b(\text{new}) = b(\text{old}) + \alpha t = 1 + 1 \cdot -1 = 0$$

Step 3: Process the third point $\mathbf{P_3} = (-1, 1)$ with true label $t = -1$:

$$\hat{y} = \text{signum}(0 \cdot -1 + 2 \cdot 1 + 0) = 1 \quad \text{(wrong prediction)}$$

Update weights:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$

$$w_1(\text{new}) = 0 + 1 \cdot -1 \cdot -1 = 1$$

$$w_2(\text{new}) = 2 + 1 \cdot -1 \cdot 1 = 1$$

Update bias:

$$b(\text{new}) = b(\text{old}) + \alpha t = 0 + 1 \cdot -1 = -1$$

Step 4: Process the fourth point $\mathbf{P_3} = (-1, -1)$ with true label $t = -1$:

$$\hat{y} = \text{signum}(1 \cdot -1 + 1 \cdot -1 + -1) = -1 \quad \text{(Correct prediction)}$$

Update weights:

$$w_i(\text{new}) = w_i(\text{old})$$

$$w_1(\text{new}) = 1$$

$$w_2(\text{new}) = 1$$

Update bias:

$$b(\text{new}) = b(\text{old}) = -1$$

Conclusion

- The Perceptron learning algorithm updates weights and biases to classify points correctly.
- The algorithm converges for linearly separable datasets.
- The final weight vector defines the separating hyperplane.

Introduction to ADALINE

- ADALINE stands for Adaptive Linear Neuron or Adaptive Linear Element.
- It is a single-layer neural network similar to the Perceptron but uses a continuous activation function.
- Unlike the Perceptron, ADALINE minimizes the error between the predicted and actual output using the least mean squares (LMS) rule.

ADALINE Model

The ADALINE algorithm computes the output as a linear combination of the input features:

$$\hat{y} = \mathbf{w} \cdot \mathbf{x} + b \tag{6}$$

where:

- $\mathbf{x}$ is the input vector: $\mathbf{x} = (x_1, x_2, \ldots, x_n)$
- $\mathbf{w}$ is the weight vector: $\mathbf{w} = (w_1, w_2, \ldots, w_n)$
- b is the bias term.

The difference from the Perceptron is that ADALINE calculates $\hat{y}$ as a continuous value rather than a binary decision.

ADALINE Learning Rule

The ADALINE learning rule minimizes the error between the predicted output $\hat{y}$ and the true label y using the least mean squares (LMS) approach.
The squared error for a single training example is:

$$E = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(y - (\mathbf{w} \cdot \mathbf{x} + b))^2 \tag{7}$$

The weight update rule is derived by taking the derivative of the error function with respect to the weights:

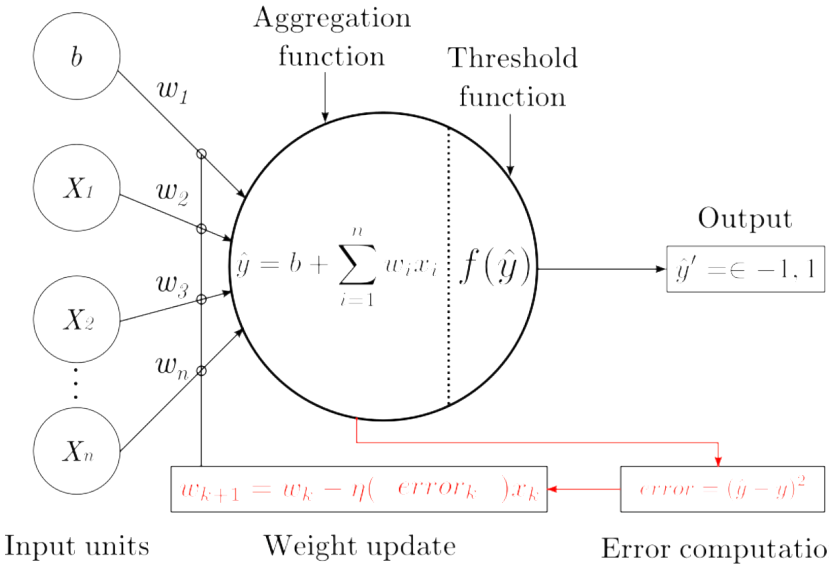
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(y - \hat{y})\mathbf{x} \tag{8}$$

and for the bias term:

$$b \leftarrow b + \alpha(y - \hat{y}) \tag{9}$$

where α is the learning rate.

ADALINE training loop



Geometric Interpretation

ADALINE uses gradient descent to minimize the error. The weight update moves in the direction that reduces the error the most.

- The squared error function is a paraboloid, and gradient descent finds the minimum by iteratively adjusting the weights.
- The algorithm tries to find the optimal hyperplane by minimizing the total squared error.

ADALINE Learning Algorithm

Step 0: Initialize the weights and the bias to some random values (but not to zero). Also, initialize the learning rate α.

Step 1: Perform Steps 2-6 when the stopping condition is false.

Step 2: Perform Steps 3-5 for each bipolar training pair $s : t$.

Step 3: Activate each input unit as follows:

$$x_i = s_i \quad (i = 1 \text{ to } n)$$

Step 4: Obtain the net input with the following relation:

$$y_{\text{in}} = \sum_{i=1}^{n} x_i \cdot w_i + b$$

where b is the bias and n is the total number of input neurons.

Step 5: Until the least mean square is obtained $(t - y_{\text{in}})$, adjust the weight and bias as follows:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha \cdot (t - y_{\text{in}}) \cdot x_i$$

$$b(\text{new}) = b(\text{old}) + \alpha \cdot (t - y_{\text{in}})$$

Now calculate the error using:

$$E_i = (t - y_{\text{in}})^2$$

Step 6: Test for the stopping condition. If the error generated $(E_T = \sum_{i=1}^{n} E_i)$ is less than or equal to the specified tolerance (E_S), then stop.

Example: Step-by-Step Solution

Consider a dataset with the following input vectors and true labels:

x_1	x_2	t
1	1	1
1	−1	−1
−1	1	−1
−1	−1	−1

with initial weights: $\mathbf{w} = (0, 0)$, bias $b = 0$, and learning rate $\alpha = 0.1$ and $E_S =$.
Step 1: Process the first point $\mathbf{P_1} = (1, 1)$ with true label $t = 1$:

$$y_{in} = 0 \cdot 1 + 0 \cdot 1 + 0 = 0$$

$$E_1 = t - y_{in} = 1 - 0 = 1$$

Update weights:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha E x_i$$

$$w_1(\text{new}) = 0 + 0.1 \cdot 1 \cdot 1 = 0.1$$

$$w_2(\text{new}) = 0 + 0.1 \cdot 1 \cdot 1 = 0.1$$

Update bias:

$$b(\text{new}) = b(\text{old}) + \alpha E = 0 + 0.1 \cdot 1 = 0.1$$

Step 2: Process the second point $\mathbf{P_2} = (1, -1)$ with true label $t = -1$:

$$y_{in} = 0.1 \cdot 1 + 0.1 \cdot -1 + 0.1 = 0.1$$

$$E_1 = t - y_{in} = -1 - 0.1 = -1.1$$

Update weights:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha E x_i$$

$$w_1(\text{new}) = 0.1 + 0.1 \cdot -1.1 \cdot 1 = -0.01$$

$$w_2(\text{new}) = 0.1 + 0.1 \cdot -1.1 \cdot -1 = 0.21$$

Update bias:

$$b(\text{new}) = b(\text{old}) + \alpha E = 0 + 0.1 \cdot -1.1 = -0.11$$

Step 3: Process the third point $P_3 = (-1, 1)$ with true label $t = -1$

$$y_{\text{in}} = -0.01 \cdot (-1) + 0.21 \cdot 1 - 0.11 = 0.11$$

$$E_3 = t - y_{\text{in}} = -1 - 0.11 = -1.11$$

Update weights:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha E x_i$$

$$w_1(\text{new}) = -0.01 + 0.1 \cdot (-1.11) \cdot (-1) = 0.1$$

$$w_2(\text{new}) = 0.21 + 0.1 \cdot (-1.11) \cdot 1 = 0.10$$

Update bias:

$$b(\text{new}) = b(\text{old}) + \alpha E = -0.11 + 0.1 \cdot (-1.11) = -0.221$$

Step 4: Process the fourth point $P_4 = (-1, -1)$ with true label $t = -1$

$$y_{\text{in}} = 0.1 \cdot (-1) + 0.10 \cdot (-1) - 0.221 = -0.421$$

$$E_4 = t - y_{\text{in}} = -1 - (-0.421) = -0.579$$

Update weights:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha E x_i$$

$$w_1(\text{new}) = 0.1 + 0.1 \cdot (-0.579) \cdot (-1) = 0.158$$

$$w_2(\text{new}) = 0.10 + 0.1 \cdot (-0.579) \cdot (-1) = 0.158$$

Update bias:

$$b(\text{new}) = b(\text{old}) + \alpha E = -0.221 + 0.1 \cdot (-0.579) = -0.2799$$

Conclusion

- ADALINE differs from the Perceptron by using a continuous output and minimizing a cost function (LMS).
- The algorithm converges to a solution where the error is minimized if the learning rate is small enough.
- It works best with linearly separable data but can be extended using non-linear activation functions and multi-layer networks.

Back Propagation Neural Networks Training Algorithm

Back Propagation Neural Networks will use a binary sigmoid activation function. The training will have the following three phases:

- **Phase 1** – Feed Forward Phase
- **Phase 2** – Back Propagation of Error
- **Phase 3** – Updating of Weights

Steps of the Algorithm

Step 0: Initialize the weights and the bias. For easy calculation and simplicity, take some small random values (but not zero). Also initialize the learning rate $\alpha(0, \alpha, 1)$.

Step 1: Continue Steps 2-10 when the stopping condition is not true.

Step 2: Continue Steps 3-9 for every training pair.

Phase 1 - Feed Forward Phase

Step 3: Each input unit receives input signal x_i and sends it to the hidden unit for all $i = 1$ to n.

Step 4: Calculate the net input at the hidden unit using the following relation:

$$Q_{\text{inj}} = \sum_{i=1}^{n} x_i v_{ij} + b_j$$

Now, calculate the net output by applying the following sigmoidal activation function:

$$Q_j = f(Q_{\text{inj}}) = \frac{1}{1 + e^{-(Q_{\text{inj}})}}$$

Step 5: Calculate the net input at the output layer unit using the following relation:

$$y_{\text{ink}} = \sum_{j=1}^{p} Q_j w_{jk} + b_k$$

Calculate the net output by applying the following sigmoidal activation function:

$$y_k = f(y_{\text{ink}}) = \frac{1}{1 + e^{-(y_{\text{ink}})}}$$

Phase 2

Step 6: Compute the error correcting term, in correspondence with the target pattern received at each **output unit**, as follows:

$$\delta_k = f(y_{\text{ink}})(1 - f(y_{\text{ink}}))(t_k - y_k)$$

Then, send δ_k back to the hidden layer.

Step 7: Now, each **hidden unit** will be the sum of its delta inputs from the output units. The error term can be calculated as follows:

$$\delta_j = f(Q_{\text{inj}})(1 - f(Q_{\text{inj}})) \sum_{k=1}^{m} \delta_k w_{jk}$$

Phase 3 - Updating of Weights and Biases

Step 8: Each **output unit** y_k (for $k = 1$ to m) updates the weight and bias as follows:

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}$$
$$b_k(\text{new}) = b_k(\text{old}) + \Delta b_k$$

Where:

$$\Delta w_{jk} = \alpha \delta_k Q_j$$
$$\Delta b_{0k} = \alpha \delta_k$$

Step 9: Each **hidden unit** q_j (for $j = 1$ to p) updates the weight and bias as follows:

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \Delta v_{ij}$$
$$b_j(\text{new}) = b_j(\text{old}) + \Delta b_j$$

Where:

$$\Delta v_{ij} = \alpha \delta_j x_i$$
$$\Delta b_j = \alpha \delta_j$$

Step 10: Check for the stopping condition, which may be either the number of epochs reached or the target output matching the actual output.

Problem Statement

Q. Consider a *multilayer feed-forward neural network* given below. Let the **learning rate** be 0.5. Assume initial values of weights and biases as given in the table below. Train the network for the training tuples $(1, 1, 0)$ and $(0, 1, 1)$, where the last number is the target output. Show weight and bias updates by using the **back-propagation algorithm**. Assume that the *sigmoid activation function* is used in the network.

W13	W14	W23	W24	W35	W45	b3	b4	b5
0.5	0.2	-0.3	0.5	0.1	0.3	0.6	-0.4	0.8

Neural Network Diagram

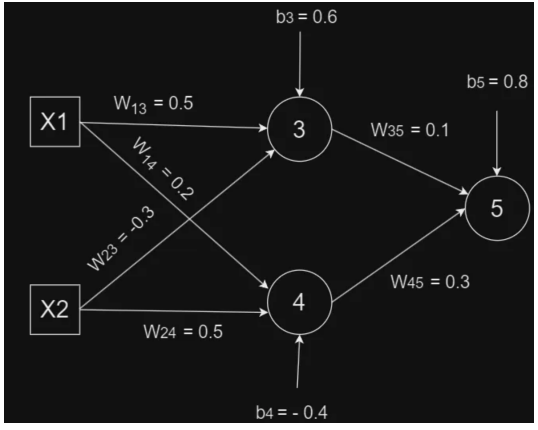


Figure: Neural Network

Training Tuples

Training tuples are as follows:

- $(1, 1, 0)$ — First two inputs are $X_1 = 1$, $X_2 = 1$, and target output $t = 0$
- $(0, 1, 1)$ — First two inputs are $X_1 = 0$, $X_2 = 1$, and target output $t = 1$

Sigmoid Activation Function

The sigmoid activation function is given by:

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

Back-Propagation Algorithm

Steps for back-propagation:

1. Perform a feed-forward step to calculate the output.
2. Compute the error at the output layer.
3. Propagate the error back through the network to adjust the weights and biases.
4. Repeat for each training tuple.

Feedforward Step for Training Tuple $(1, 1, 0)$

Input: $X_1 = 1, X_2 = 1$

1. Calculate the net input for the hidden neurons (3 and 4):

$$Q_3 = X_1 W_{13} + X_2 W_{23} + b_3 = 1 \cdot 0.5 + 1 \cdot (-0.3) + 0.6 = 0.8$$

$$Q_4 = X_1 W_{14} + X_2 W_{24} + b_4 = 1 \cdot 0.2 + 1 \cdot 0.5 + (-0.4) = 0.3$$

2. Apply the sigmoid activation function:

$$O_3 = \frac{1}{1 + e^{-0.8}} = 0.689$$

$$O_4 = \frac{1}{1 + e^{-0.3}} = 0.574$$

Feedforward Step for Output Layer $(1, 1, 0)$

1. Calculate the net input for the output neuron (5):

$$Q_5 = O_3 W_{35} + O_4 W_{45} + b_5 = 0.689 \cdot 0.1 + 0.574 \cdot 0.3 + 0.8 = 1.003$$

2. Apply the sigmoid activation function:

$$O_5 = \frac{1}{1 + e^{-1.003}} = 0.731$$

Target: $t = 0$
Output: $O_5 = 0.731$

Error Calculation for Output Layer

1. Compute the error at the output layer:

$$\delta_5 = (t - O_5) \cdot O_5 \cdot (1 - O_5) = (0 - 0.731) \cdot 0.731 \cdot (1 - 0.731) = -0.144$$

2. Update the weights for the output layer:

$$\Delta W_{35} = \alpha \cdot \delta_5 \cdot O_3 = 0.5 \cdot (-0.144) \cdot 0.689 = -0.0497$$

$$\Delta W_{45} = \alpha \cdot \delta_5 \cdot O_4 = 0.5 \cdot (-0.144) \cdot 0.574 = -0.0413$$

Updated weights:

$$W_{35}(\text{new}) = 0.1 + (-0.0497) = 0.0503$$

$$W_{45}(\text{new}) = 0.3 + (-0.0413) = 0.2587$$

3. Update the bias for the output layer:

$$\Delta b_5 = \alpha \cdot \delta_5 = 0.5 \cdot (-0.144) = -0.072$$

$$b_5(\text{new}) = 0.8 + (-0.072) = 0.728$$

Backpropagation to Hidden Layer (First Training Tuple)

1. Compute the error for the hidden layer neurons:

$$\delta_3 = \delta_5 \cdot W_{35}(\text{new}) \cdot O_3 \cdot (1 - O_3) = (-0.144) \cdot 0.0503 \cdot 0.689 \cdot (1 - 0.689) = -0.0152$$

$$\delta_4 = \delta_5 \cdot W_{45}(\text{new}) \cdot O_4 \cdot (1 - O_4) = (-0.144) \cdot 0.2587 \cdot 0.574 \cdot (1 - 0.574) = -0.0249$$

2. Update the weights for the hidden layer:

$$\Delta W_{13} = \alpha \cdot \delta_3 \cdot X_1 = 0.5 \cdot (-0.0152) \cdot 1 = -0.0076$$

$$\Delta W_{14} = \alpha \cdot \delta_4 \cdot X_1 = 0.5 \cdot (-0.0249) \cdot 1 = -0.0125$$

$$\Delta W_{23} = \alpha \cdot \delta_3 \cdot X_2 = 0.5 \cdot (-0.0152) \cdot 1 = -0.0076$$

$$\Delta W_{24} = \alpha \cdot \delta_4 \cdot X_2 = 0.5 \cdot (-0.0249) \cdot 1 = -0.0125$$

Updated weights:

$$W_{13}(\text{new}) = 0.5 + (-0.0076) = 0.4924$$

$$W_{14}(\text{new}) = 0.2 + (-0.0125) = 0.1875$$

$$W_{23}(\text{new}) = -0.3 + (-0.0076) = -0.3076$$

$$W_{24}(\text{new}) = 0.5 + (-0.0125) = 0.4875$$

Bias Updates for Hidden Layer (First Training Tuple)

1. Update the biases for the hidden layer:

$$\Delta b_3 = \alpha \cdot \delta_3 = 0.5 \cdot (-0.0152) = -0.0076$$

$$\Delta b_4 = \alpha \cdot \delta_4 = 0.5 \cdot (-0.0249) = -0.0125$$

Updated biases:

$$b_3(\text{new}) = 0.6 + (-0.0076) = 0.5924$$

$$b_4(\text{new}) = -0.4 + (-0.0125) = -0.4125$$

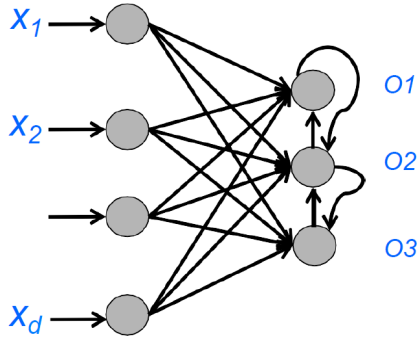
Training Tuple $(0, 1, 1)$

Input: $X_1 = 0, X_2 = 1$

Repeat the feedforward and backpropagation steps for the second training tuple $(0, 1, 1)$. Similar steps as shown previously can be followed, adjusting for the new inputs and target.

Competitive Learning

- A form of unsupervised training where output units compete for input patterns.
- During training, the output unit with the highest activation is declared the winner and its weights are moved closer to the input pattern.
- The strategy is called "winner-take-all," as only the winning neuron is updated.
- Output units may have lateral inhibitory connections, where a winning neuron can inhibit others proportional to its activation level.



Competitive Learning: Activation Function

For normalized vectors, the activation function for the i-th unit can be computed as:

$$g_i(x^{(n)}) = \mathbf{w_i}^T \mathbf{x^{(n)}}$$

Where:

- $\mathbf{w_i}$ is the weight vector of the i-th neuron.
- $\mathbf{x^{(n)}}$ is the input pattern.
- The neuron with the largest activation adapts to be more like the input that caused the excitation.

Afterward, the weight vector is renormalized:

$$\mathbf{w_i}(t+1) = \mathbf{w_i}(t) + \eta \mathbf{x^{(n)}}$$

Competitive Learning: Euclidean Distance

If weights and input patterns are un-normalized, the activation function becomes the Euclidean distance:

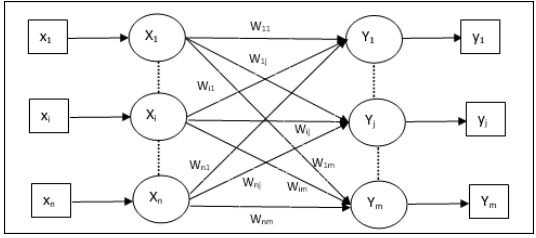
$$g_i(x^{(n)}) = -\sum_{i=1}^{d}(w_i^{(n)} - x_i^{(n)})^2$$

The learning rule becomes:

$$\mathbf{w_i}(t+1) = \mathbf{w_i}(t) + \eta(\mathbf{x^{(n)}} - \mathbf{w_i}(t))$$

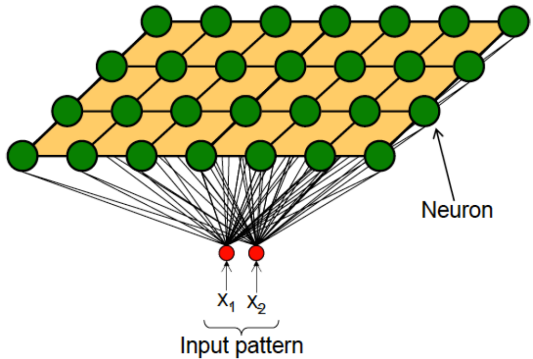
Kohonen Self-Organizing Maps (SOM)

- SOMs map a multidimensional input space onto a lattice of neurons or clusters.
- A key feature is that the mapping is topology-preserving, meaning that neighboring neurons respond to similar input patterns.
- SOMs are typically organized as one- or two-dimensional lattices (a string or mesh) for visualization and dimensionality reduction.
- SOMs have a strong neurobiological basis, mimicking how the mammalian brain processes sensory information.

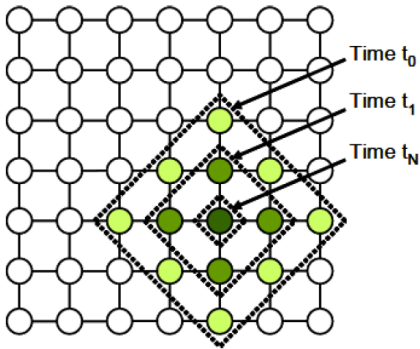


SOM: Competition, Cooperation, Adaptation

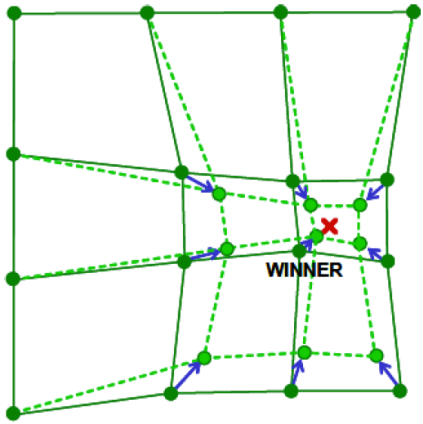
- **Competition**: Each neuron is assigned a weight vector. The neuron whose weight vector is closest to the input pattern is declared the winner.



- **Cooperation**: The activation of the winning neuron spreads to neurons in its immediate neighborhood.



- **Adaptation**: During training, the winner neuron and its neighbors adapt to become more similar to the input pattern.



SOM: Neighborhood Function

The size of the neighborhood changes over time:

- Initially large, promoting a topology-preserving mapping.
- Shrinks over time, allowing neurons to specialize in later stages of training.
- The neighborhood function is commonly a Gaussian function.

Learning Rate Decay

- The learning rate η is crucial for the adaptation process in SOMs.
- It typically starts large to allow for significant changes early in training, which helps with global organization.
- Over time, the learning rate decays according to a predefined schedule, ensuring the model gradually shifts from exploration to exploitation.

The common decay function for the learning rate is:

$$\eta(t) = \eta_0 \cdot e^{-t/\tau}$$

where:

- η_0 is the initial learning rate.
- t is the current iteration.
- τ is the time constant for the decay.

Kernel Function in SOM

- The kernel function $h_{ij}(t)$, also known as the neighborhood function, defines how the adaptation of weights spreads from the winning neuron to its neighbors.
- It often takes a Gaussian form:

$$h_{ij}(t) = e^{-\frac{d^2(i,j)}{2\sigma(t)^2}}$$

where:

- $d(i,j)$ is the lattice distance between neuron i (winner) and neuron j (neighbor).
- $\sigma(t)$ represents the radius of the neighborhood, which also decays over time.

Decay Rule for Neighborhood Radius

The radius of the neighborhood, $\sigma(t)$, is another critical parameter that decays over time:

$$\sigma(t) = \sigma_0 \cdot e^{-t/\tau_\sigma}$$

where:

- σ_0 is the initial radius.
- τ_σ is the decay constant for the neighborhood radius.

This decay ensures that the SOM gradually moves from a global ordering phase to a fine-tuning phase, allowing neurons to specialize more precisely to parts of the input space.

SOM Learning Algorithm

Step 1: Initialize the weight vectors randomly.
Step 2: For each input vector $\mathbf{x^{(n)}}$:

1. Find the winning neuron i such that:

$$\|\mathbf{x^{(n)}} - \mathbf{w_i}\| = \min_j \|\mathbf{x^{(n)}} - \mathbf{w_j}\|$$

2. Update the weights of the winning neuron and its neighbors:

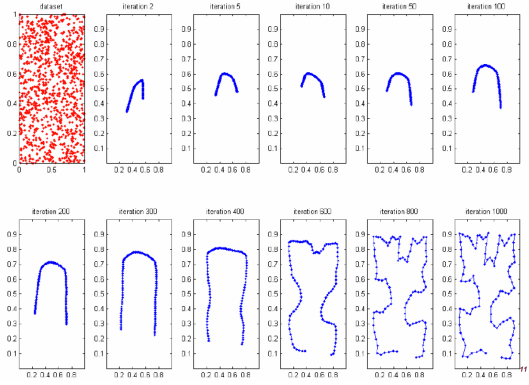
$$\mathbf{w_i}(t+1) = \mathbf{w_i}(t) + \eta(t) \cdot h_{ij}(t) \cdot (\mathbf{x^{(n)}} - \mathbf{w_i}(t))$$

where $h_{ij}(t)$ is the neighborhood function.

SOM Example (1D)

In a one-dimensional SOM, neurons are organized in a linear array. During training, each neuron competes to represent a section of the input space, and the winning neuron adjusts its weights to match the input better.

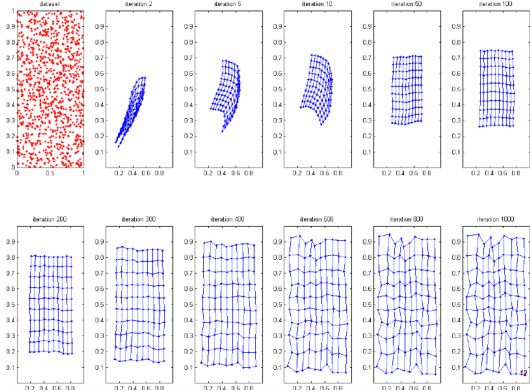
The competition leads to a smooth ordering of the neurons, preserving the topology of the input space.



SOM Example (2D)

In a two-dimensional SOM, neurons are arranged in a grid. Each neuron represents a small part of the input space, and during training, neurons close to each other in the grid become sensitive to similar inputs.

This allows for effective dimensionality reduction and visualization of high-dimensional data.



Thank you !!!