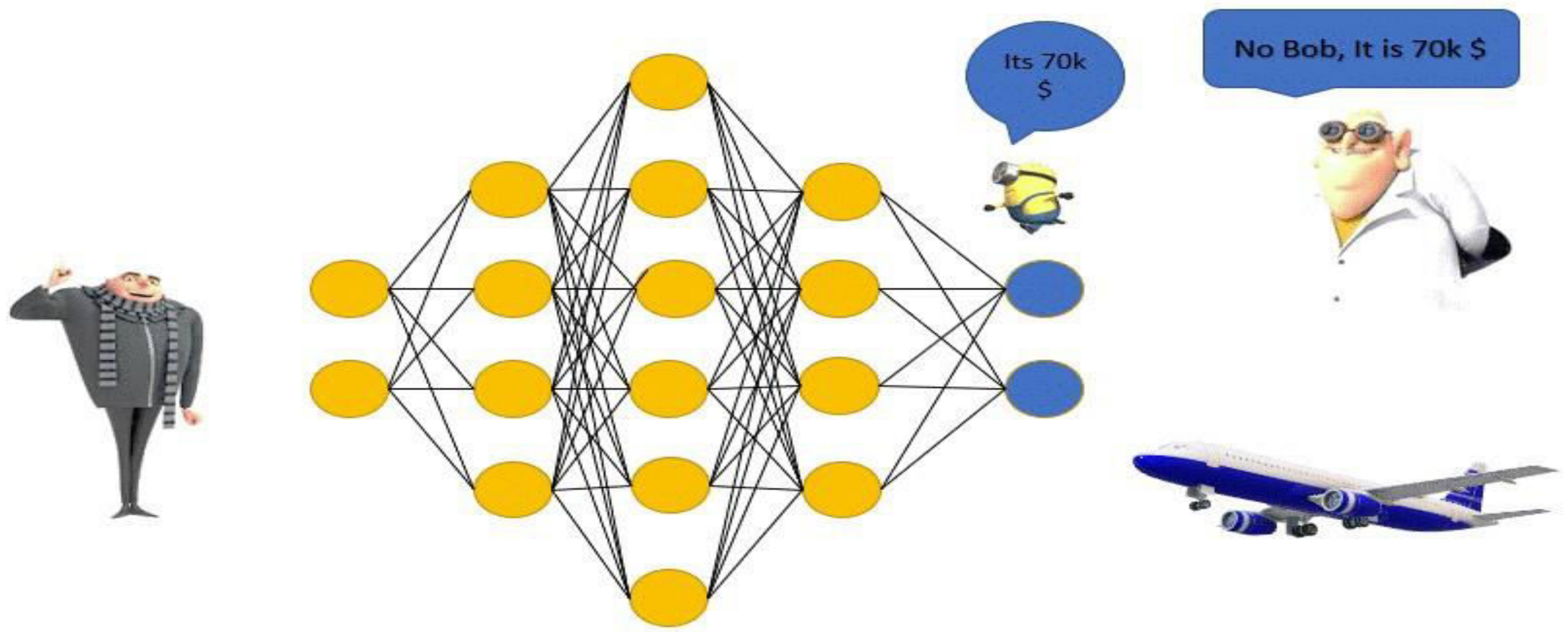
Module No. 2 - Practical Deep Networks

Multilayer Perceptron



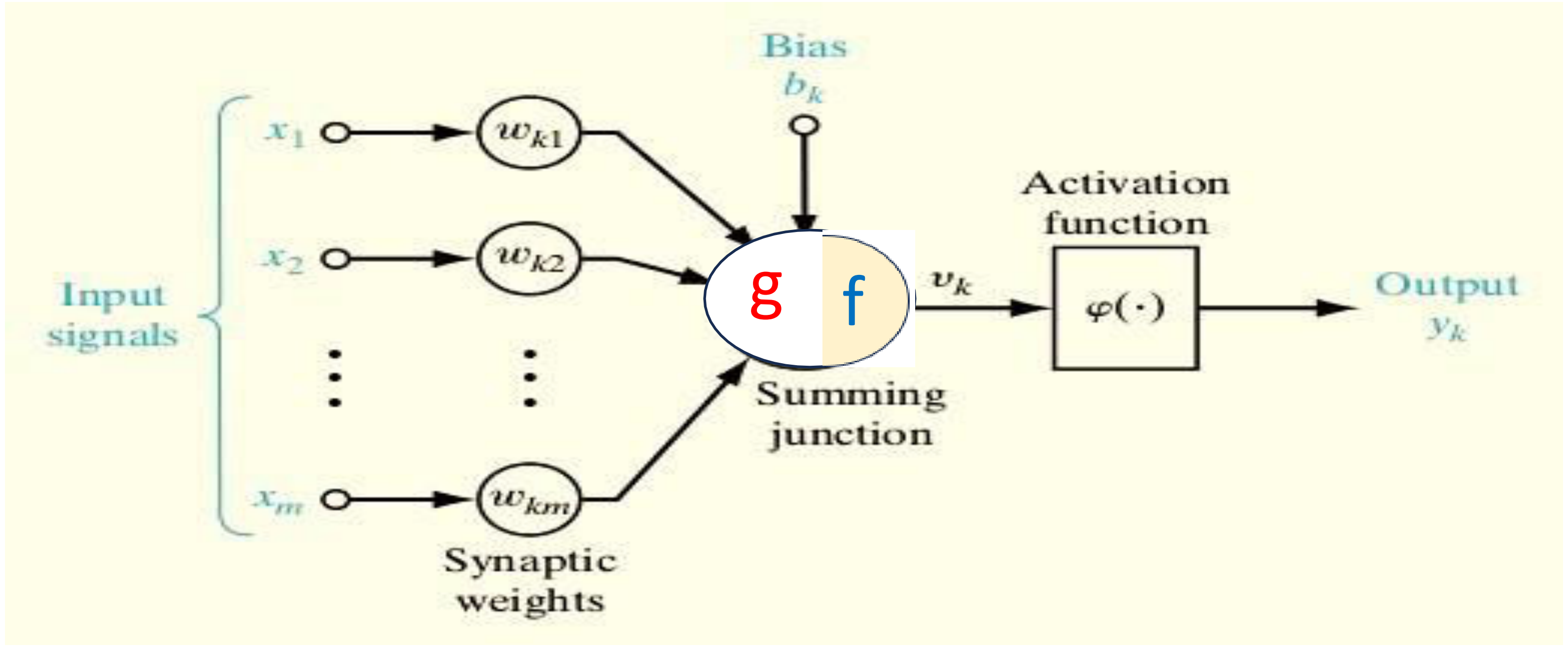
Topics

- McCullough-Pitts Model
- Learning Rules
- Why MLP?
- MLP
- The Multi-layered Perceptron

McCullough-Pitts Model

- It may be divided into 2 parts. The first part, g takes input, performs an aggregation, and based on the aggregated value the second part, f makes a decision

Warren McCulloch and Walter Pitts
(1943)



- Let us suppose that I want to predict my own decision, whether to watch a random football game or not on TV. The inputs are all boolean i.e., {0,1} and my output variable is also boolean {0: Will watch it, 1: Won't watch it}. The McCulloch-Pitts neural model is also known as the linear threshold gate

McCullough-Pitts Model

This model uses the following components:

- **Weights** (w): Represent the strength of the connections.
- **Threshold** (t): Determines when the neuron "fires" (outputs 1).
- **Output Rule**:

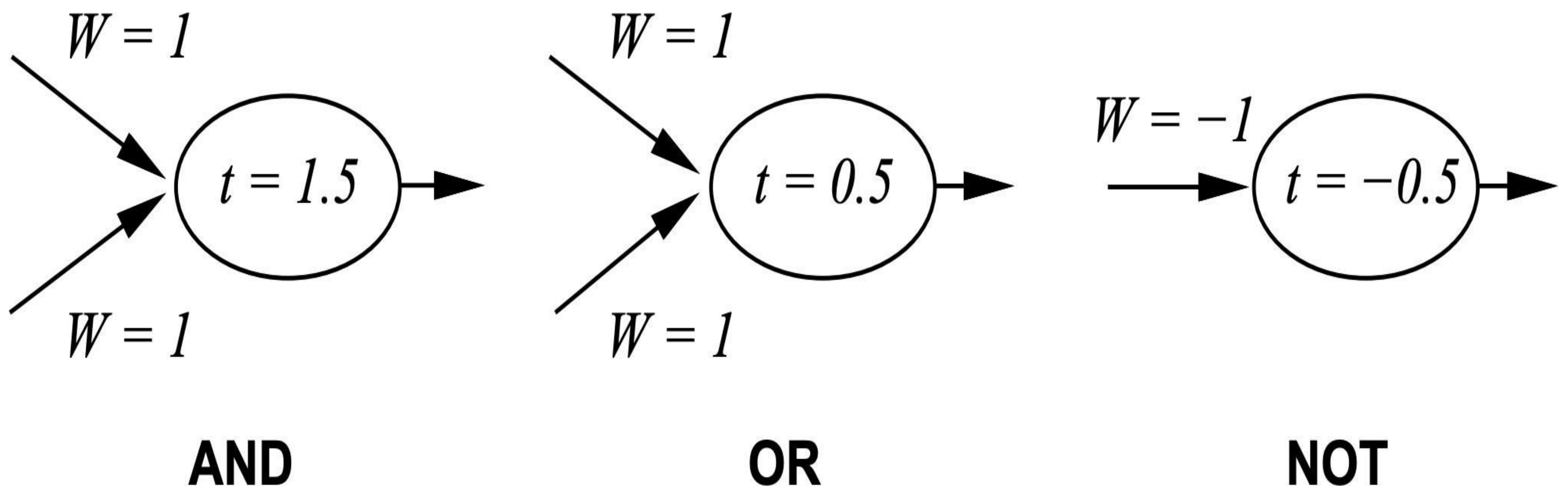
$$y = \text{sign}\left(\sum w_i x_i - t\right)$$

where:

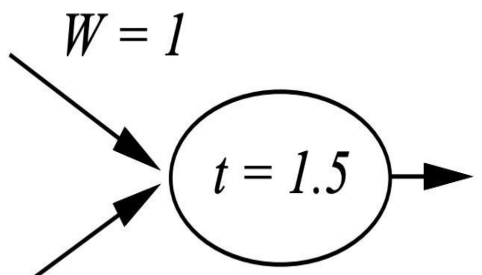
- $\sum w_i x_i$: Weighted sum of inputs.
- t: Threshold value.
- $y = 1$ if $\sum w_i x_i \geq t$, otherwise $y = 0$.

McCullough-Pitts Model

- McCulloch and Pitts (1943) showed how linear threshold units can be used to compute logical functions.



$$y = sign(\sum w_i I_i - T)$$

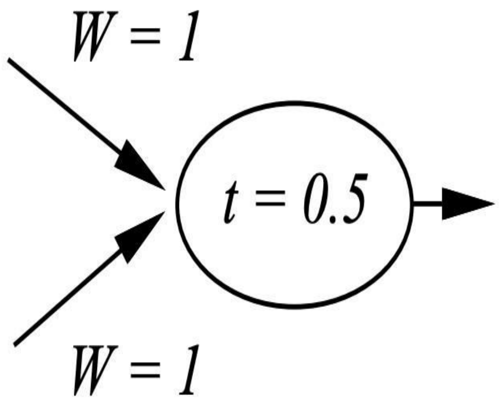


AND

The AND gate outputs `1` only when both inputs are `1`.

Input x_1	Input x_2	Weighted Sum $w_1x_1 + w_2x_2$	Threshold $t = 1.5$	Output y
0	0	$0 \cdot 1 + 0 \cdot 1 = 0$	$0 < 1.5$	0
0	1	$0 \cdot 1 + 1 \cdot 1 = 1$	$1 < 1.5$	0
1	0	$1 \cdot 1 + 0 \cdot 1 = 1$	$1 < 1.5$	0
1	1	$1 \cdot 1 + 1 \cdot 1 = 2$	$2 \geq 1.5$	1

- **Weights**: $w = 1$ for both inputs.
- **Threshold**: $t = 1.5$.
- **Behavior**:
  - For $x_1 = 1$ and $x_2 = 1$: $w_1x_1 + w_2x_2 = 2 \geq t$, so output = 1.
  - For other combinations (e.g., $x_1 = 0, x_2 = 1$), the weighted sum is less than 1.5, so output = 0.
- This mimics the behavior of an AND gate.

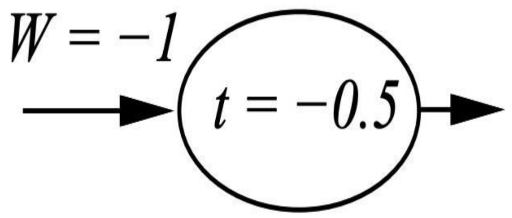


OR

The OR gate outputs `1` if at least one of the inputs is `1`.

Input x_1	Input x_2	Weighted Sum $w_1x_1 + w_2x_2$	Threshold $t = 0.5$	Output y
0	0	$0 \cdot 1 + 0 \cdot 1 = 0$	$0 < 0.5$	0
0	1	$0 \cdot 1 + 1 \cdot 1 = 1$	$1 \geq 0.5$	1
1	0	$1 \cdot 1 + 0 \cdot 1 = 1$	$1 \geq 0.5$	1
1	1	$1 \cdot 1 + 1 \cdot 1 = 2$	$2 \geq 0.5$	1

- **Weights**: $w = 1$ for both inputs.
- **Threshold**: $t = 0.5$.
- **Behavior**:
  - For $x_1 = 1$ or $x_2 = 1$: $w_1x_1 + w_2x_2 \geq 0.5$, so output = 1.
  - For $x_1 = 0$ and $x_2 = 0$: $w_1x_1 + w_2x_2 = 0 < t$, so output = 0.
- This mimics the behavior of an OR gate.



NOT

The NOT gate inverts the input, outputting `1` if the input is `0`, and `0` if the input is `1`.

Input x	Weighted Sum $w \cdot x$	Threshold $t = -0.5$	Output y
0	$(-1) \cdot 0 = 0$	$0 \geq -0.5$	1
1	$(-1) \cdot 1 = -1$	$-1 < -0.5$	0

- **Weight**: $w = -1$.
- **Threshold**: $t = -0.5$.
- **Behavior**:
  - For $x = 0$: $wx = 0 \geq t$, so output = 1.
  - For $x = 1$: $wx = -1 < t$, so output = 0.
- This mimics the behavior of a NOT gate.

What are the Learning Rules in ANN

- A learning rule is a method or mathematical logic.
- It helps a Neural Network to learn from the existing conditions and improve its performance.
- Thus learning rules update the weights and bias levels of a network when a network simulates in a specific data environment

The different learning rules in the Neural network

1. Hebbian learning rule – It identifies, how to modify the weights of nodes of a network.
2. Perceptron learning rule – The network starts its learning by assigning a random value to each weight.
3. Delta learning rule – Modification in sympatric weight of a node is equal to the multiplication of error and the input.
4. Correlation learning rule – The correlation rule is supervised learning.
5. Outstar learning rule – We can use it when it assumes that nodes or neurons in a network are arranged in a layer.

Learning Rules in ANN

Hebbian Learning Rule: "Neurons that fire together wire together"

Donald Hebb

- The Hebbian rule was the first learning rule.
- In 1949 Donald Hebb developed it as a learning algorithm of the unsupervised neural network.

$$W_{ij} = x_i * x_j$$

- The Hebb learning rule assumes that – If two neighbor neurons are activated and deactivated at the same time, then the weight connecting these neurons should increase.
- At the start, values of all weights are set to zero.
- This learning rule can be used for both soft- and hard-activation functions.
- Since desired responses of neurons are not used in the learning procedure, this is the unsupervised learning rule.
- The absolute values of the weights are usually proportional to the learning time, which is undesired.

Learning Rules in ANN

Perceptron Learning Rule: "To classify linearly separable data"

- Each connection in a neural network has an associated weight, which changes in the course of learning.
- According to it, an example of supervised learning, the network starts its learning by assigning a random value to each weight.

$$\sum_i \sum_j \left(E_{ij} - O_{ij}\right)^2$$

- Calculate the output value on the basis of a set of records for which we can know the expected output value.
- The network then compares the calculated output value with the expected value.
- Next calculates an error function $\in$, which can be the sum of squares of the errors occurring for each individual in the learning sample.
- E_{ij} and O_{ij} are the expected and obtained values of the j^{th} unit for the ith individual.
- The network then adjusts the weights of the different units, checking each time to see if the error function has increased or decreased.

Learning Rules in ANN

Delta Learning Rule: “A.k.a Gradient descent rule, minimizes the error between the target and the output”

- Developed by Widrow and Hoff, the delta rule, is one of the most common learning rule.
- It depends on supervised learning.

$$\Delta w_{ji} = \alpha(t_j - y_j)g'(h_j)x_i$$

where
α is the learning rate,
g' is the derivative of the activation function g,
t_j is the target output,
h_j is the weighted sum of the neuron's inputs obtained as $\sum x_i w_{ji}$,
y_j is the predicted output,
x_i is the ith input.

- For a neuron with a linear activation function, the derivative of the activation function is constant and so the delta rule in this case can be simplified as.

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

- The generalized delta rule is important in creating useful networks capable of learning complex relations between inputs and outputs.

Learning Rules in ANN

Correlation Learning Rule:

- The correlation learning rule is based on a similar principle as the Hebbian learning rule depends on supervised learning.
- It assumes that weights between responding neurons should be more positive, and weights between neurons with opposite reactions should be more negative.

$$\Delta w_{ij} = \eta x_i d_j$$

- Contrary to the Hebbian rule, the correlation rule is supervised learning.
- Where, d_j is the desired value of the output signal. This training algorithm usually starts with the initialization of weights to zero. Since assigning the desired weight by users, the correlation learning rule is an example of supervised learning.

Learning Rules in ANN

Out Star Learning Rule:

- The Out Star Learning Rule when we assume that nodes or neurons in a network are arranged in a layer.
- Here the weights connected to a certain node should be equal to the desired outputs for the neurons connected through those weights.

$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

$$W_{jk} = \begin{cases} \eta(y_k - w_{jk}) & \text{if node j wins the competition} \\ 0 & \text{if node j losses the competition} \end{cases}$$

- The out-start rule produces the desired response t for the layer of n nodes.
- This is a supervised training procedure because desired outputs must be known.

Learning Rule	Type	Key Idea
Hebbian Rule	Unsupervised	Strengthen connections for correlated input and output neurons.
Perceptron Rule	Supervised	Adjust weights using error between predicted and target outputs.
Delta Rule	Supervised	Update weights proportional to error and input (gradient descent-based).
Correlation Rule	Supervised	Strengthen weights based on input-target correlation.
Outstar Rule	Layered Networks	Adjust weights from a neuron to multiple neurons for output pattern learning.

Feature	Hebbian	Perceptron	Delta	Correlation	Outstar
Learning Type	Unsupervised	Supervised	Supervised	Supervised	Supervised
Objective	Strengthen co-activated connections	Classify linearly separable data	Minimize error (gradient descent)	Strengthen based on correlation	Produce desired layer outputs
Weight Update Rule	$\eta \cdot x_i \cdot y_j$	$\eta \cdot (d - y) \cdot x_i$	$\eta \cdot (d - y) \cdot x_i$	$\eta \cdot d_j \cdot x_i$	$\eta \cdot (t_j - w_{ij})$
Activation Function	Soft/Hard	Hard	Soft	Soft/Hard	Linear
Initialization	Zero	Random	Random	Zero	Zero
Key Applications	Pattern recognition	Binary classification	Multi-layer networks	Supervised classification	Layered networks

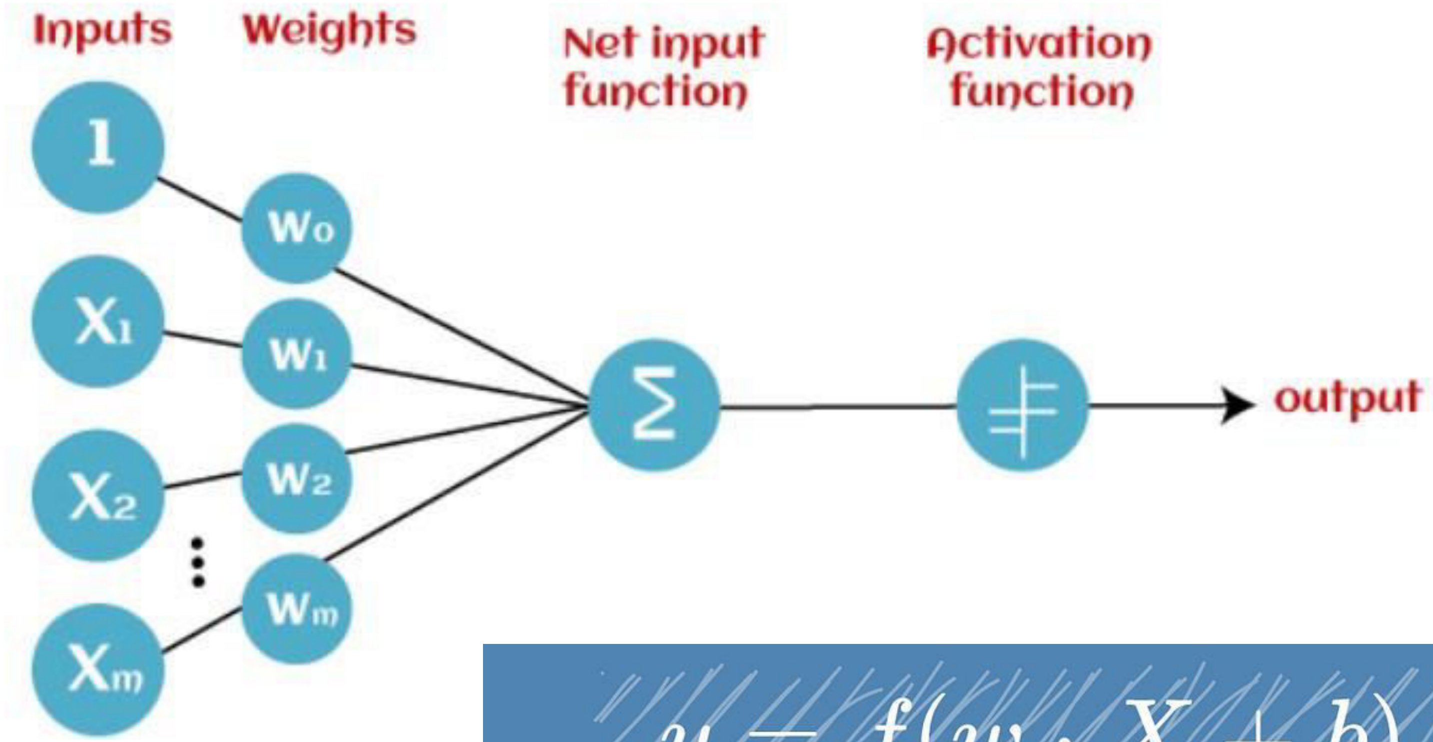
Why Multi-layered Perceptron(MLP)

- The perceptron is a classification algorithm. Specifically, it works as a linear binary classifier. It was invented in the late 1950s by Frank Rosenblatt.
- The basic building block (or) unit of the neural network.
- The perceptron basically works as a threshold function — non-negative outputs are put into one class while negative ones are put into the other class.

Why Multi-layered Perceptron(MLP)

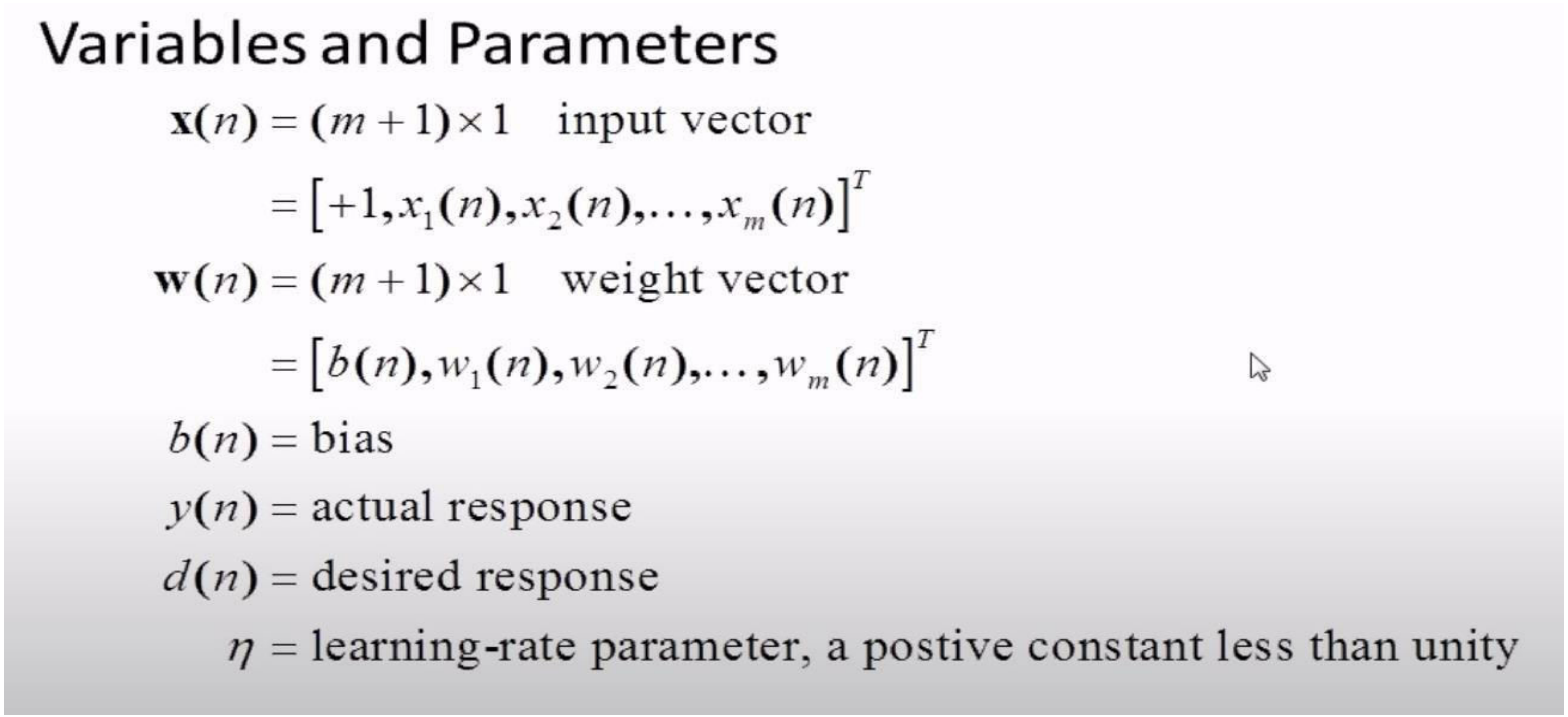
Perceptron –Components

- Input nodes
- Output node
- *An activation function*
- Weights and biases
- Error function



$$y = f(w \cdot X + b)$$

Perceptron Convergence (Learning) Algorithm



Variables and Parameters

$$\mathbf{x}(n) = (m+1)\times 1 \quad \text{input vector}$$

$$= \left[+1, x_1(n), x_2(n), \ldots, x_m(n)\right]^T$$

$$\mathbf{w}(n) = (m+1)\times 1 \quad \text{weight vector}$$

$$= \left[b(n), w_1(n), w_2(n), \ldots, w_m(n)\right]^T$$

$$b(n) = \text{bias}$$

$$y(n) = \text{actual response}$$

$$d(n) = \text{desired response}$$

$$\eta = \text{learning-rate parameter, a postive constant less than unity}$$

Perceptron Training Rule

The learning problem is to determine a weight vector that causes the perceptron to produce the correct + 1 or - 1 output for each of the given training examples.

1. **Initialization**: Start with random weight values.
2. **Iterative Learning**: For each training example, calculate the perceptron's output (+1 or -1).
3. **Error Detection**: If the perceptron misclassifies an example, update the weights using the formula:

$$w_i \leftarrow w_i + \Delta w_i$$

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

where:

- η: Learning rate
- t: Target output
- o: Observed output
- x_i: Input value

4. **Repeat**: Continue iterating through the training examples until all are correctly classified.

This rule guarantees that if the data is **linearly separable**, the perceptron will find a weight vector that separates the classes.

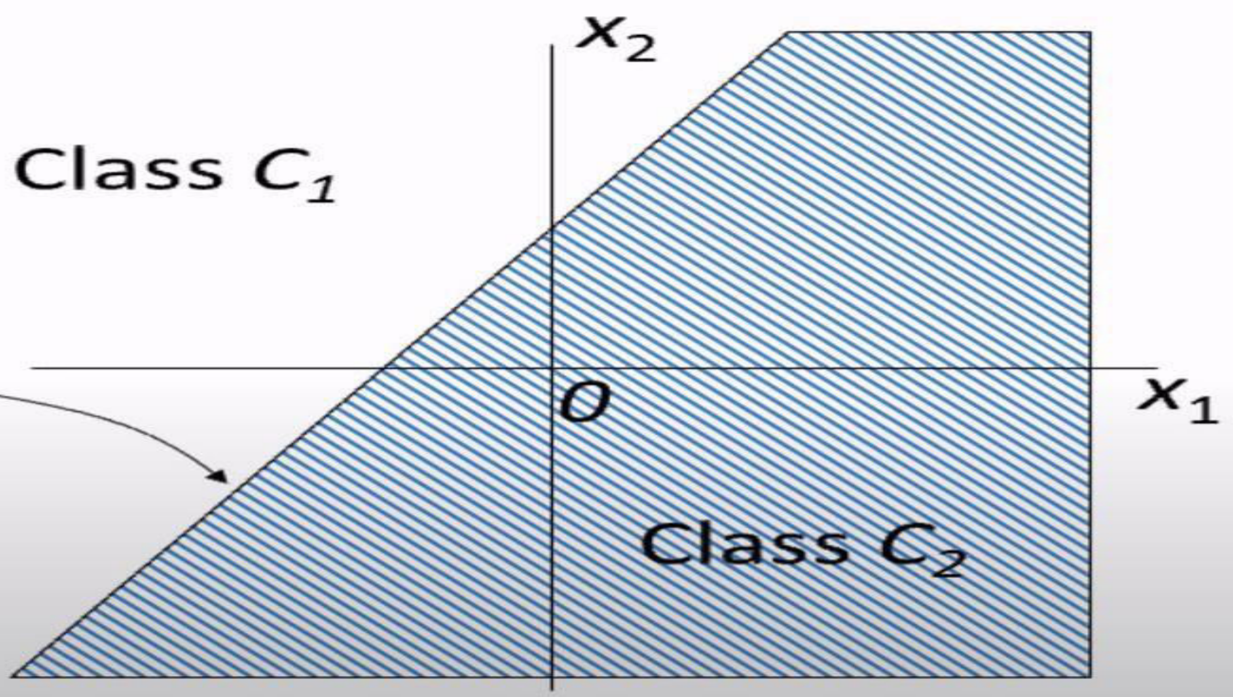
Decision Boundary

The hyper-plane

$$\sum_{i=1}^{m} w_i x_i + b = 0$$

or

$$w_1 x_1 + w_2 x_2 + b = 0$$



is the decision boundary for a two class classification problem.

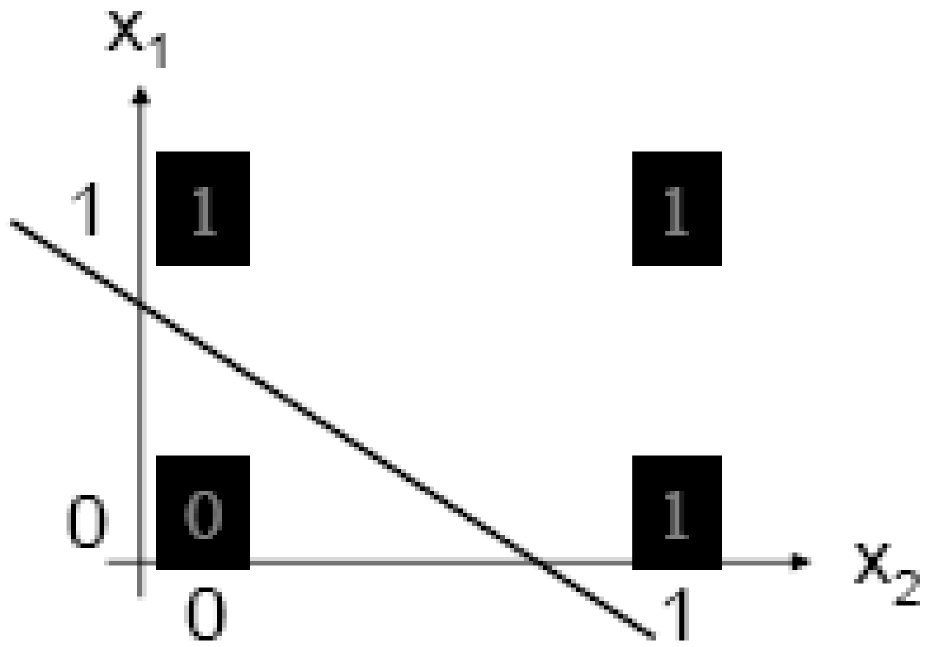
- The decision boundary is always orthogonal to the weight vector.
- Single-layer perceptron can only classify linearly separable vectors.

Decision Boundary

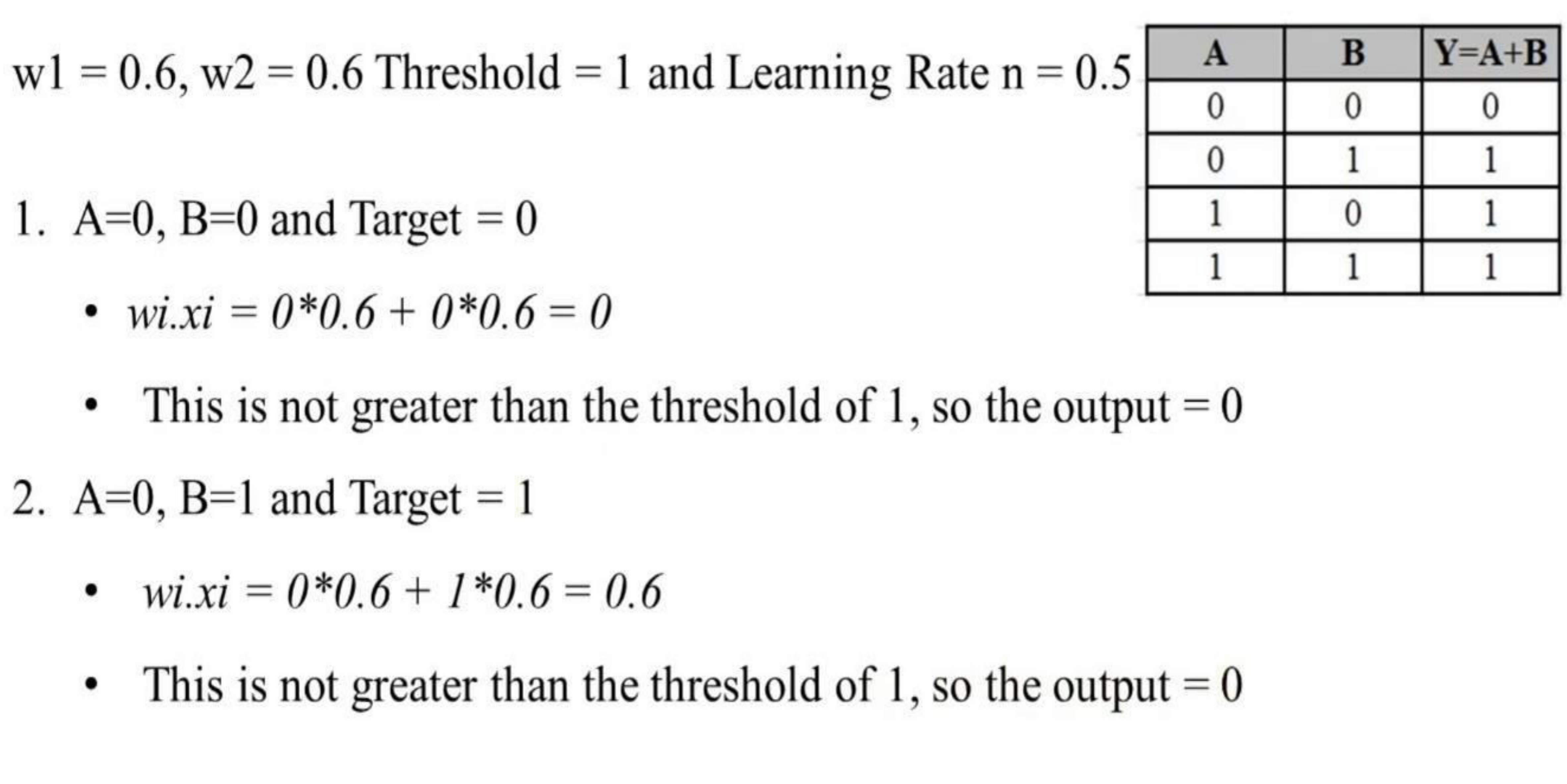
- A perceptron (threshold unit) can *learn anything* that it can *represent* (i.e. anything separable with a hyperplane)

OR function

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



OR GATE Perceptron Training Rule

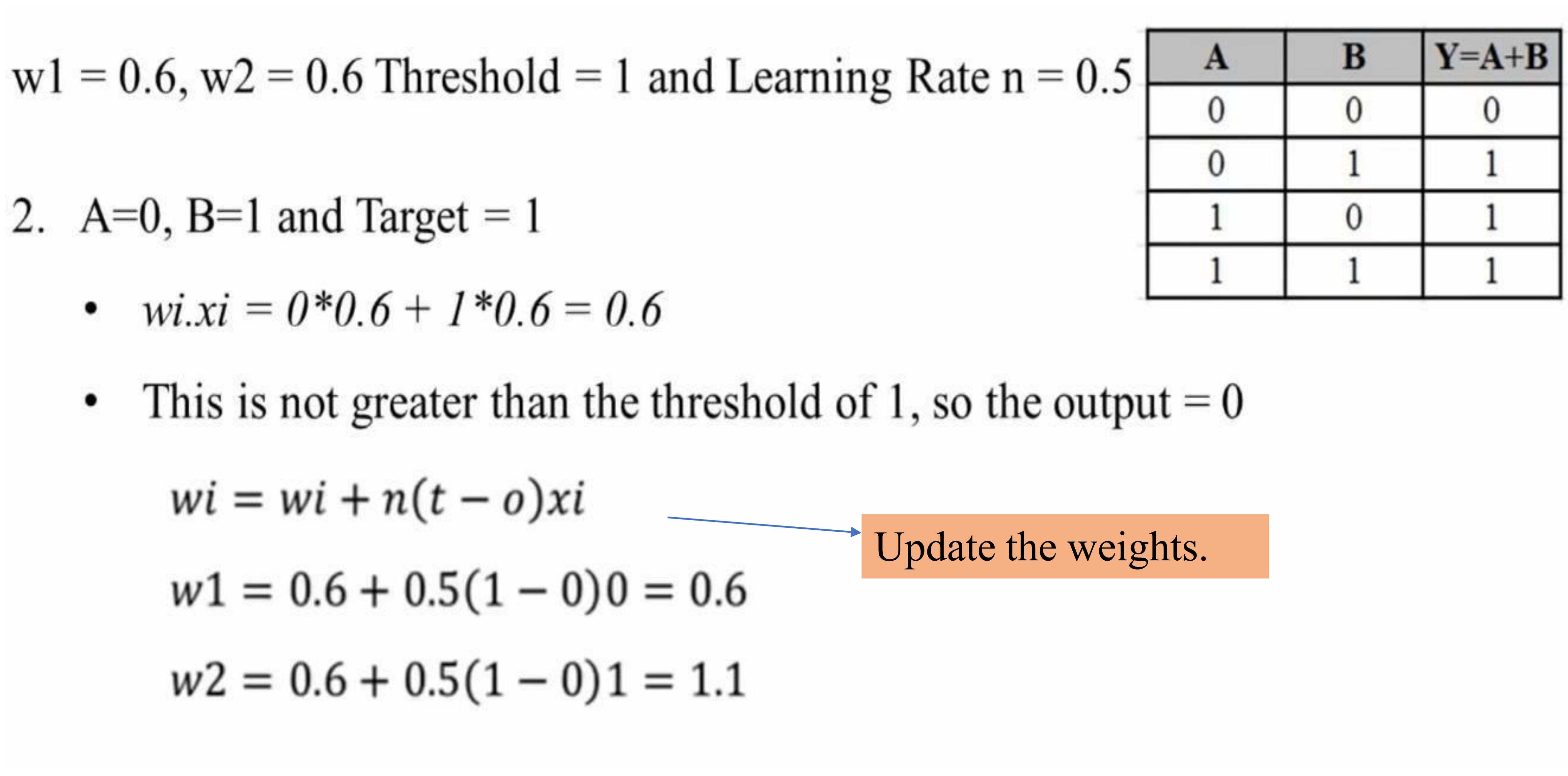


w1 = 0.6, w2 = 0.6 Threshold = 1 and Learning Rate n = 0.5

A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

1. A=0, B=0 and Target = 0
   - *wi.xi = 0*0.6 + 0*0.6 = 0*
   - This is not greater than the threshold of 1, so the output = 0
2. A=0, B=1 and Target = 1
   - *wi.xi = 0*0.6 + 1*0.6 = 0.6*
   - This is not greater than the threshold of 1, so the output = 0

OR GATE Perceptron Training Rule



w1 = 0.6, w2 = 0.6 Threshold = 1 and Learning Rate n = 0.5

A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

2. A=0, B=1 and Target = 1

- *wi.xi = 0*0.6 + 1*0.6 = 0.6*
- This is not greater than the threshold of 1, so the output = 0

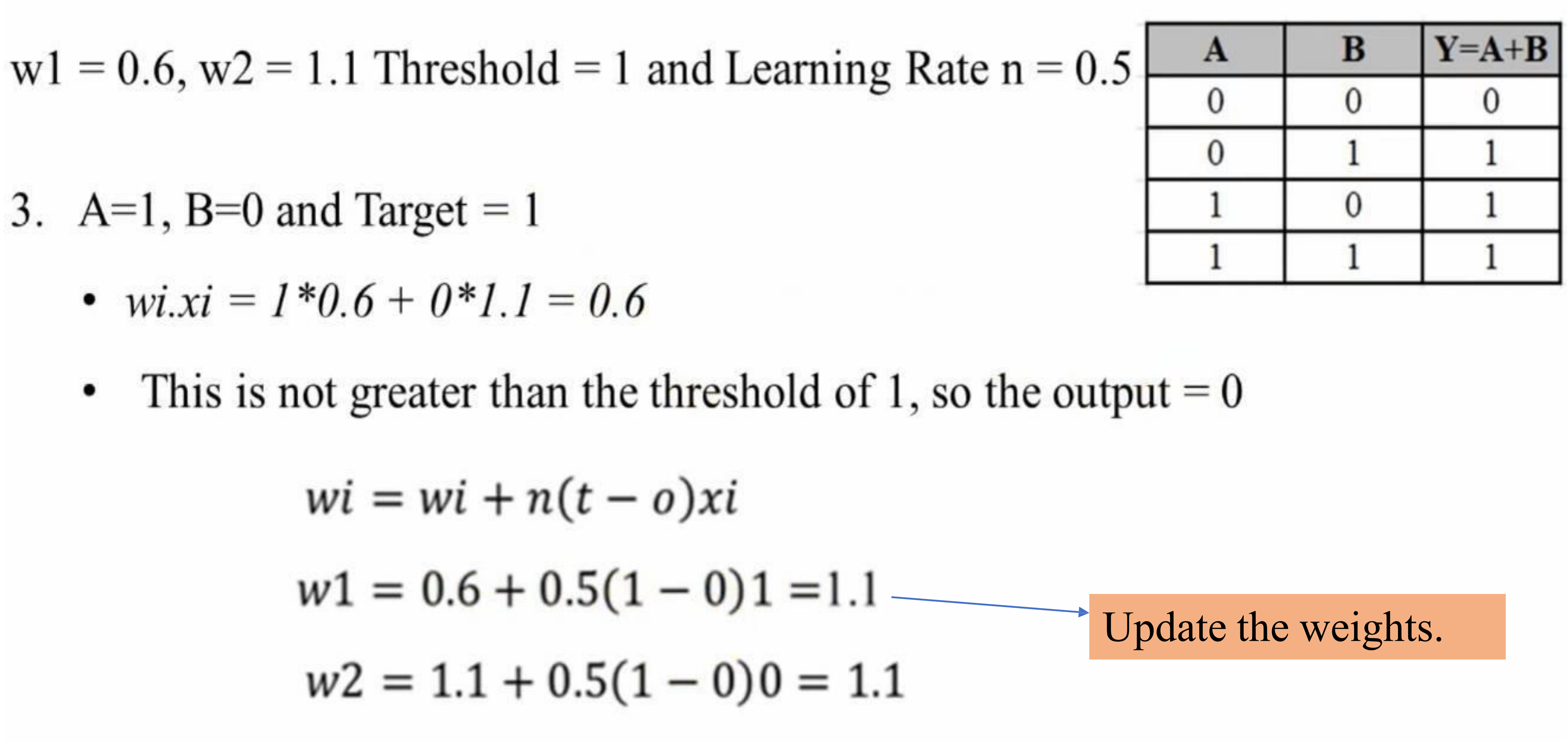
$$wi = wi + n(t - o)xi$$

$$w1 = 0.6 + 0.5(1 - 0)0 = 0.6$$

$$w2 = 0.6 + 0.5(1 - 0)1 = 1.1$$

Update the weights.

OR GATE Perceptron Training Rule



w1 = 0.6, w2 = 1.1 Threshold = 1 and Learning Rate n = 0.5

A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

3. A=1, B=0 and Target = 1
   - *wi.xi = 1*0.6 + 0*1.1 = 0.6*
   - This is not greater than the threshold of 1, so the output = 0

$$wi = wi + n(t - o)xi$$

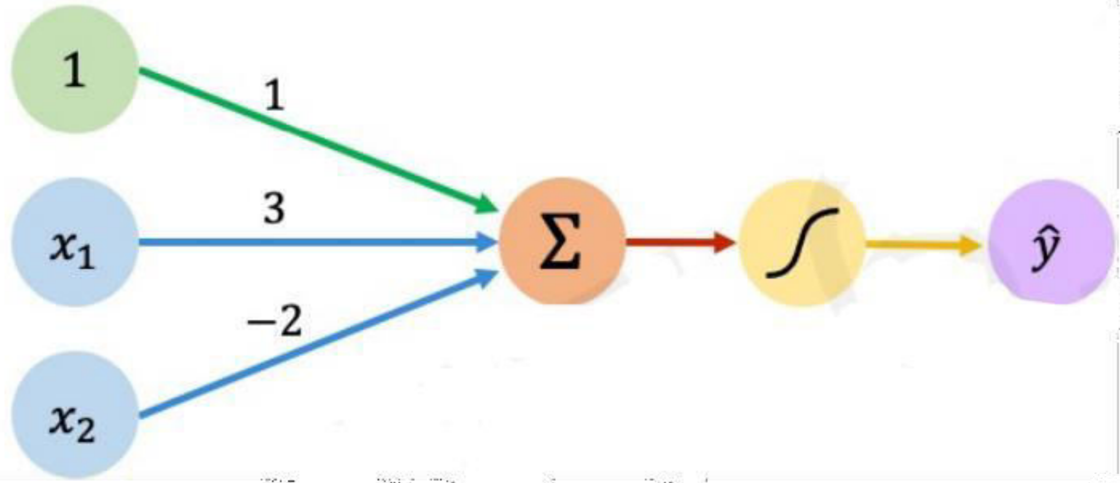
$$w1 = 0.6 + 0.5(1 - 0)1 = 1.1$$

$$w2 = 1.1 + 0.5(1 - 0)0 = 1.1$$

Update the weights.

Example

We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

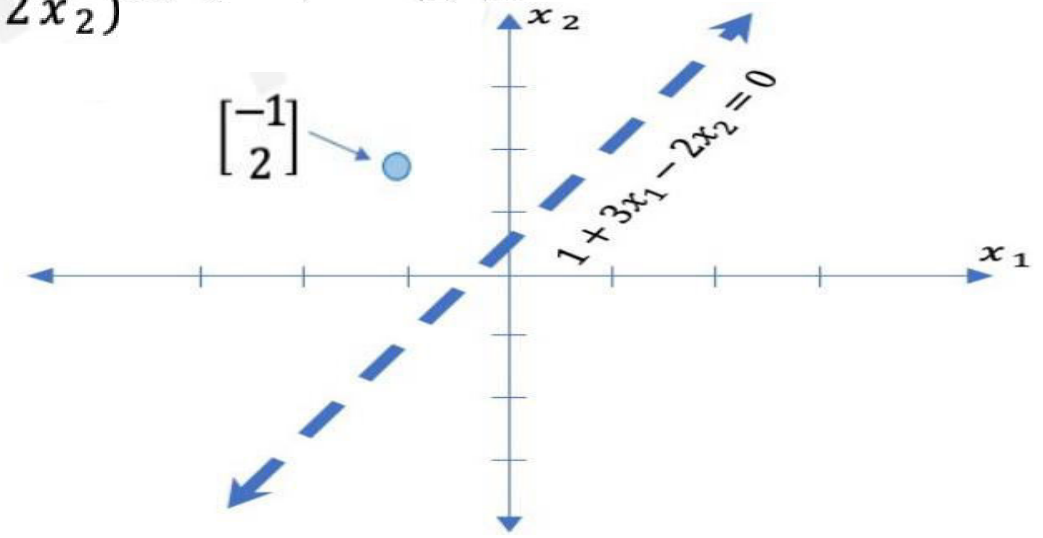


$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is just a line in 2D!}})$$

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\hat{y} = g\left(1 + (3 * -1) - (2 * 2)\right)$$
$$= g(-6) \approx 0.002$$

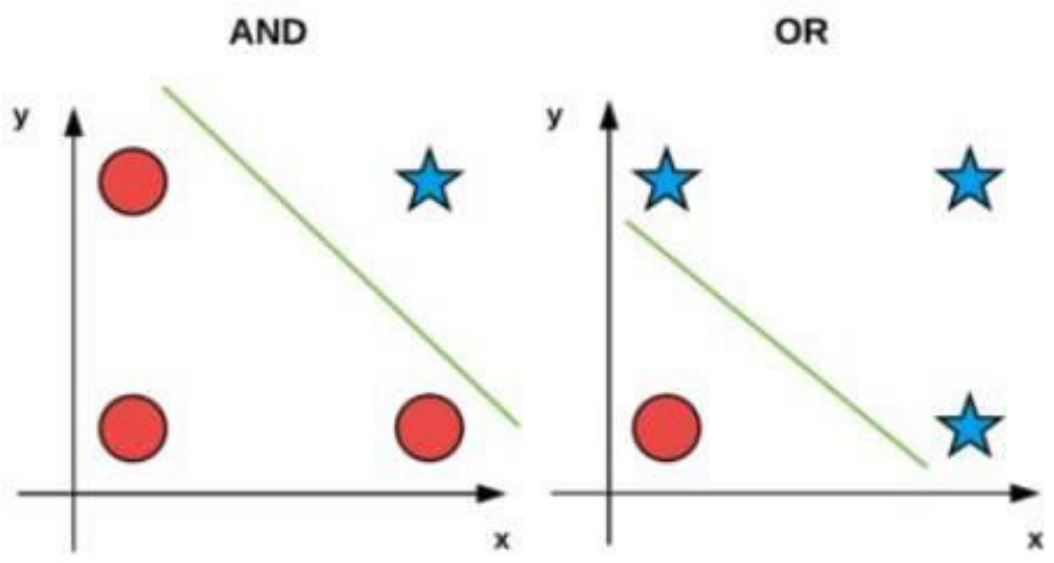


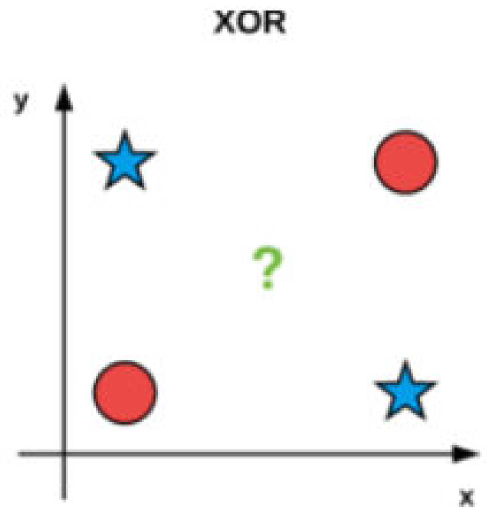
Why Multi-layered Perceptron(MLP)

Classification

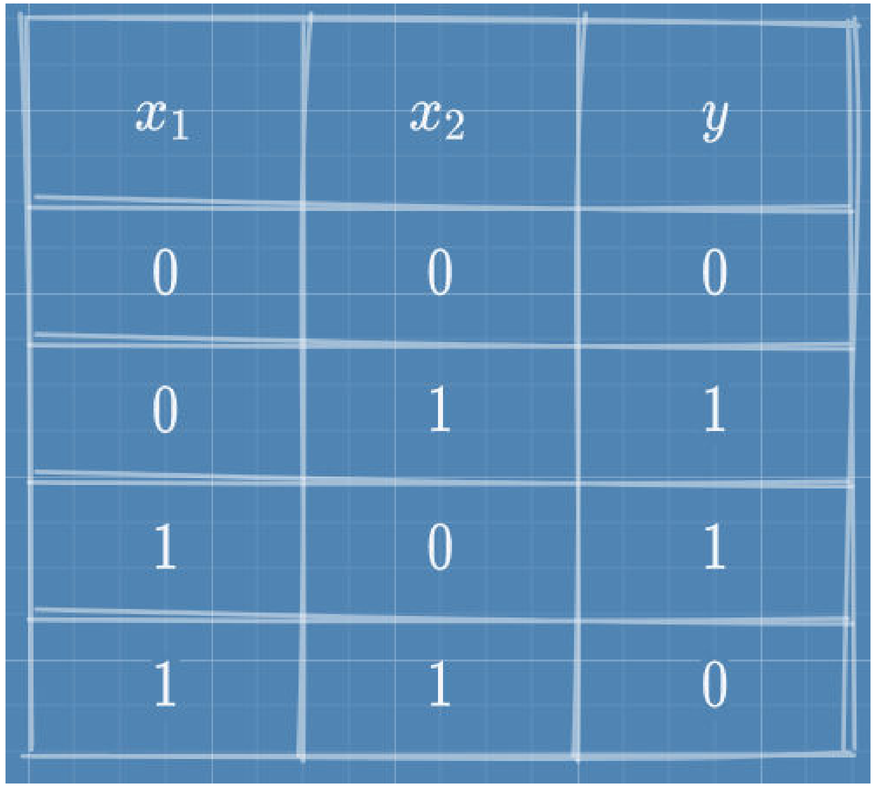
$$Class_1 : w \cdot X + b \geq 0$$

$$Class_2 : w \cdot X + b < 0$$



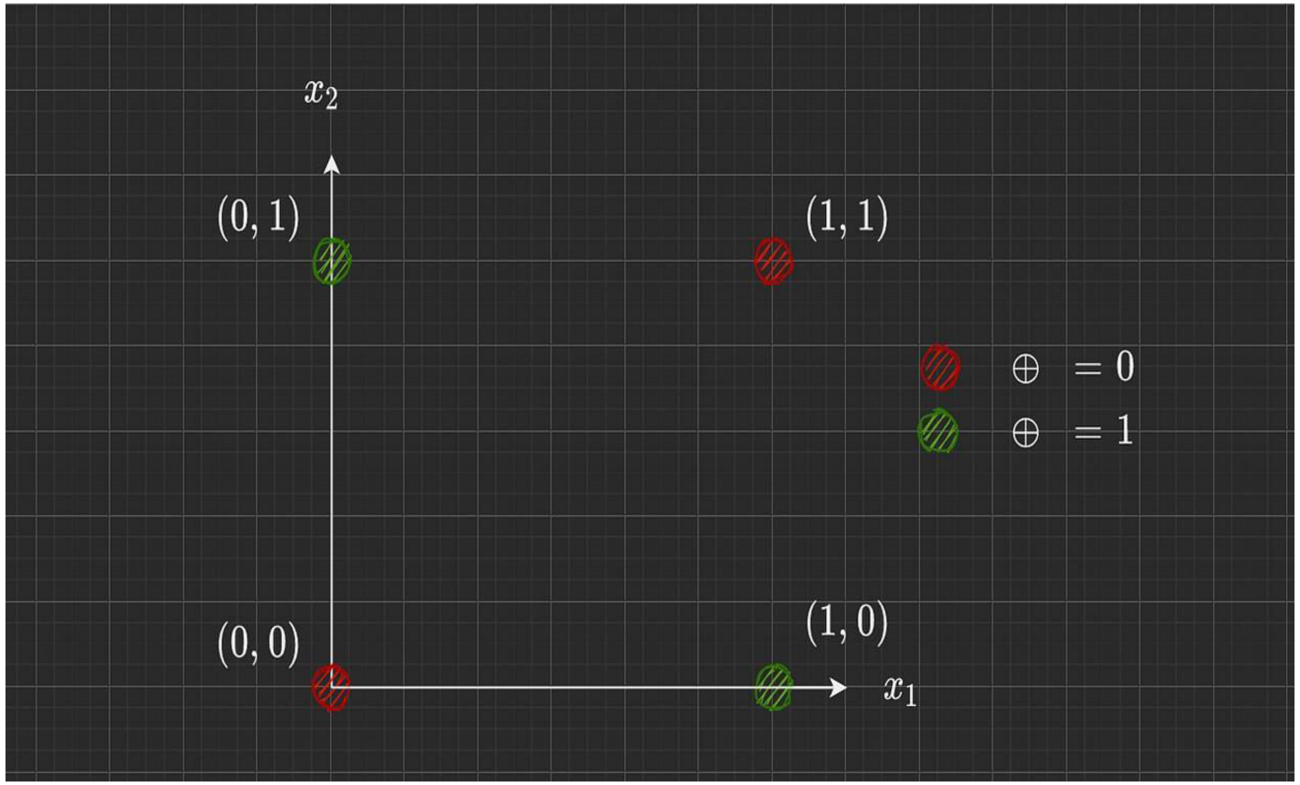


Model to Mimic XoR Problem



x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

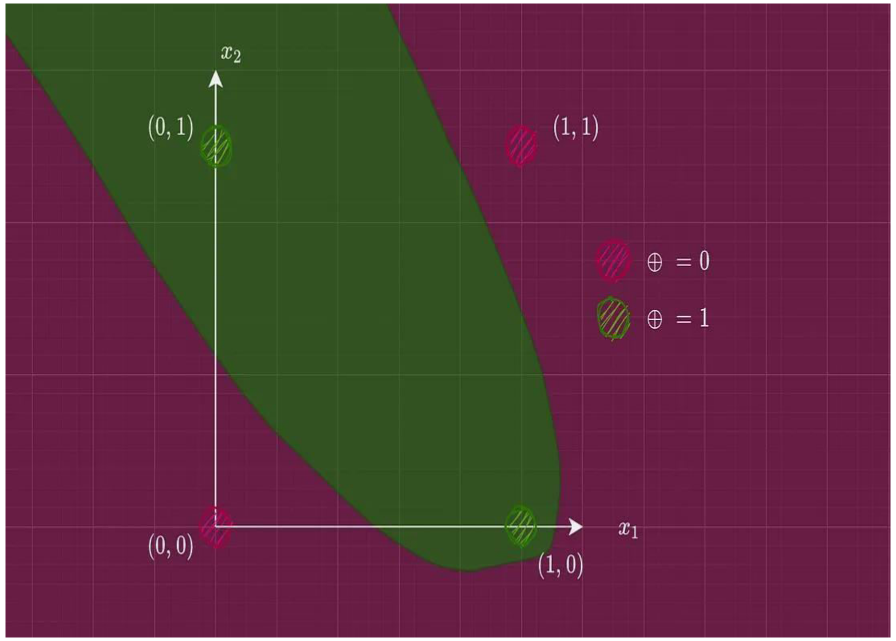
The XOR function



The XOR output plot

Attempt #1: The Single Layer Perceptron

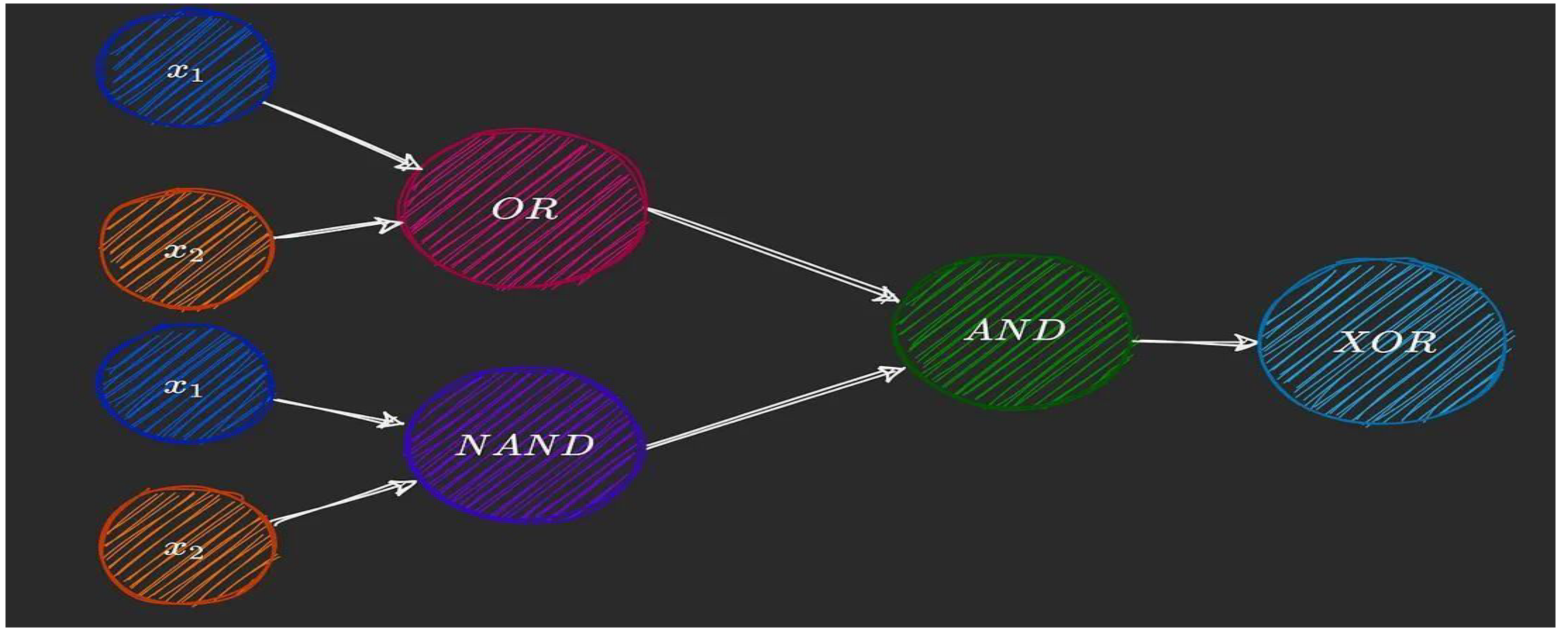
- A perceptron can only converge on linearly separable data. Therefore, it isn't capable of imitating the XOR function.
- A perceptron must correctly classify the entire training data in one go.
- Non-linearity allows for more complex decision boundaries. One potential decision boundary for our XOR data could look like this.

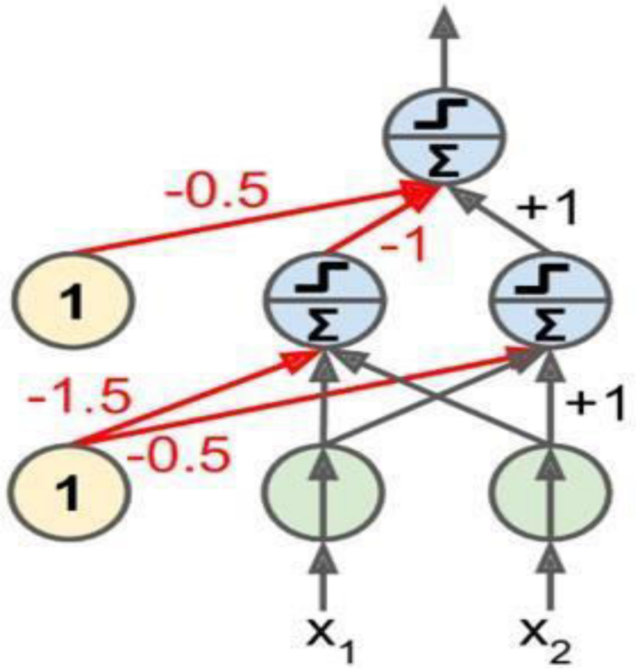


The 2d XOR Problem — Attempt #2

The XOR function can be condensed into two parts (**NAND and an OR**)

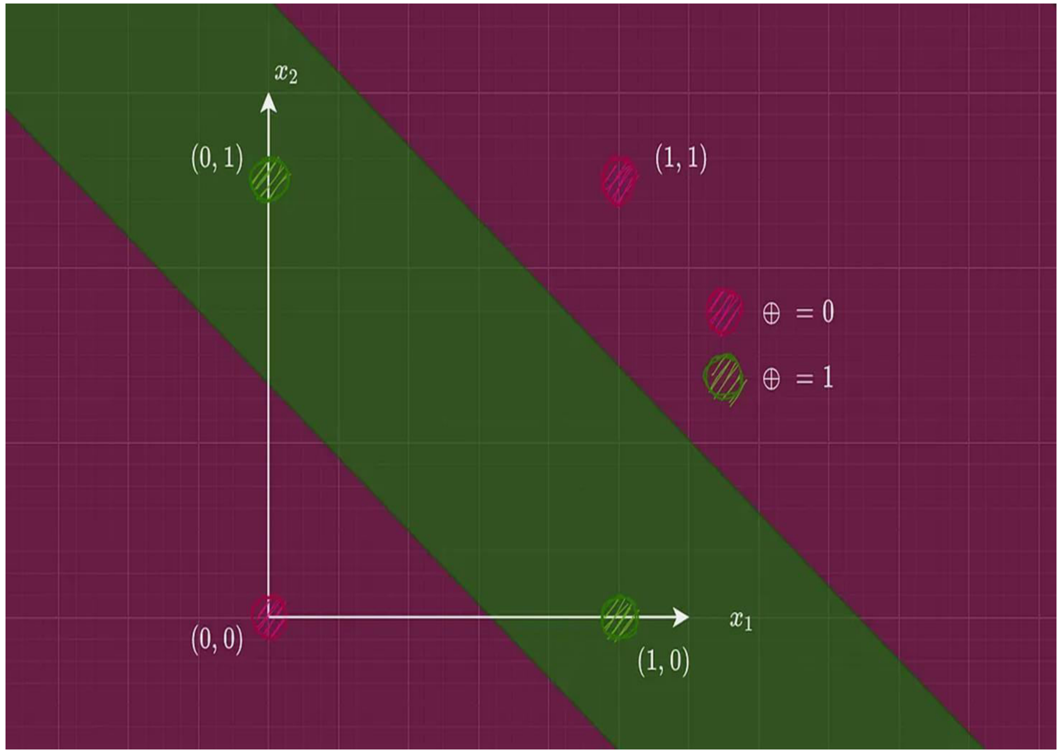
$$XOR(x_1, x_2) = (x_1 + x_2) \cdot (\overline{x_1 x_2})$$





The 2d XOR Problem — Attempt #2

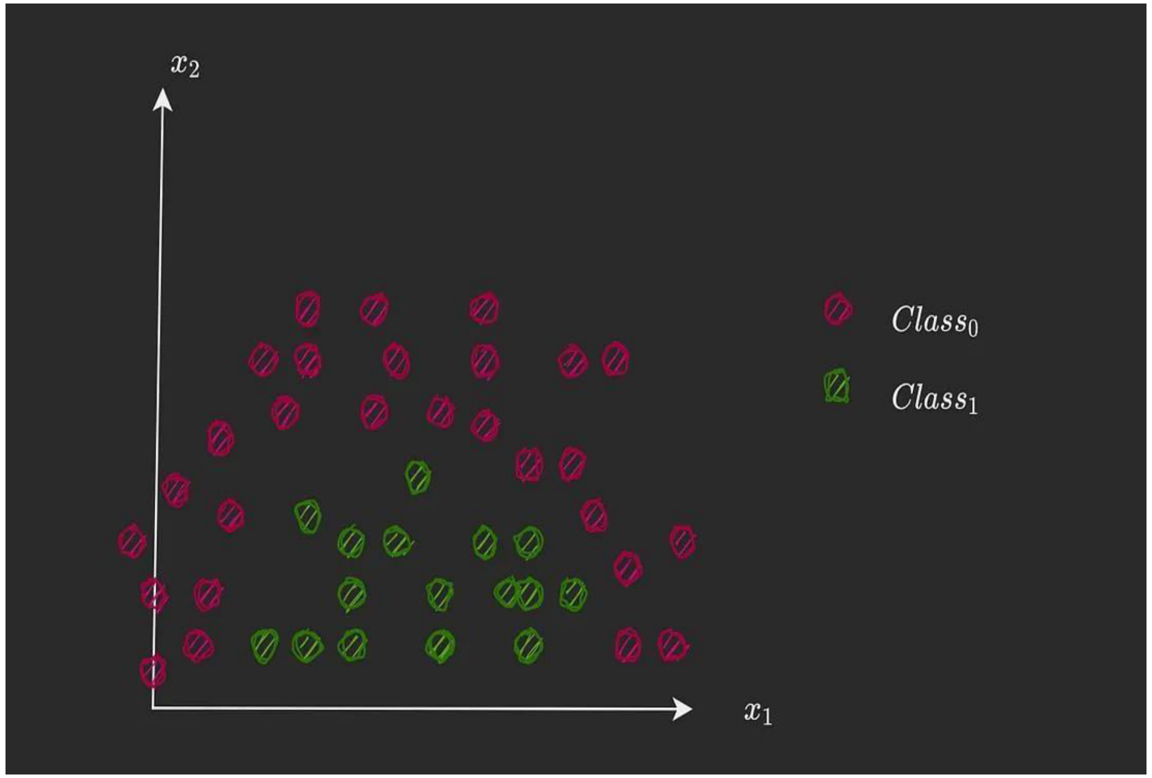
- Imitating the XOR function would require a non-linear decision boundary.
- The XOR problem with neural networks can be solved by using Multi-Layer Perceptrons or a neural network architecture with an input layer, hidden layer, and output layer.
- So during the forward propagation through the neural networks, the weights get updated to the corresponding layers and the XOR logic gets executed.



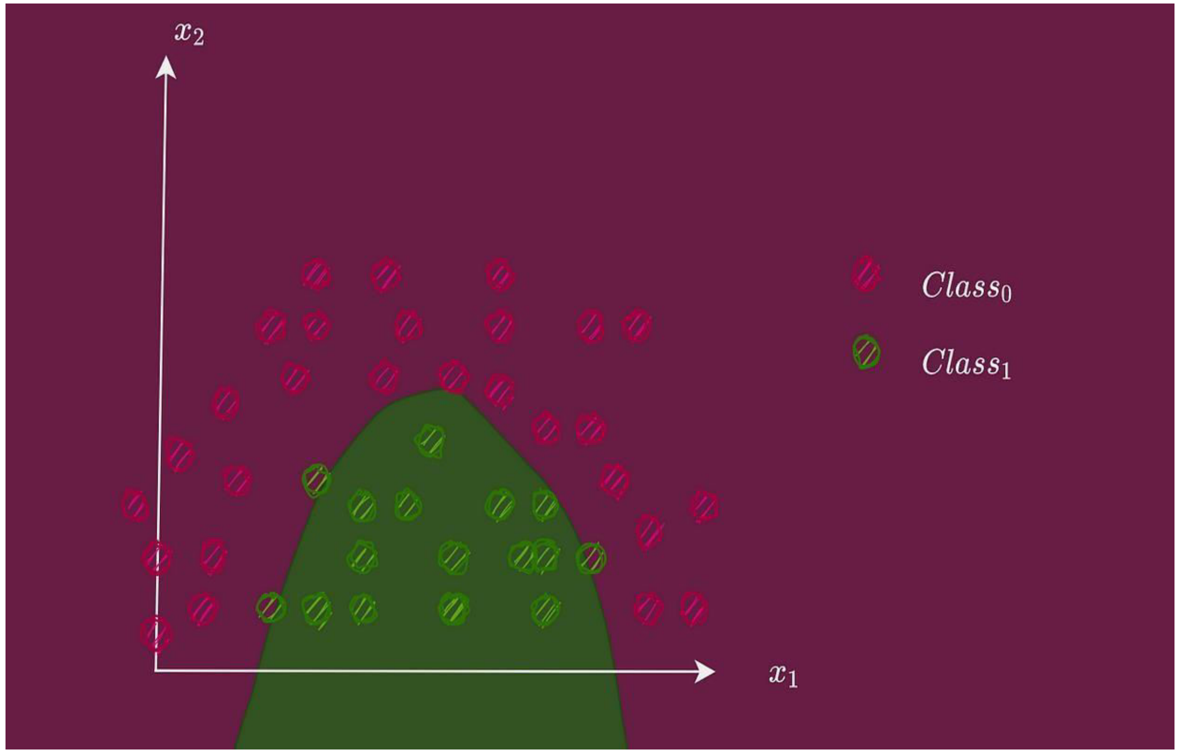
AND of our OR and NAND models

Why Multi-layered Perceptron(MLP) Cont'd

- The XOR and XNOR gates are the only ones that are not linearly separable.



A binary classification problem in two dimensions



A potential decision boundary

Why Multi-layered Perceptron(MLP) Cont'd

- The single neurons are not able to solve complex tasks (e.g. restricted to linear calculations)
- Creating networks by hand is too expensive; we want to learn from data.
- Nonlinear features also have to be generated by hand; tessellations become intractable for larger dimensions.

➢ We want to have a generic model that can adapt to some training data.

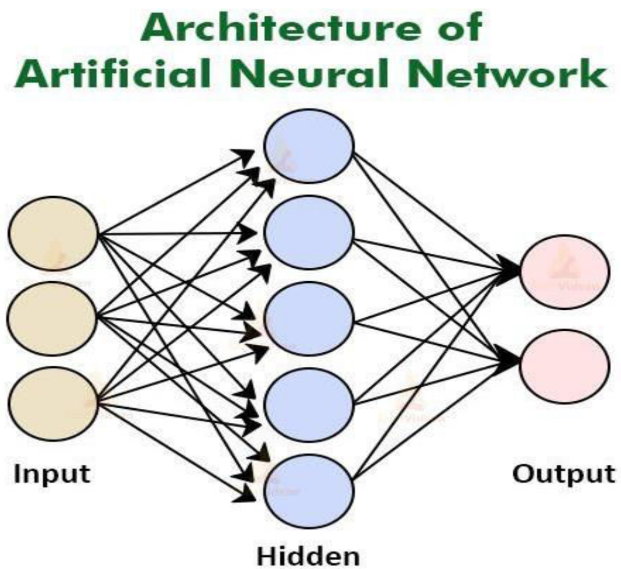
➢ Basic idea: multi-layer perceptron (Werbos 1974, Rumelhart, McClelland, Hinton 1986), also named feedforward networks

The Multi-layered Perceptron

- The simplest kind of feed-forward network is a multilayer perceptron (MLP)
- A multi-layer perceptron (MLP) is a finite acyclic graph.
- the units are arranged into a set of layers, and each layer contains some number of identical units.
- Every unit in one layer is connected to every unit in the next layer; we say that the network is fully connected.

Layers

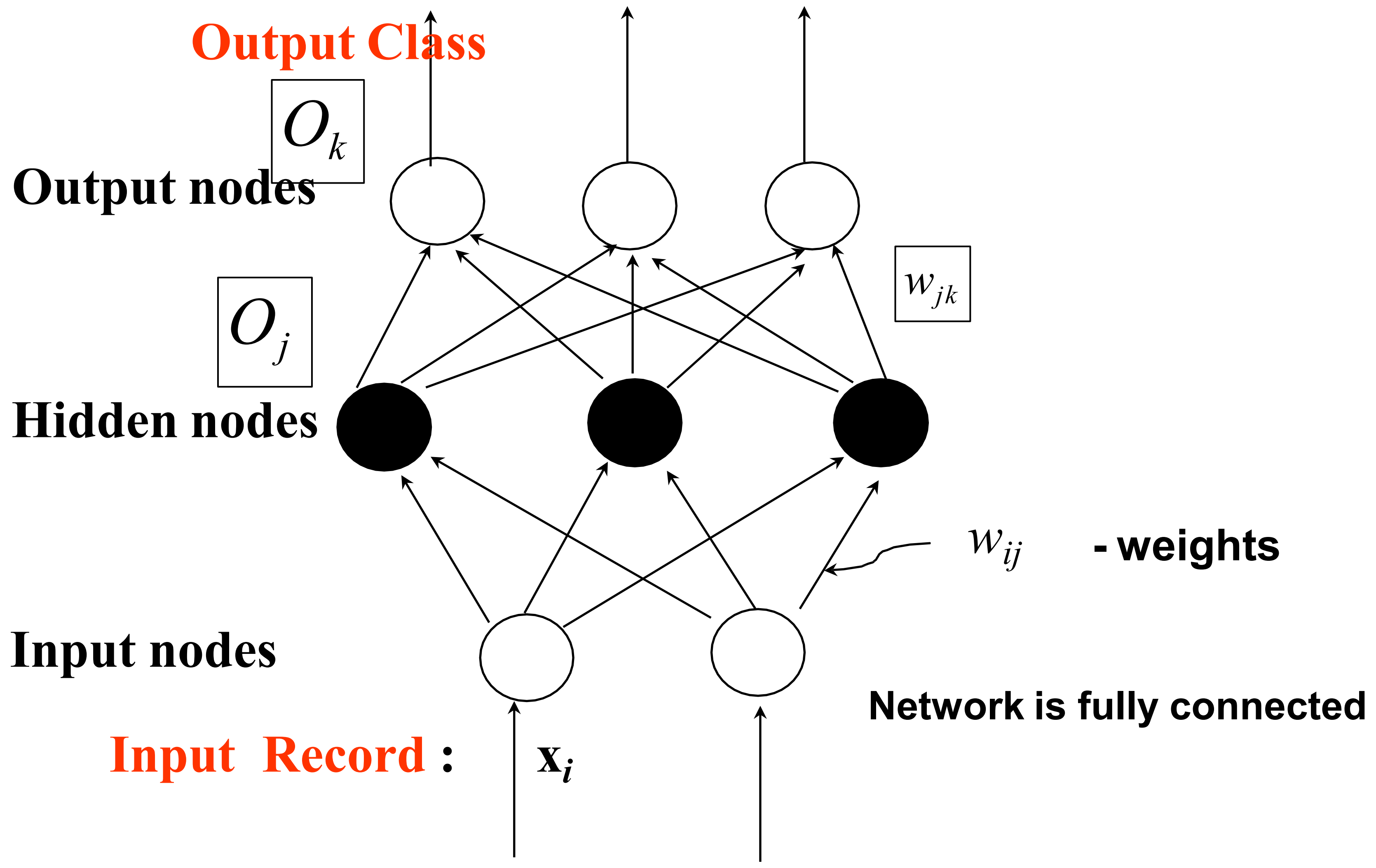
- Input layer
- Output layer
- Hidden layer



What is Multilayer Perceptron

- A multilayer perceptron (MLP) is a group of perceptrons, organized in multiple layers, that can accurately answer complex questions.
- Each perceptron in the first layer (on the left) sends signals to all the perceptrons in the second layer, and so on.
- An MLP contains an input layer, at least one hidden layer, and an output layer.
- No connections within a layer
- No direct connections between input and output layers
- When an ANN has two or more hidden layers, it is called a deep neural network (DNN).

A Multilayer Feed-Forward Neural Network



Neural Network Learning

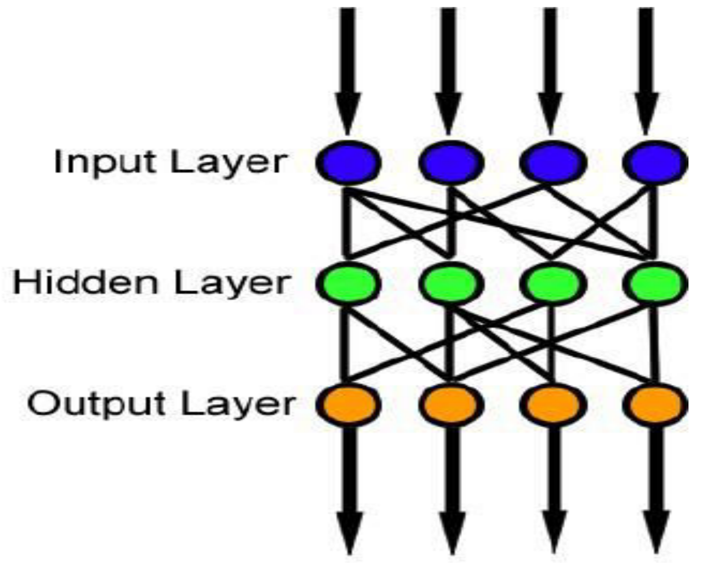
- The inputs are fed simultaneously into the input layer.
- The weighted outputs of these units are fed into the hidden layer.
- The weighted outputs of the last hidden layer are inputs to units making up the output layer.

A Multilayer Feed Forward Network

- The units in the hidden layers and output layer are sometimes referred to as neurodes, due to their symbolic biological basis, or as output units.
- A network containing two hidden layers is called a three-layer neural network, and so on.
- The network is feed-forward in that none of the weights cycles back to an input unit or to an output unit of a previous layer.

A Multilayer Feed Forward Network Cont'd

- Information always moves one direction.
  - No loops.
  - Never goes backwards.
  - Forms a directed acyclic graph.



- Each node receives input only from immediately preceding layer.
- Simplest type of artificial neural network.

A Multilayered Feed –Forward Network

- INPUT: Records without class attributes with normalized attribute values.
- INPUT VECTOR: X = { x1, x2, …. xn}

  where n is the number of (non class) attributes.
- INPUT LAYER – there are as many nodes as non-class attributes i.e. as the length of the input vector.
- HIDDEN LAYER – the number of nodes in the hidden layer and the number of hidden layers depends on implementation.

A Multilayered Feed–Forward Network

- **OUTPUT LAYER** – corresponds to the class attribute.
- There are as many nodes as classes (values of the class attribute).

$$\boxed{O_k}$$

k= 1, 2,.. #classes

•Network is fully connected, i.e. each unit provides input to each unit in the next forward layer.

Classification by Backpropagation

- *Back Propagation learns by iteratively processing a set of training data (samples).*
- For each sample, weights are modified to minimize the error between the network's classification and the actual classification.

MLP Training

- Mean Square Error (MSE):

$$J(\boldsymbol{\theta}) = J(\mathbf{W}, \mathbf{b}) = \frac{1}{N} \sum_{i=1}^{N} \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|^2$$

  - It is suitable for regression and classification

- Categorical Cross Entropy Error:

$$J_{CCE} = -\sum_{i=1}^{N} \sum_{j=1}^{m} y_{ij} \log(\hat{y}_{ij})$$

  - It is suitable for classifiers that use softmax output layers

- Exponential Loss :

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} e^{-\beta \mathbf{y}_i^T \hat{\mathbf{y}}_i}$$

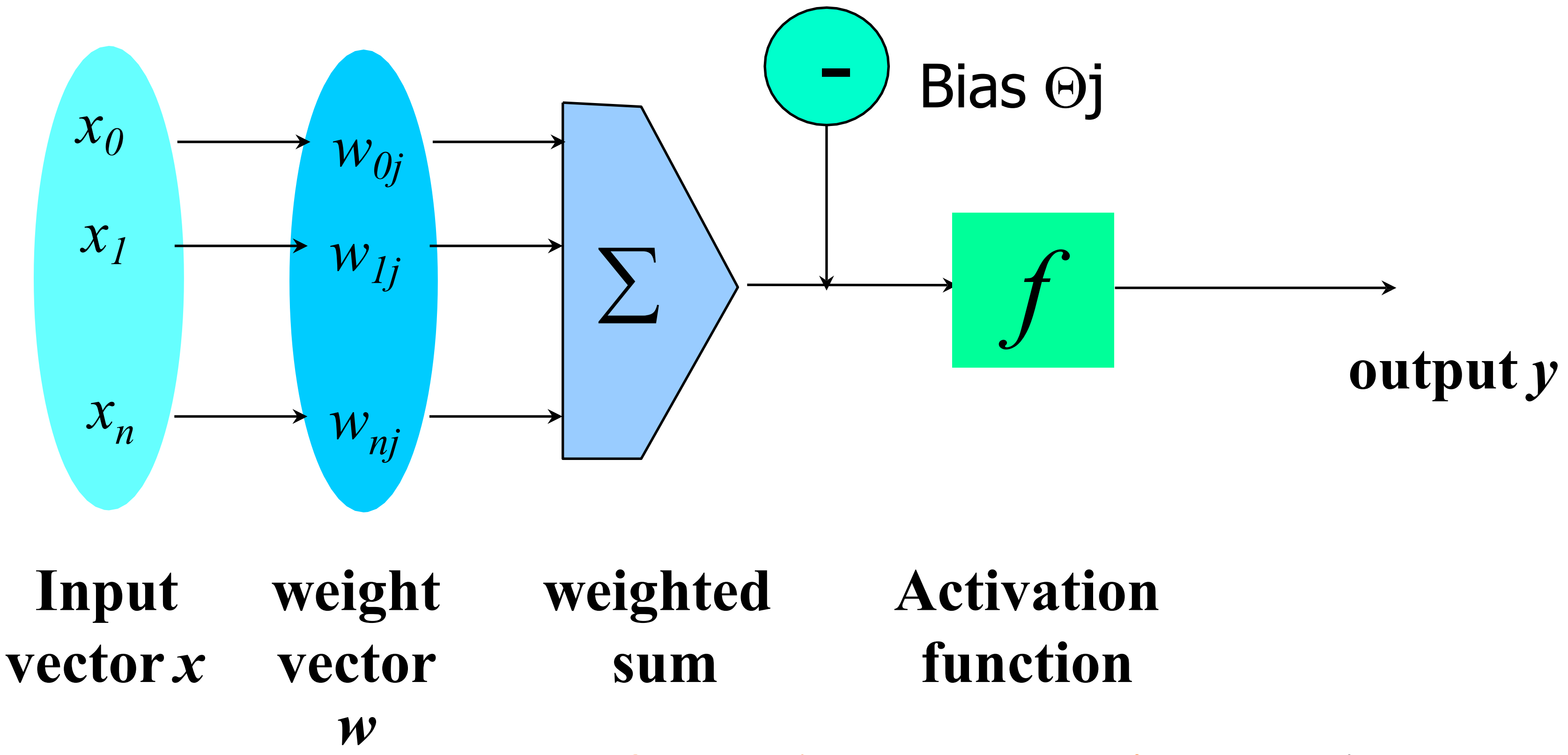
Steps in Back propagation Algorithm

- STEP ONE: initialize the weights and biases.
  - The weights in the network are initialized to random numbers from the interval [-1,1].
  - Each unit has a BIAS associated with it
  - The biases are similarly initialized to random numbers from the interval [-1,1].
- STEP TWO: feed the training sample.

Steps in Back propagation Algorithm (cont..)

- STEP THREE: Propagate the inputs forward; we compute the net input and output of each unit in the hidden and output layers.
- STEP FOUR: backpropagate the error.
- STEP FIVE: update weights and biases to reflect the propagated errors.
- STEP SIX: terminating conditions.

Propagation through Hidden Layer (One Node)



- The inputs to unit j are outputs from the previous layer. These are multiplied by their corresponding weights in order to form a weighted sum, which is added to the bias associated with unit j.
- A nonlinear activation function f is applied to the net input.

Propagate the inputs forward

- For unit j in the input layer, its output is equal to its input, that is,

$$\boxed{O_j =}$$

for input unit j.

- The net input to each unit in the hidden and output layers is computed as follows.
- Given a unit j in a hidden or output layer, the net input is

$$\boxed{I_j = \sum_i w_{ij} O_i + \theta_j}$$

where w_{ij} is the weight of the connection from unit i in the previous layer to unit j; Oi is the output of unit I from the previous layer;

$\boxed{\theta_j}$ is the bias of the unit

Propagate the inputs forward

- Each unit in the hidden and output layers takes its net input and then applies an activation function. The function symbolizes the activation of the neuron represented by the unit. It is also called a logistic, sigmoid, or squashing function.

$$\boxed{O_j = \frac{1}{1+e^{-I_j}}}$$

- Given a net input Ij to unit j, then

$$Oj = f(Ij),$$

the output of unit j, is computed as

Back propagate the error

- When reaching the Output layer, the error is computed and propagated backward.

- For a uni.t k in the output layer the error is computed by a formula:

$$\boxed{Err_k = O_k(1-O_k)(T_k - O_k)}$$

Where O_k – actual output of unit k (computed by an activation function.

$$\boxed{O_k = \frac{1}{1+e^{-I_k}}}$$

T_k – True output based of known class label; classification of training sample

$O_k(1-O_k)$ – is a Derivative (rate of change) of the activation function.

Back propagate the error

- The error is propagated backward by updating weights and biases to reflect the error of the network classification.
- For a unit j in the hidden layer the error is computed by a formula:

$$\boxed{Err_j = O_j(1 - O_j)\sum_k Err_k w_{jk}}$$

where w_{jk} is the weight of the connection from unit j to unit k in the next higher layer, and E_{rrk} is the error of unit k.

Update weights and biases

- Weights are updated by the following equations, where l is a constant between 0.0 and 1.0 reflecting the learning rate, this learning rate is fixed for implementation.

$$\boxed{\Delta w_{ij} = (l)}$$

$$\boxed{w_{ij} = w_{ij} + \Delta w_{ij}}$$

- Biases are updated by the following equations

$$\boxed{\Delta \theta_j = (l) Err_j}$$

$$\boxed{\theta_j = \theta_j + \Delta \theta_j}$$

Update weights and biases

- We are updating weights and biases after the presentation of each sample.
- This is called case updating.

- Epoch --- One iteration through the training set is called an epoch.

- Epoch updating ------------
- Alternatively, the weight and bias increments could be accumulated in variables and the weights and biases updated after all of the samples of the training set have been presented.

- Case updating is more accurate

Terminating Conditions

- Training stops

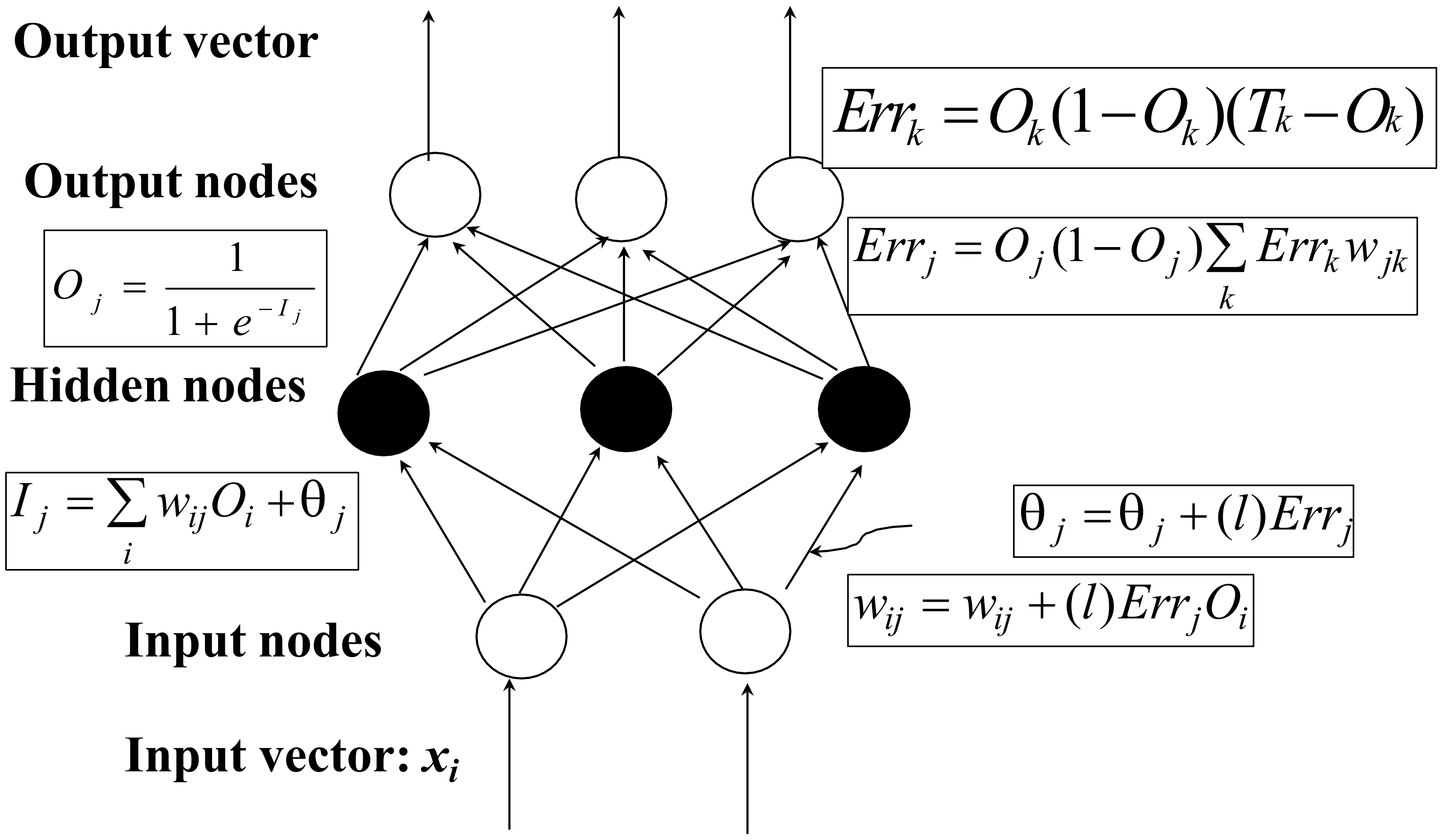
  - All $\boxed{\Delta w_{ij}}$ in the previous epoch are below some threshold, or

- The percentage of samples misclassified in the previous epoch is below some threshold, or

- a pre-specified number of epochs has expired.

- In practice, several hundreds of thousands of epochs may be required before the weights will converge.

Backpropagation Formulas



$$Err_k = O_k(1-O_k)(T_k-O_k)$$

$$Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}$$

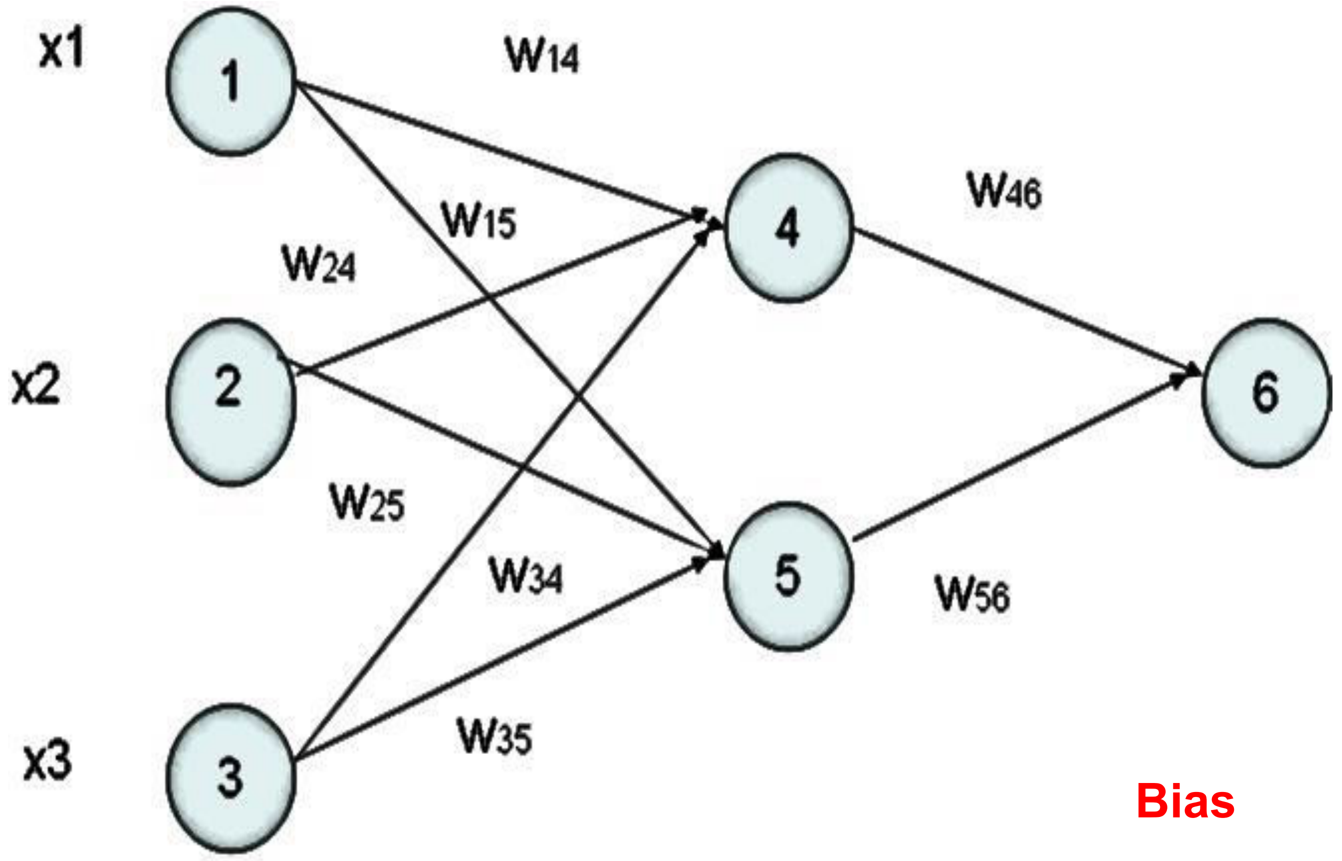
$$O_j = \frac{1}{1+e^{-I_j}}$$

$$I_j = \sum_i w_{ij}O_i + \theta_j$$

$$\theta_j = \theta_j + (l)Err_j$$

$$w_{ij} = w_{ij} + (l)Err_j O_i$$

Back propagation



Input = 3, Hidden Neuron = 2 Output =1

Initialize weights :

Random Numbers from -1.0 to 1.0

Initial Input and weight

x1	x2	x3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}
1	0	1	0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2

Bias

(Random Values from -1.0 to 1.0)

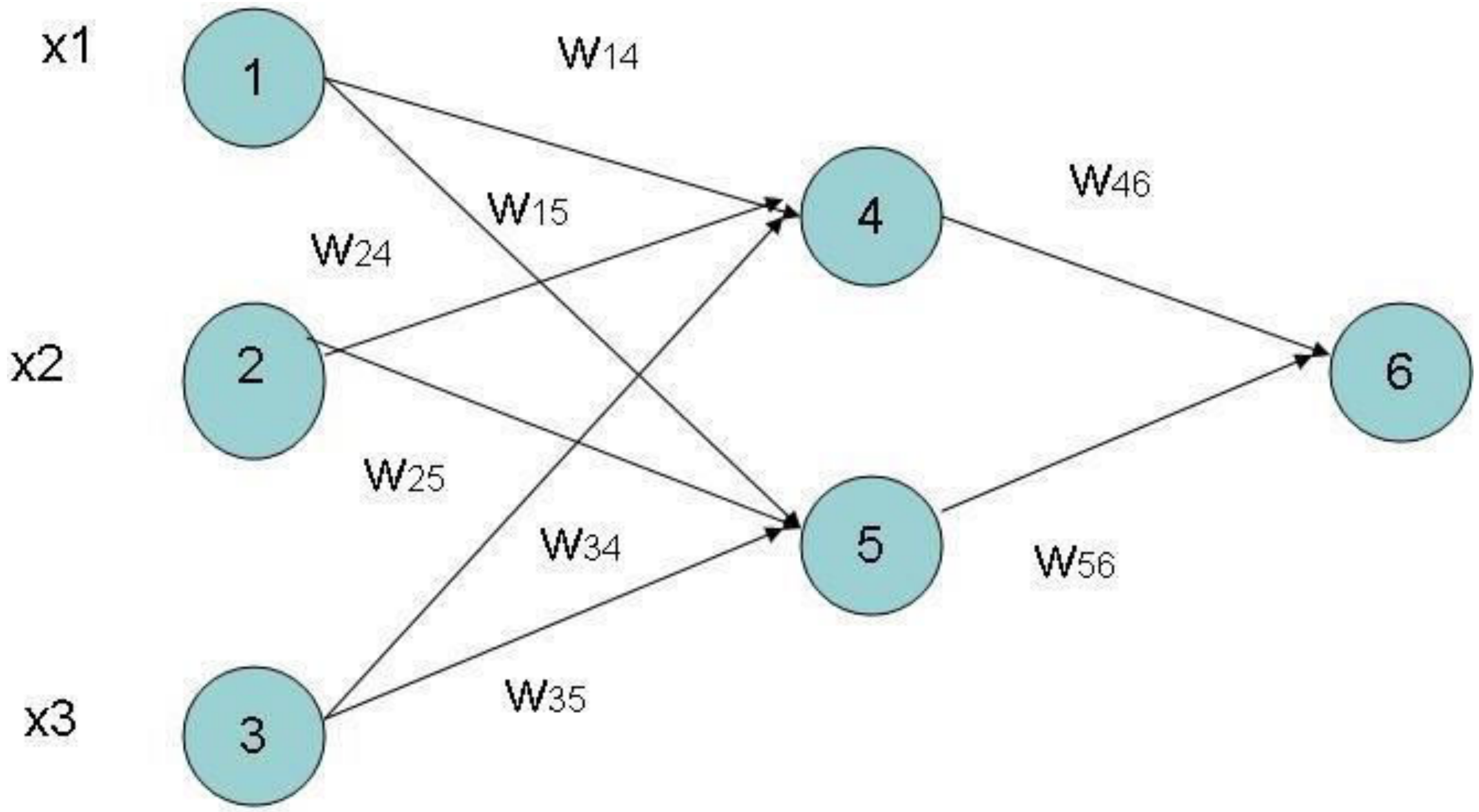
θ_4	θ_5	θ_6
-0.4	0.2	0.1

Example (cont..)

- Bias added to Hidden
- + Output nodes
- Initialize Bias
- Random Values from
- -1.0 to 1.0

- Bias (Random)

θ_4	θ_5	θ_6
-0.4	0.2	0.1



Net Input and Output Calculation

Unit j	Net Input I_j	Output O_j
4	**0.2 + 0 + 0.5 -0.4 = -0.7**	$\boxed{O_j = \frac{1}{1+e^{0.7}}}$ = 0.332
5	**-0.3 + 0 + 0.2 + 0.2 =0.1**	$\boxed{O_j = \frac{1}{1+e^{-0.1}}}$ = 0.525
6	**(-0.3)0.332-(0.2)(0.525)+0.1= -0.105**	$\boxed{O_j = \frac{1}{1+e^{0.105}}}$ = 0.475

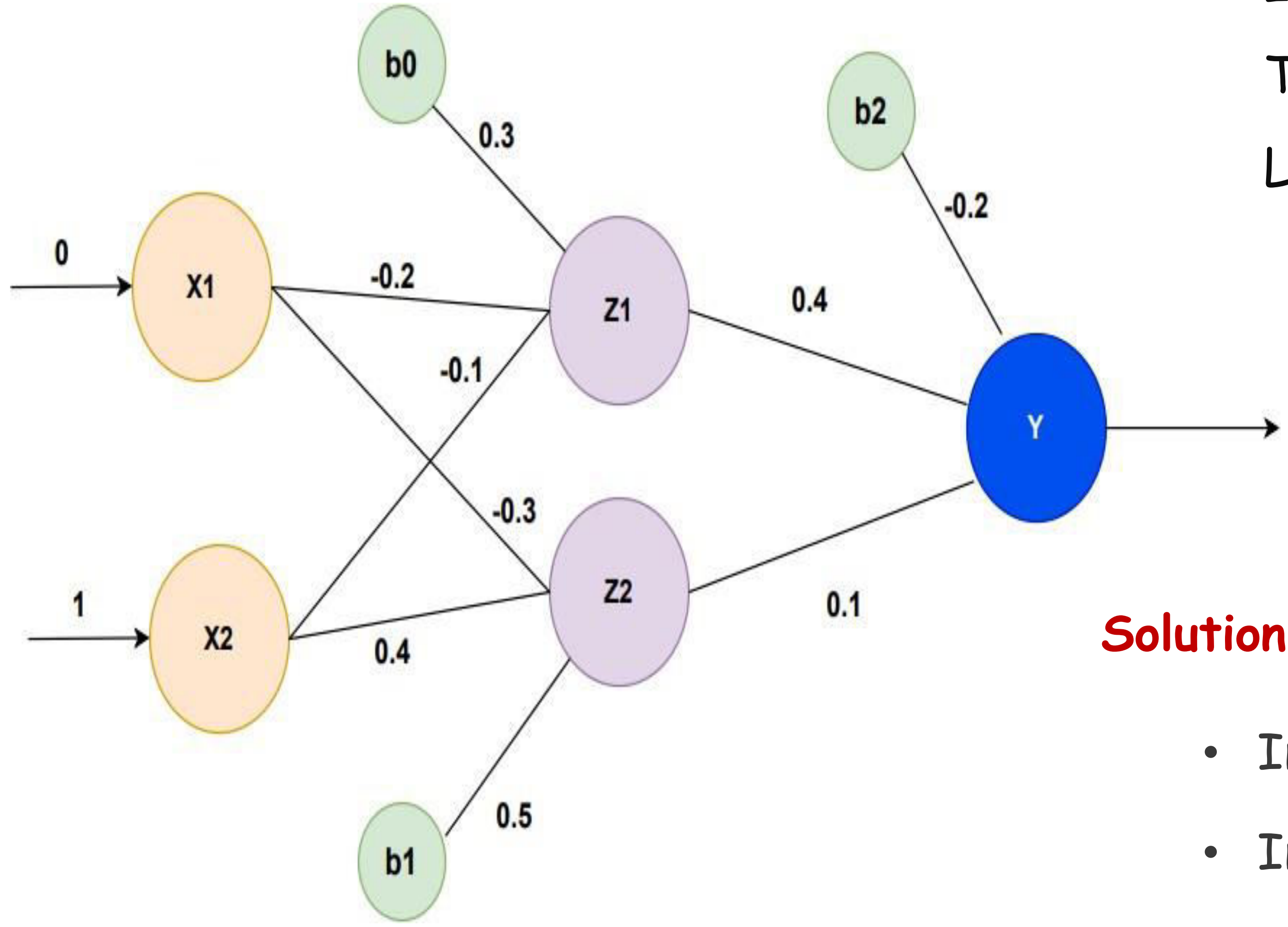
Calculation of Error at Each Node

Unit $_j$	Error $_j$
6	**0.475(1-0.475)(1-0.475) =0.1311** **We assume T_6 = 1**
5	**0.525 x (1- 0.525)x 0.1311x (-0.2) = 0.0065**
4	**0.332 x (1-0.332) x 0.1311 x (-0.3) = -0.0087**

Calculation of weights and Bias Updating

Learning Rate l =0.9

Weight	New Values
w46	-0.3 + 0.9(0.1311)(0.332) = -0.261
w56	-0.2 + (0.9)(0.1311)(0.525) = -0.138
w14	0.2 + 0.9(-0.0087)(1) = 0.192
w15	-0.3 + (0.9)(-0.0065)(1) = -0.306
……..similarly	………similarly
θ6	0.1 +(0.9)(0.1311)=0.218



Input vector = [0,1]

Target output = 1

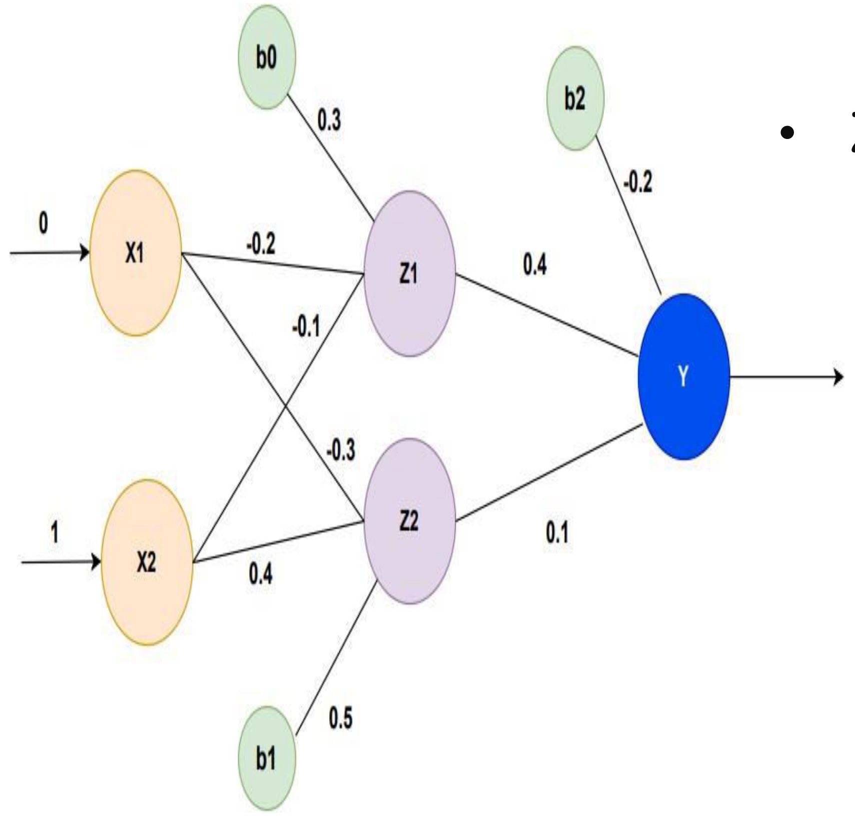
Learning Rate = 0.25

Activation function= binary sigmoidal

Solution:

- Input vector to Z_1 [0.6, -0.1, 0.3]
- Input vector to Z_2 [-0.3, 0.3, 0.5]
- Input vector to Y: [0.4, 0.1, -0.2]

Example – II



- Step 1: Calculate the net input weight for Z1

$Z_{in1} = v_{01} + x_1 * v_{11} + x_2 * v_{21}$

$= 0.3 + 0 * 0.6 + 1 * 0.1 = 0.2$

- $Z_{in2} = v_{02} + x1 * v_{21} + x2 * v_{22}$

$= 0.5 + 0 * 0.6 + 1 * (0.4) = 0.9$

Step 2: Apply the Activation Function

$z_i = f(Z_{in1}) = 1/1+e^{-z}{}_{in1}$

$= 1/1+e^{-0.2}$

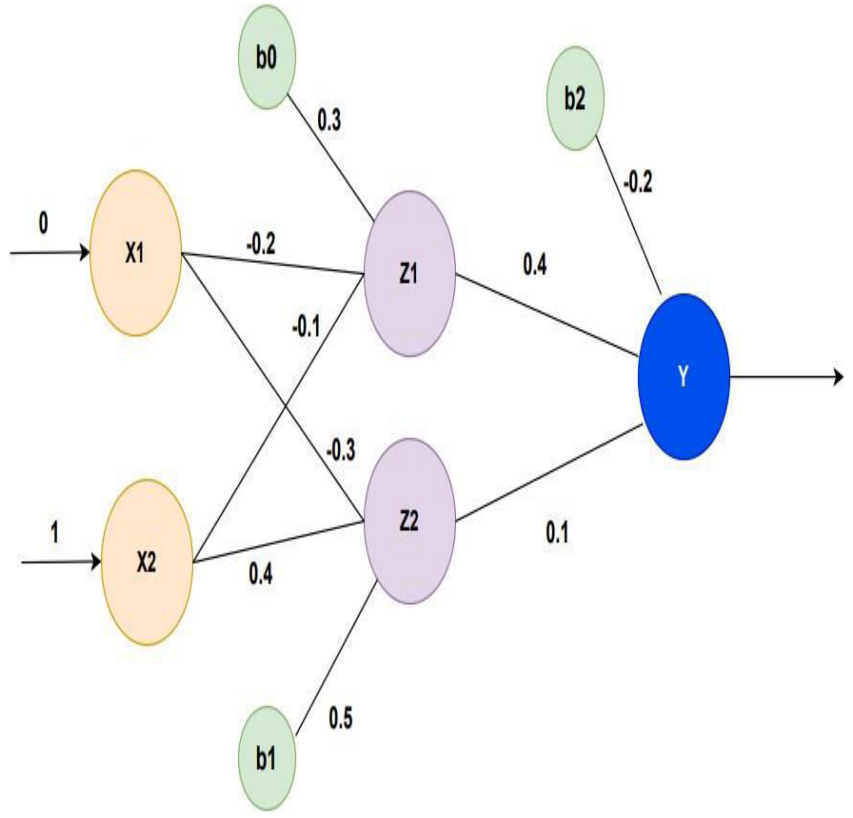
$= 0.5498$

$z_j = f(Z_{in2}) = 1/1+e^{-z}{}_{in2}$

$= 1/1+e^{-0.9}$

$= 0.7109$

Example – II



Step 3: Calculate the Net input of Output layer

$y_{in} = w_0 + z_i{*}w_1 + z_j{*}w_2$

$= -0.2 + 0.5498 * 0.4 + 0.7109 * 0.1$

$= 0.09101$

Step 4: Calculate the Net output using activation

$y = f(y_{in}) = 1/1+e^{-y_{in}}$

$= 1/1+e^{-0.09101}$

$= 0.5227$

Step 5: Calculation of Error in Y

$$\boxed{Err_k = O_k(1 - O_k)(T_k - O_k)}$$

$E_{rrk} = 0.5227(1-0.5227)(1-0.5227)$

$= 0.119079$

Example – II

Step 6: Weight Updation

$$w_{ij} = w_{ij} + (l) Err_j O_i$$

Learning Rate = 0.25

ΔW_{31}= 0.25*0.119079*0.5498

= 0.0164

new W_{31} = W_{31} (old) + ΔW_{31}

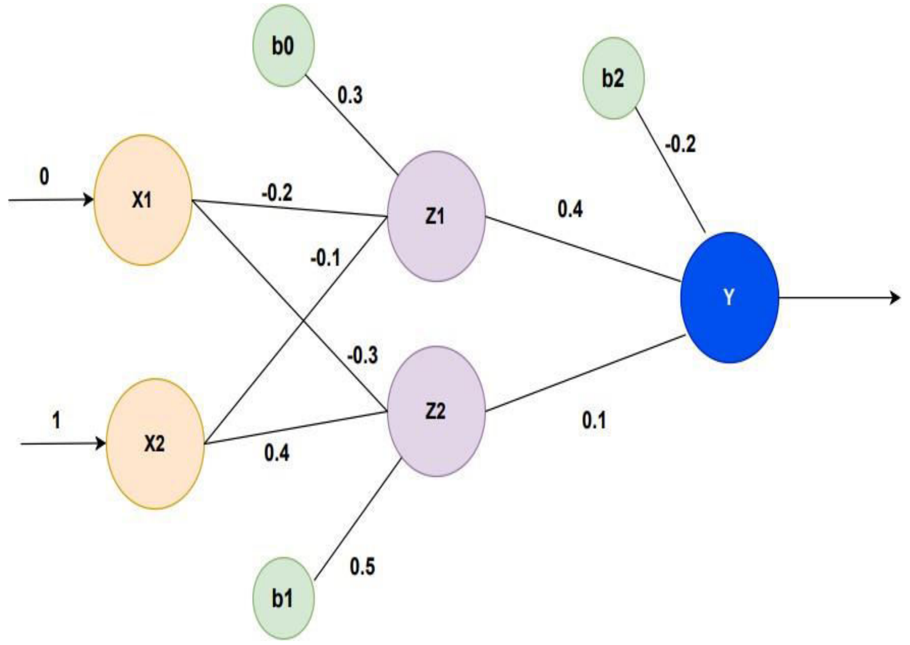
= 0.4 + 0.0164 = 0.4164

ΔW_{41}= 0.25*0.119079*0.7109

= 0.0211

new W_{32} = W_{31} (old) + ΔW_{31}

= 0.1 + 0.0211 = 0.1211



Step 7: Calculation of Error in Z_2 and Z_1

$$Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}$$

Err Z_1 = 0.5498*(1-0.5498)*0.1190*0.4

= 0.011782

Err Z_2 = 0.7109*(1-0.7109)*0.1190*0.1

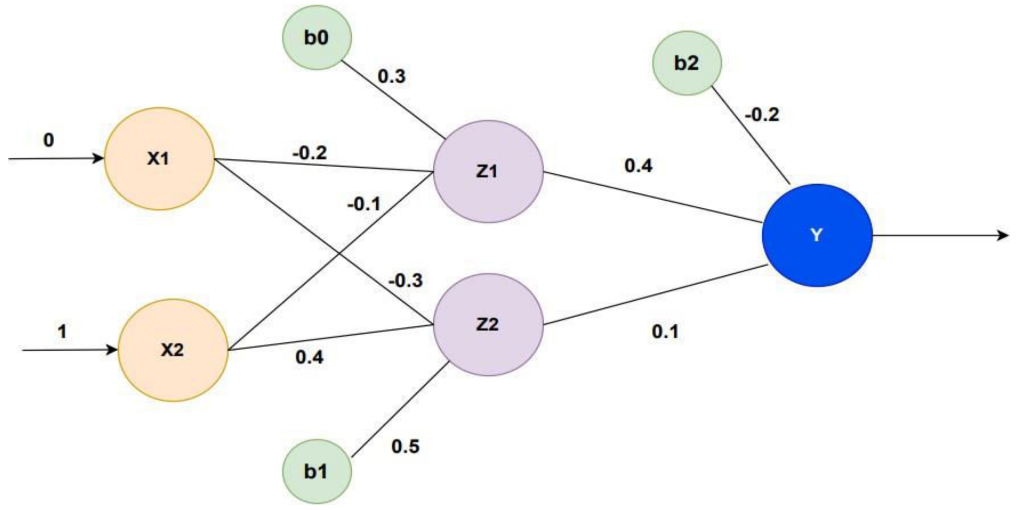
= 0.002446

Example – II

Step 8: Weight Updation

$$w_{ij} = w_{ij} + (l) Err_j O_i$$

Learning Rate = 0.25



ΔW_{11}= 0.25*0.0.1178* 0.2

= 0.000589

new W_{11} = W_{11} (old) + ΔW_{11}

= -0.2 + 0.000589 = -0.19941

ΔW_{21}= 0.25*0.1178*0.2

= 0.00589

new W_{21} = W_{21} (old) + ΔW_{21}

= -0.1 + 0.00589 = -0.09411

Similarly

ΔW_{12}= 0.25* 0.002446 * 0.9

= 0.00055

new W_{12} = W_{12} (old) + ΔW_{12}

= -0.3+ 0.00055 = -0.29945

ΔW_{22}= 0.25* 0.002446 * 0.9

= 0.00055

new W_{22} = W_{22} (old) + ΔW_{22}

= 0.4+ 0.00055 = 0.40055

Gradient Descent

A common cost function is a squared error

$$\varepsilon(i) = \frac{1}{2}\sum_{m=1}^{k_L}\left(e_m(i)\right)^2 = \frac{1}{2}\sum_{m=1}^{k_L}\left(\hat{y}_m(i) - y_m(i)\right)^2$$

- Gradient descent starts with an initial guess at the weights over all layers of the network.
- We then use these weights to compute the network output for each input vector x(i) in the training data.
- This allows us to calculate the error ε(i) for each of these inputs.
- Then, in order to minimize this error, we incrementally update the weights in the negative gradient direction:

$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) - \mu\frac{\partial J}{\partial \mathbf{w}_j^r} = \mathbf{w}_j^r(\text{old}) - \mu\sum_{i=1}^{N}\frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r}$$

Gradient Descent

Since,

$$v_j^r(i) = \left(\mathbf{w}_j^r\right)^t \mathbf{y}^{r-1}(i)$$

- The influence of the jth weight of the rth layer on the error can be expressed as

$$\frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r} = \frac{\partial \varepsilon(i)}{\partial v_j^r(i)} \frac{\partial v_j^r(i)}{\partial \mathbf{w}_j^r}$$
$$= \delta_j^r(i)\mathbf{y}^{r-1}(i)$$

where

$$\delta_j^r(i) \triangleq \frac{\partial \varepsilon(i)}{\partial v_j^r(i)}$$

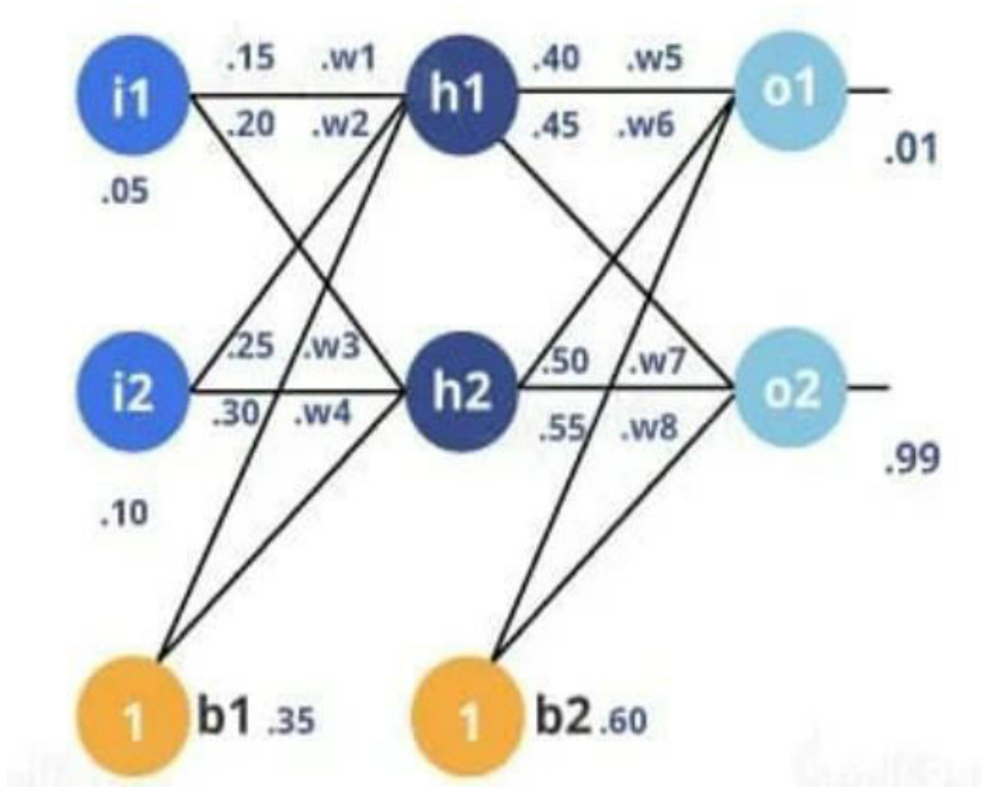
$$\frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r} = \delta_j^r(i)\mathbf{y}^{r-1}(i),$$

where

$$\delta_j^r(i) \triangleq \frac{\partial \varepsilon(i)}{\partial v_j^r(i)}$$

- For an intermediate layer r, we cannot compute $\delta_j^r(i)$ directly. However, $\delta_j^r(i)$ can be computed inductively, starting from the output layer.

Example



Inputs(i1): 0.05 Output (o1): 0.01

Inputs(i2): 0.10 Output(o2):0.99

net h1 = $w_1 * i_1 + w_2 * i_2 + b_1 * 1$ .... (Equation 1)

net h1 = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775

Out h2= $1/1+e^{-0.3775}$

=0.59327

Similarly,
out h2 = 0.596884378

net o1 = w_5 * out h1 + w_6 * out h2 + b_2 * 1 .... (Equation 3)

net o1 = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967

out o1 = $1/(1 + e^{-\text{net o1}}) = 1/(1 + e^{-1.105905967}) = 0.75136507$ .... (Equation 4)

Similarly, out o2 = 0.772928465

Calculating the Total Error:

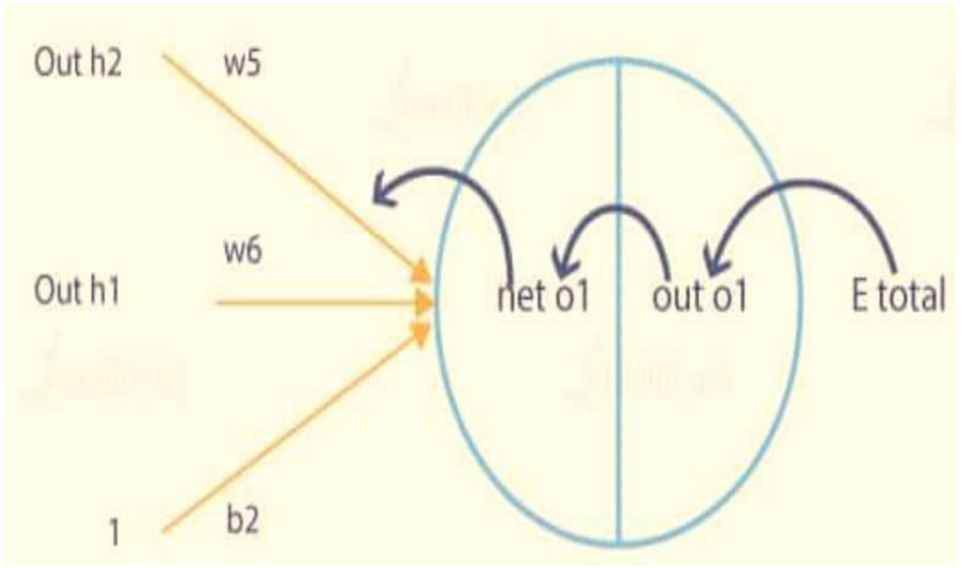
E o1 = 1/2(target o1 - out o1)2 = 1/2(0.01 - 0.75136507)2 = 0.27481108

E o2 = 0.023560026

E total = E o1 + E o2 = 0.274811083 + 0.023560026 = 0.298371109

Example

Backward Propagation:



- Putting all values together and calculating the updated weight value:

$$(\partial E\ total)/\partial w_5 = (\partial E\ total)/(\partial out\ o1) * (\partial out\ o1)/(\partial net\ o1) * (\partial net\ o1)/\ \partial w_5 = 0.082167041$$

- Calculate the updated value of w5:

$$W_5^+ = W_5 - n*(\partial E\ total)/\partial w_5 = 0.4 - 0.5 * 0.082167041 = 0.35891648\ \textit{(This is gradient descent)}$$

- We can repeat this process to get the new weights w6, w7, and w8.

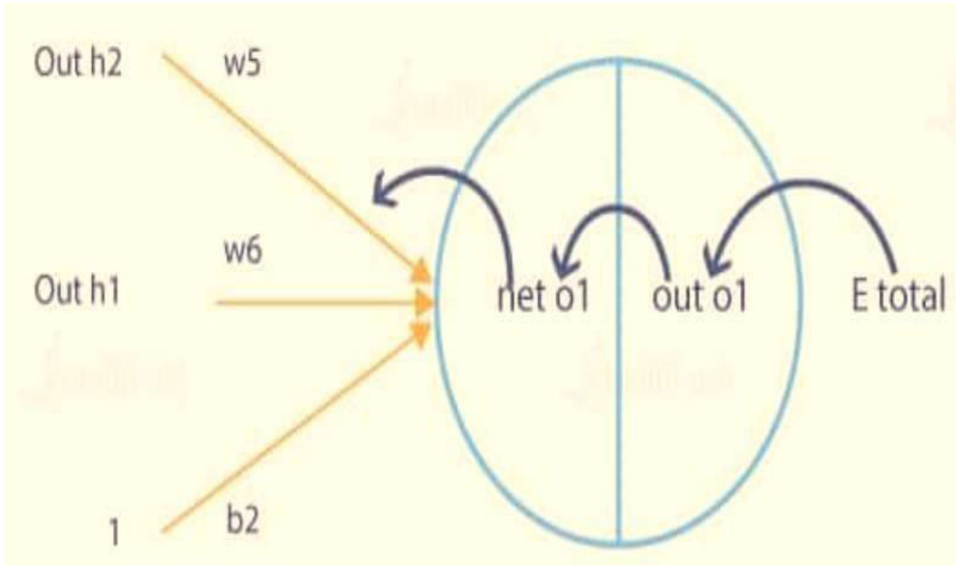
$$W_6^+ = 0.408666186$$

$$W_7^+ = 0.511301270$$

$$W_8^+ = 0.561370121$$

Example

Backward Propagation:



- Change in the weight w5:

$$(\partial E\ total)/\partial w_5 = (\partial E\ total)/(\partial out\ o1) * (\partial out\ o1)/(\partial net\ o1) * (\partial net\ o1)/\ \partial w_5$$

- The first thing we need to calculate the change in total errors w.r.t the outputs o1 and o2:

$$E\ total = (1/2)*(target\ o1 - out\ o1)^2 + (1/2)*(target\ o2 - out\ o2)^2$$
$$(\partial E\ total)/(\partial out\ o1) = -(target\ o1 - out\ o1) = -(0.01 - 0.75136507) = 0.74136507$$

- Backward and calculate the change in the output o1 w.r.t to its total net input:

$out\ o1 = 1/(1 + e^{-net\ o1})$ (from Equation 4)

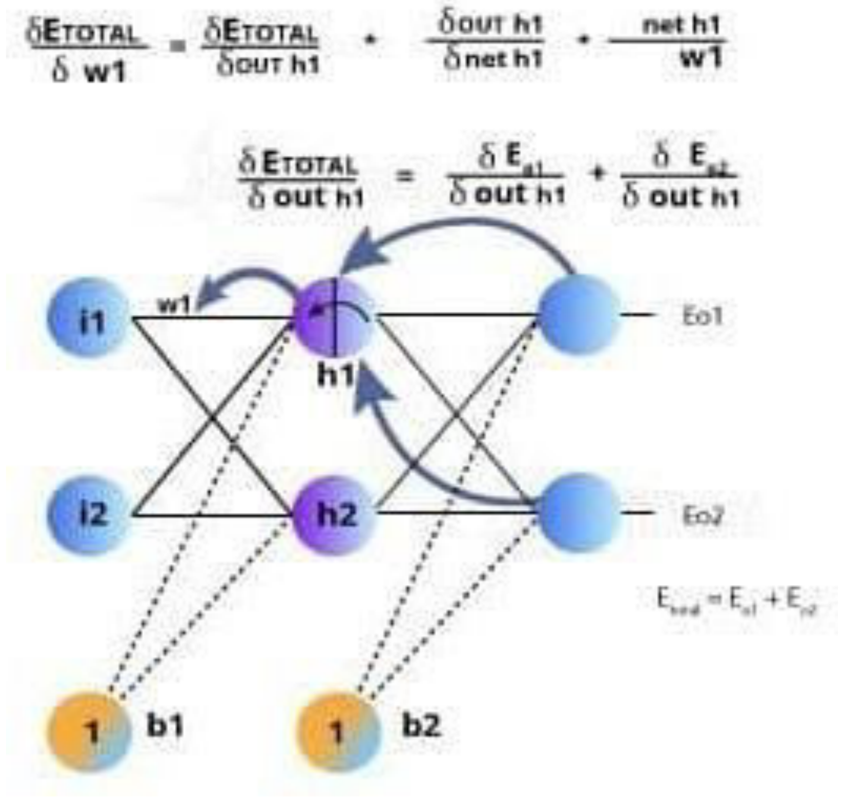
$$(\partial out\ o1)/(\partial net\ o1) = out\ o1(1-out\ o1) = 0.75136507(1 - 0.75136507) = 0.186815602$$

$net\ o1 = w_5*out\ h1 + w_6*out\ h2 + b_2*1$ (from Equation 3)

$$(\partial net\ o1)/\ \partial w_5 = 1*out\ h1*w_5^{(1-1)} + 0 + 0 = out\ h1 = 0.593269992$$

Example

Backward Propagation:



δEtotal/δw1 = δEtotal/δout h1 * δout h1/δnet h1 * net h1/w1
δEtotal/δout h1 = δE_o1/δout h1 + δE_o2/δout h1
i1
i2
w1
h1
h2
Eo1
Eo2
E_total = E_o1 + E_o2
1
b1
1
b2

- Starting with h1:

(∂E total)/(∂out h1) = (∂E o1)/(∂out h1) + (∂E o2)/(∂out h1)

(∂E o1)/(∂out h1) = (∂E o1)/(∂net o1) * (∂net o1)/(∂out h1)

We can calculate (∂E o1)/(∂net o1) using the values calculated earlier.

(∂E o1)/(∂net o1) = (∂E o1)/(∂out h1) * (∂out h1)/(∂net o1)

= 0.74136507 * 0.186815602 = 0.138498562

net o1 = w_5*out h1 + w_6*out h2 + b_2*1

(∂net o1)/(∂out h1) = w_5 = 0.40

Let's put the values in the equation.

(∂E o1)/(∂out h1) = (∂E o1)/(∂net o1) * (∂net o1)/(∂out h1)

= 0.138498562 * 0.40 = 0.055399425

Following the same process for (∂E o2)/(∂out h1), we get:

(∂E o2)/(∂out h1) = -0.019049119

(∂E total)/(∂out h1) = (∂E o1)/(∂out h1) + (∂E o2)/(∂out h1)

= 0.055399425 + (-0.019049119) = 0.036350306

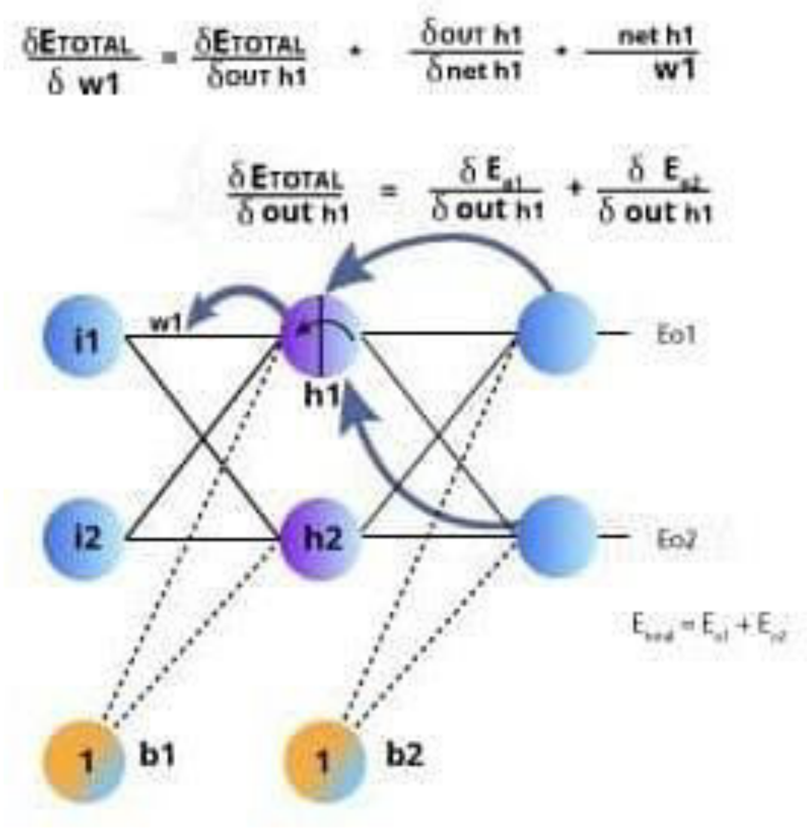
Now that we have (∂E total)/(∂out h1), we need to figure out (∂out h1)/(∂net h1)and (∂net h1)/∂w *for each weight*

out h1 = 1/(1 + $e^{-net\ h1}$)

(∂out h1)/(∂net h1) = out h1(1- out h1) = 0.59326999(1 - 0.59326999) = 0.241300709

Example

Backward Propagation:



δETOTAL / δ w1 = δETOTAL / δOUT h1 * δOUT h1 / δnet h1 * net h1 / w1
δETOTAL / δ out h1 = δ E_o1 / δ out h1 + δ E_o2 / δ out h1
i1
w1
h1
i2
h2
Eo1
Eo2
E_total = E_o1 + E_o2
1
b1
1
b2

- Calculate the partial derivative of the total net input of h1 w.r.t w1.

The same way as we did for the output neuron.:

$$net\ h1 = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$(\partial net\ h1)/\partial w1 = i_1 = 0.05$$

$(\partial E\ total)/\partial w_1 = (\partial E\ total)/(\partial out\ h1) * (\partial out\ h1)/(\partial net\ h1) * (\partial net\ h1)/\ \partial w_1$

$(\partial E\ total)/\partial w_1 = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$

We can now update w1.

$w_1^+ = w_1 - n * (\partial E\ total)/\partial w_1 = 0.15 - 0.5 * 0.000438568 = 0.149780716$

Let's update other weights similarly.

$w_2^+ = 0.19956143$

$w_3^+ = 0.24975114$

$w_4^+ = 0.29950229$

Sequential (online) vs. Batch Training

Sequential mode:

– Update rule applied after each input-target presentation.

– Order of presentation should be randomized.

– Benefits: less storage, stochastic search through weight space helps avoid local minima.

– Disadvantages: hard to establish theoretical convergence conditions.

Batch mode:

– Update rule applied after all input-target pairs are seen.

– Benefits: an accurate estimate of the gradient, convergence to local minimum is guaranteed under simpler conditions.