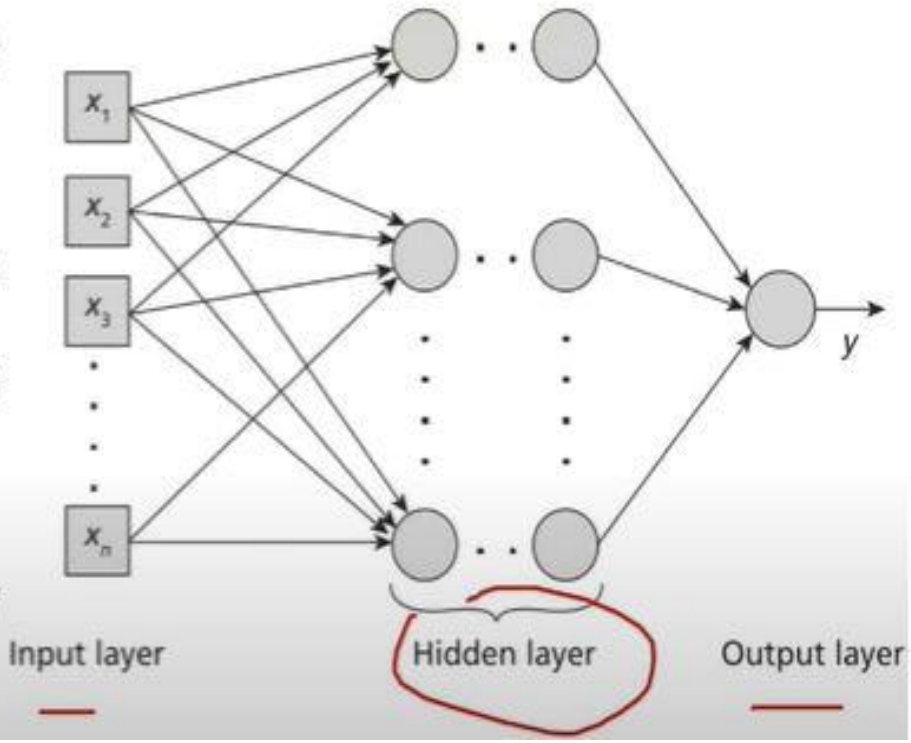
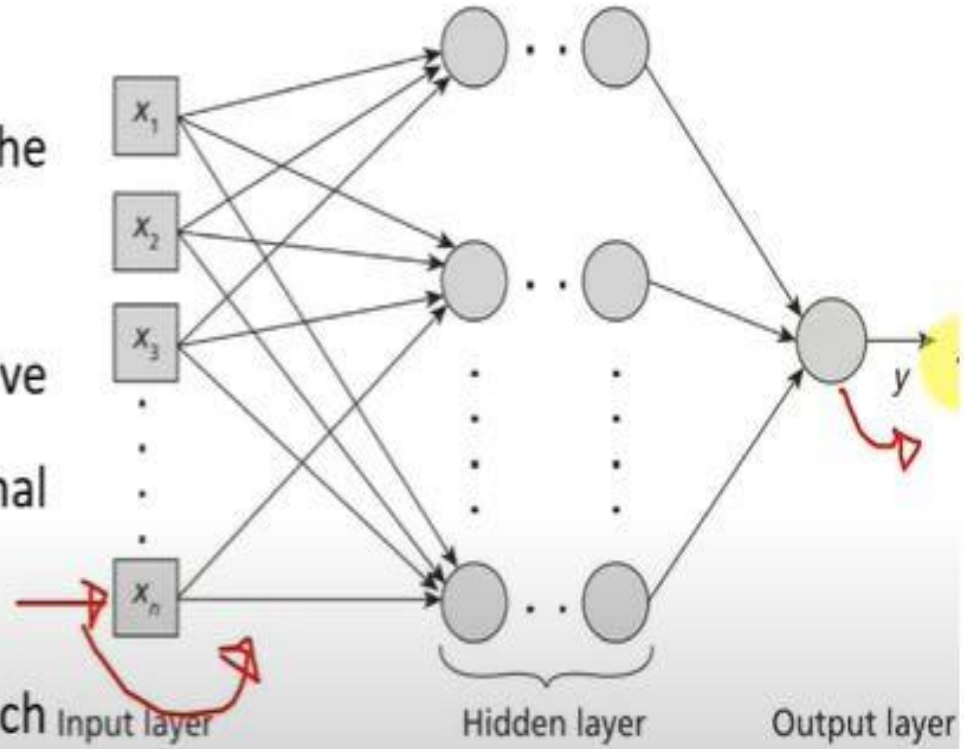
DEEP LEARNING

Multi-Layer Perceptron Learning Algorithm

- A multi-layer perceptron is a type of Feed Forward Neural Network with multiple neurons arranged in layers.
- The network has at least three layers with an input layer, one or more hidden layers and an output layer.
- All the neurons in a layer are fully connected to the neurons in the next layer.



- The input layer is the visible layer.
- It just passes the input to the next layer.
- The layers following the input layer are the hidden layers.
- The hidden layers neither directly receive inputs nor send outputs to the external environment.
- The final layer is the output layer which outputs a single value or a vector of values.



- The activation functions used in the layers can be linear or non-linear depending on the type of the problem modelled.
- Typically, a sigmoid activation function is used if the problem is a binary classification problem and a softmax activation function is used in a multi-class classification problem.

MULTILAYER PERCEPTRON LEARNING ALGORITHM

Input: Input vector $(x_1, x_2, ..., x_n)$

Output: Y_n

Learning rate: α

Assign random weights and biases for every connection in the network in the range [–0.5, +0.5].

Step 1: Forward Propagation

1. **Calculate Input and Output in the *Input Layer*:**

(Input layer is a direct transfer function, where the output of the node equals the input).

Input at Node j 'I$_j$' in the Input Layer is

$$I_j = x_j$$

Net Input at Node j in the ***Output Layer*** *is*

$$I_j = \sum_{i=1}^{n} O_i w_{ij} + x_0 \times \theta_j$$

where,

O_i is the output from Node i

w_{ij} is the weight in the link from Node i to Node j

x_0 is the input to bias node '0' which is always assumed as 1

θ_j is the weight in the link from the bias node '0' to Node j

Output at Node j

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

where,

I_j is the input received at Node j

3. Estimate error at the node in the *Output Layer*:

$$\text{Error} = O_{Desired} - O_{Estimated}$$

where,

$O_{Desired}$ is the desired output value of the Node in the Output Layer

$O_{Estimated}$ is the estimated output value of the Node in the Output Layer

Step 2: Backward Propagation

1. Calculate Error at each node:

For each Unit k in the ***Output Layer***

$$\text{Error}_k = O_k(1 - O_k)(O_{Desired} - O_k)$$

where,

O_k is the output value at Node k in the Output Layer.

$O_{Desired}$ is the desired output value of the Node in the Output Layer.

For each unit j in the ***Hidden Layer***

$$\text{Error}_j = O_j(1 - O_j) \sum_k \text{Error}_k \, w_{jk}$$

where,

O_j is the output value at Node j in the Hidden Layer.

Error_k is the error at Node k in the Output Layer.

w_{jk} is the weight in the link from Node j to Node k.

2. Update all weights and biases:

Update weights

$$\Delta w_{ij} = \alpha \times \text{Error}_j \times O_i$$
$$w_{ij} = w_{ij} + \Delta w_{ij}$$

where,

O_i is the output value at Node i.

Error_j is the error at Node j.

α is the learning rate.

w_{ij} is the weight in the link from Node i to Node j.

Δw_{ij} is the difference in weight that has to be added to w_{ij}.

Update Biases

$$\Delta\theta_j = \propto \times \text{Error}_j$$
$$\theta_j = \theta_j + \Delta\theta_j$$

where,

Error_j is the error at Node j.

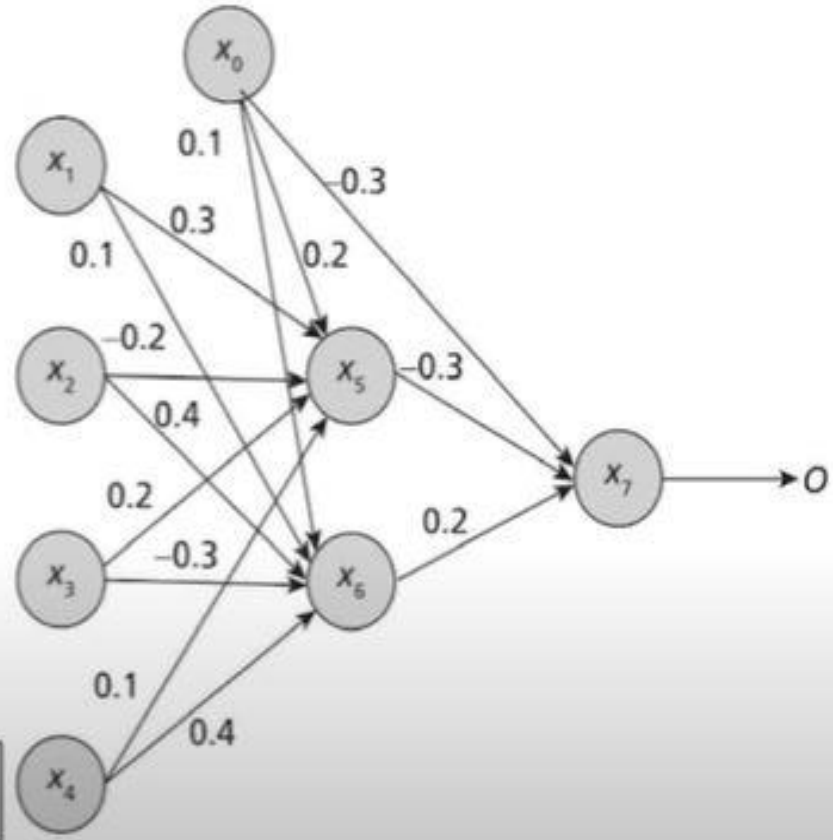
$\propto$ is the learning rate.

θ_j is the bias value from Bias Node 0 to Node j.

$\Delta\theta_j$ is the difference in bias that has to be added to θ_j.

MULTILAYER PERCEPTRON EXAMPLE

- The given MLP consists of an Input layer, one Hidden layer and an Output layer.
- The input layer has 4 neurons, the hidden layer has 2 neurons and the output layer has a single neuron.
- Train the MLP by updating the weights and biases in the network.
- Learning Rate = 0.8



x_1	x_2	x_3	x_4	$O_{Desired}$
1	1	0	1	1

x_1	x_2	x_3	x_4	w_{15}	w_{16}	w_{25}	w_{26}	w_{35}	w_{36}	w_{45}	w_{46}	w_{57}	w_{67}	θ_5	θ_6	θ_7
1	1	0	1	0.3	0.1	–0.2	0.4	0.2	–0.3	0.1	0.4	–0.3	0.2	0.2	0.1	–0.3

2. Calculate Net Input and Output in the Hidden Layer and Output Layer as

Unit $_j$	Net Input I_j	Output O_j
x_5	$I_5 = x_1 \times w_{15} + x_2 \times w_{25} + x_3 \times w_{35} + x_4 \times w_{45} + x_0 \times \theta_5$ $I_5 = 1 \times 0.3 + 1 \times -0.2 + 0 \times 0.2 + 1 \times 0.1 + 1 \times 0.2 = 0.4$	$O_5 = \frac{1}{1+e^{-I_5}} \quad \frac{1}{1+e^{-0.4}} = 0.599$
x_6	$I_6 = x_1 \times w_{15} + x_2 \times w_{26} + x_3 \times w_{36} + x_4 \times w_{46} + x_0 \times \theta_6$ $I_6 = 1 \times 0.3 + 1 \times 0.4 + 0 \times -0.3 + 1 \times 0.4 + 1 \times 0.1 = 1.2$	$O_6 = \frac{1}{1+e^{-I_6}} \quad \frac{1}{1+e^{-1.2}} = 0.769$
x_7	$I_7 = O_5 \times w_{57} + O_6 \times w_{67} + x_0 \times \theta_7$ $I_7 = 0.599 \times -0.3 + 0.769 \times 0.2 + 1 \times -0.3 = -0.326$	$O_7 = \frac{1}{1+e^{-I_7}} \quad \frac{1}{1+e^{0.326}} = 0.419$

3. Calculate Error $= O_{desired} - O_{Estimated}$

So, error for this network is:

$\text{Error} = O_{desired} - O_7 = 1 - 0.419 = 0.581$

So, we need to back propagate to reduce the error.

Step 2: Backward Propagation

1. Calculate Error at each node

   For each unit k in the output layer, calculate:

   $$\text{Error}_k = O_k(1 - O_k)(O_{desired} - O_k)$$

   For each unit j in the hidden layer, calculate:

   $$\text{Error}_j = O_j(1 - O_j)\sum_k \text{Error}_k w_{jk}$$

Error Calculation for Each Unit in the Output Layer and Hidden Layer

For Output Layer Unit_k	Error_k
x_7	$\text{Error}_7 = O_7(1-O_7)(Y_n - O_7)$ $= 0.419 \times (1-0.419) \times (1-0.419) = 0.141$
For Hidden Layer Unit_j	**Error_j**
x_6	$\text{Error}_6 = O_6(1-O_6)\sum_k \text{Error}_k\, w_{jk} = O_6(1-O_6)\,\text{Error}_7\, w_{67}$ $= 0.769\,(1-0.769) \times 0.2 \times 0.141 = 0.005$
x_5	$\text{Error}_5 = O_5(1-O_5)\sum_k \text{Error}_k\, w_{jk} = O_5(1-O_5)\,\text{Error}_7\, w_{57}$ $= 0.599(1-0.599) \times 0.141 \times -0.3 = -0.0101$

2. Update weight using the below formula:

Learning rate $\alpha = 0.8$.

$w_{ij} = w_{ij} + \propto \times \text{Error}_j \times O_i$

The updated weights and bias

w_{ij}	$w_{ij} = w_{ij} + \propto \times \text{Error}_j \times O_i$	New Weight
w_{15}	$w_{15} = w_{15} + 0.8 \times \text{Error}_5 \times O_1$ $= 0.3 + 0.8 \times -0.0101 \times 1$	0.292
w_{16}	$w_{16} = w_{16} + 0.8 \times \text{Error}_6 \times O_1$ $= 0.1 + 0.8 \times 0.005 \times 1$	0.104
w_{25}	$w_{25} = w_{25} + 0.8 \times \text{Error}_5 \times O_2$ $= -0.2 + 0.8 \times -0.0101 \times 1$	–0.208
w_{26}	$w_{26} = w_{26} + 0.8 \times \text{Error}_6 \times O_2$ $= 0.4 + 0.8 \times 0.005 \times 1$	0.404

2. Update weight using the below formula:

Learning rate $\alpha = 0.8$.

$w_{ij} = w_{ij} + \propto \times \text{Error}_j \times O_i$

The updated weights and bias

w_{ij}	$w_{ij} = w_{ij} + \propto \times \text{Error}_j \times O_i$	New Weigl
w_{35}	$w_{35} = w_{35} + 0.8 \times \text{Error}_5 \times O_3$ $= 0.2 + 0.8 \times -0.0101 \times 0$	0.2
w_{36}	$w_{36} = w_{36} + 0.8 \times \text{Error}_6 \times O_3$ $= -0.3 + 0.8 \times 0.005 \times 0$	–0.3
w_{45}	$w_{45} = w_{45} + 0.8 \times \text{Error}_5 \times O_4$ $= 0.1 + 0.8 \times -0.0101 \times 1$	0.092
w_{46}	$w_{46} = w_{46} + 0.8 \times \text{Error}_6 \times O_4$ $= 0.4 + 0.8 \times 0.005 \times 1$	0.404
w_{57}	$w_{57} = w_{57} + 0.8 \times \text{Error}_7 \times O_5$ $= -0.3 + 0.8 \times 0.141 \times 0.599$	–0.232
w_{67}	$w_{67} = w_{67} + 0.8 \times \text{Error}_7 \times O_6$ $= 0.2 + 0.8 \times 0.141 \times 0.769$	0.287

Update bias using the below formula:

$$\theta_j = \theta_j + \alpha \times \text{Error}_j$$

θ_j	$\theta_j = \theta_j + \alpha \times \text{Error}_j$	New Bias
θ_5	$\theta_5 = \theta_5 + \alpha \times \text{Error}_5$ $= 0.2 + 0.8 \times -0.0101$	0.192
θ_6	$\theta_6 = \theta_6 + \alpha \times \text{Error}_6$ $= 0.1 + 0.8 \times 0.005$	0.104
θ_7	$\theta_7 = \theta_7 + \alpha \times \text{Error}_7$ $= -0.3 + 0.8 \times 0.141$	-0.187

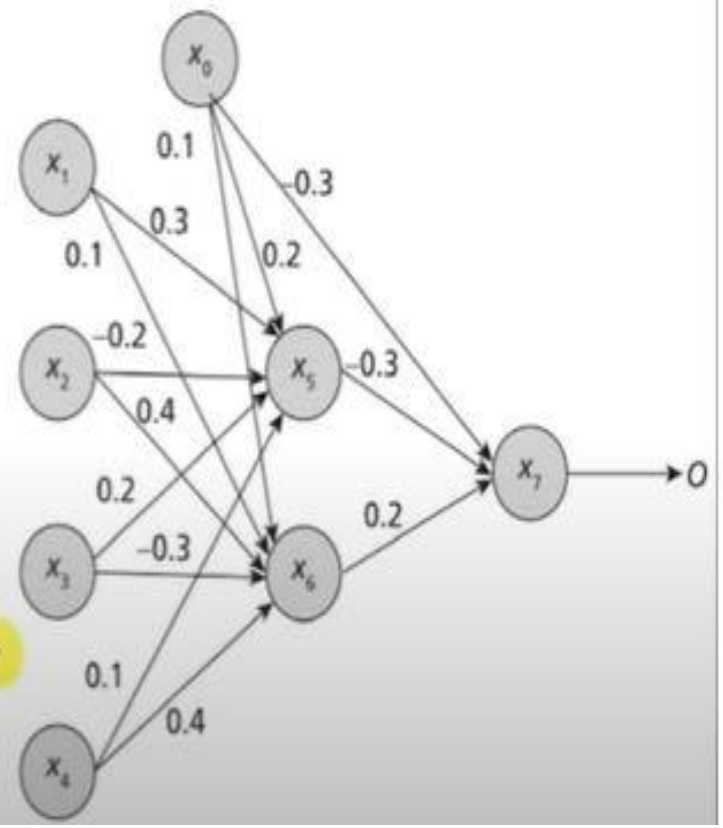
Iteration 2

Now, with the updated weights and biases:

1. Calculate Input and Output in the Input Layer

Net Input and Output Calculation

Input Layer	I_j	O_j
x_1	1	1
x_2	1	1
x_3	0	0
x_4	1	1



2. Calculate Net Input and Output in the Hidden Layer and Output Layer

Net Input and Output Calculation in the Hidden Layer and Output Layer

Unit j	Net Input I_j	Output O_j
x_5	$I_5 = x_1 \times w_{15} + x_2 \times w_{25} + x_3 \times w_{35} + x_4 \times w_{45} + x_0 \times \theta_5$ $I_5 = 1 \times 0.292 + 1 \times -0.208 + 0 \times 0.2 + 1 \times 0.092 + 1 \times 0.192 = 0.368$	$O_5 = \frac{1}{1+e^{-I_5}} = \frac{1}{1+e^{-0.368}} = 0.591$
x_6	$I_6 = x_1 \times w_{15} + x_2 \times w_{26} + x_3 \times w_{36} + x_4 \times w_{46} + x_0 \times \theta_6$ $I_6 = 1 \times 0.292 + 1 \times 0.404 + 0 \times -0.3 + 1 \times 0.404 + 1 \times 0.104 = 1.204$	$O_6 = \frac{1}{1+e^{-I_6}} = \frac{1}{1+e^{-1.204}} = 0.7692$
x_7	$I_7 = O_5 \times w_{57} + O_6 \times w_{67} + x_0 \times \theta_7$ $I_7 = 0.591 \times -0.232 + 0.7692 \times 0.287 + 1 \times -0.187 = -0.326$	$O_7 = \frac{1}{1+e^{-I_7}} = \frac{1}{1+e^{0.1034}} = 0.474$

- The output we receive in the network at node 7 is 0.474

Error= 1 - 0.474 = 0.526

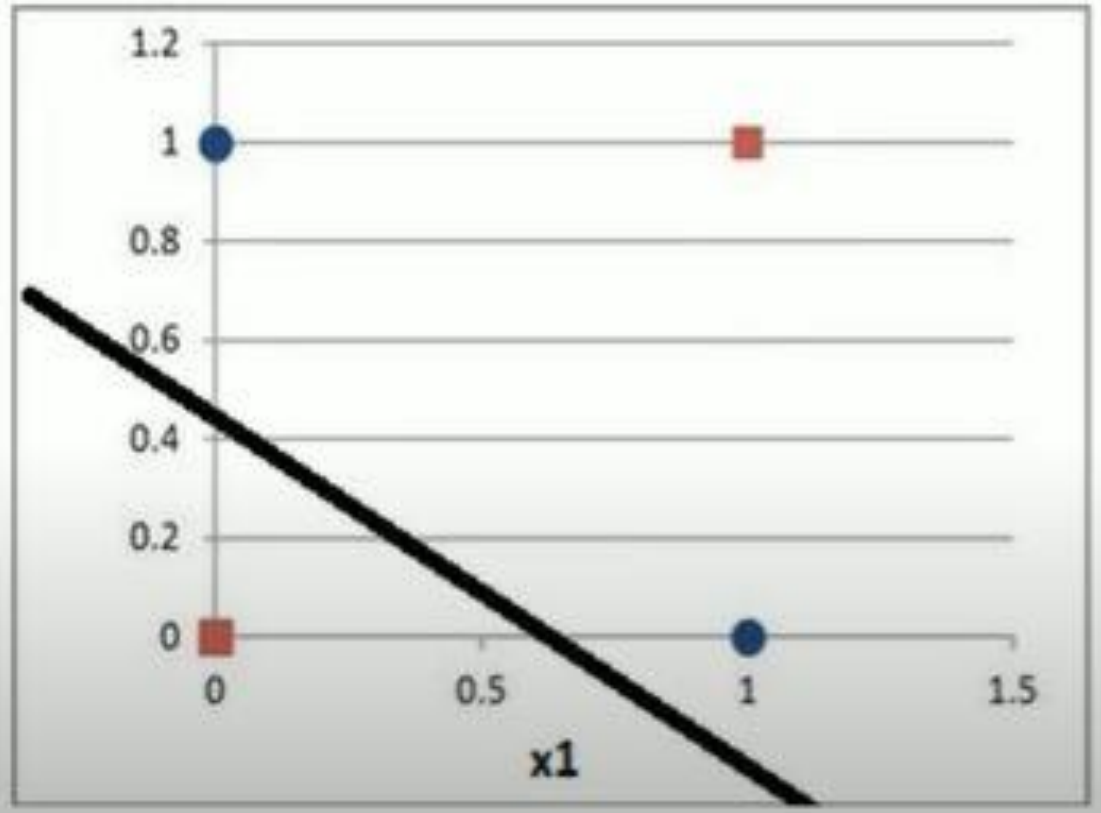
- Now, when we compare the error, we get in the previous iteration and in the current iteration,

Error reduced is 0.581 - 0.526 = 0.055

- It is visible that the network has learnt and reduced the error by 0.055.
- Thus, the training is continued for a predefined number of epochs or until the training error is reduced below a threshold value.

LEARNING XOR

- Consider the truth table for XOR function



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

$$y = x_1\overline{x_2} + \overline{x_1}x_2$$

$$y = x_1\overline{x_2} + \overline{x_1}x_2$$

$$y = z_1 + z_2$$

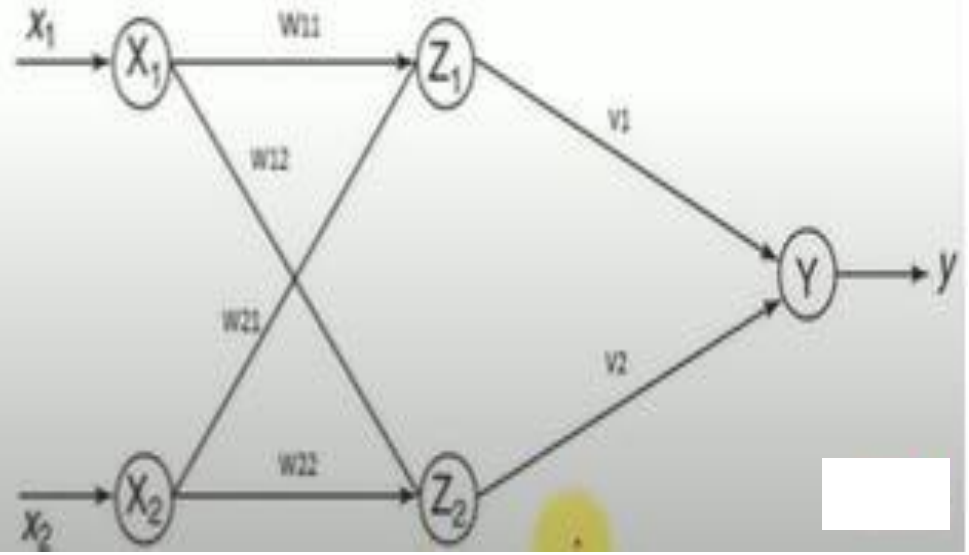
where

$z_1 = x_1\overline{x_2}$ (function 1)

$z_2 = \overline{x_1}x_2$ (function 2)

$y = z_1$ (OR) z_2 (function 3)

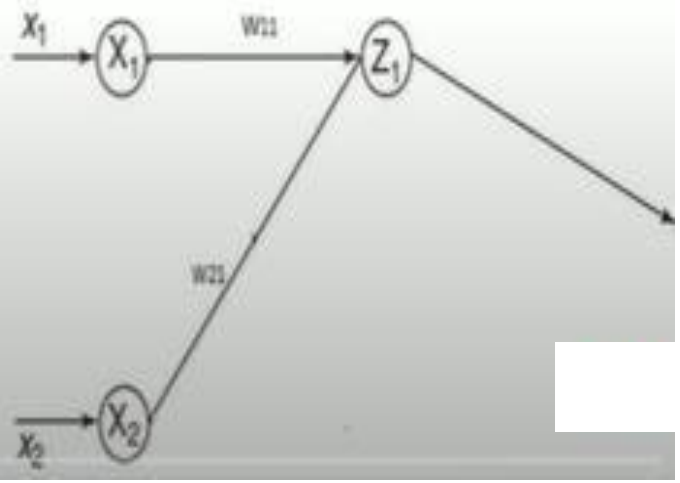
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



- First function $z_1 = x_1 \overline{x_2}$
- Assume the Initial weights are $w_{11} = w_{21} = 1$
- Threshold = 1 and learning rate = 1.5
- $(0,0) \rightarrow Z_{1in} = w_{ij} * x_i = 1 * 0 + 1 * 0 = 0 \ (out = 0)$
- $(0,1) \rightarrow Z_{1in} = w_{ij} * x_i = 1 * 0 + 1 * 1 = 1 \ (out = 1)$
  - $w_{ij} = w_{ij} + n(t - o)x_i$
  - $w_{11} = 1 + 1.5 * (0 - 1) * 0 = 0$
  - $w_{21} = 1 + 1.5 * (0 - 1) * 1 = -0.5$

X_1	X_2	Z_1
0	0	0
0	1	0
1	0	1
1	1	0

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq \theta \\ 0 & \text{if } y_{in} < \theta \end{cases}$$

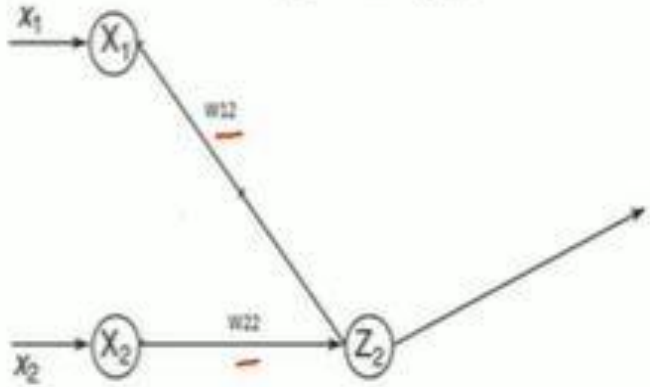


- First function $z_1 = x_1 \overline{x_2}$
- Assume the Initial weights are $w_{11} = 1 \; and \; w_{21} = -0.5$
- Threshold = 1 and learning rate = 1.5
- $(0,0) \rightarrow Z_{1in} = w_{ij} * x_i = 1 * 0 + (-0.5) * 0 = 0 \quad (out = 0)$
- $(0,1) \rightarrow Z_{1in} = w_{ij} * x_i = 1 * 0 + (-0.5) * 1 = -0.5 \; (out = 0)$
- $(1,0) \rightarrow Z_{1in} = w_{ij} * x_i = 1 * 1 + (-0.5) * 0 = 1 \quad (out = 1)$
- $(1,1) \rightarrow Z_{1in} = w_{ij} * x_i = 1 * 1 + (-0.5) * 1 = 0.5 \quad (out = 0)$

- First function $z_2 = \overline{x_1} x_2$
- Assume the Initial weights are $w_{12} = w_{22} = 1$
- Threshold = 1 and learning rate = 1.5
- $(0,0) \rightarrow Z_{2in} = w_{ij} * x_i = 1 * 0 + 1 * 0 = 0 \quad (out = 0)$
- $(0,1) \rightarrow Z_{2in} = w_{ij} * x_i = 1 * 0 + 1 * 1 = 1 \quad (out = 1)$
- $(1,0) \rightarrow Z_{2in} = w_{ij} * x_i = 1 * 1 + 1 * 0 = 1 \quad (out = 1)$
  - $w_{ij} = w_{ij} + n(t - o)x_i$
  - $w_{12} = 1 + 1.5 * (0 - 1) * 1 = -0.5$
  - $w_{22} = 1 + 1.5 * (0 - 1) * 0 = 1$

X_1	X_2	Z_2
0	0	0
0	1	1
1	0	0
1	1	0

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq \theta \\ 0 & \text{if } y_{in} < \theta \end{cases}$$

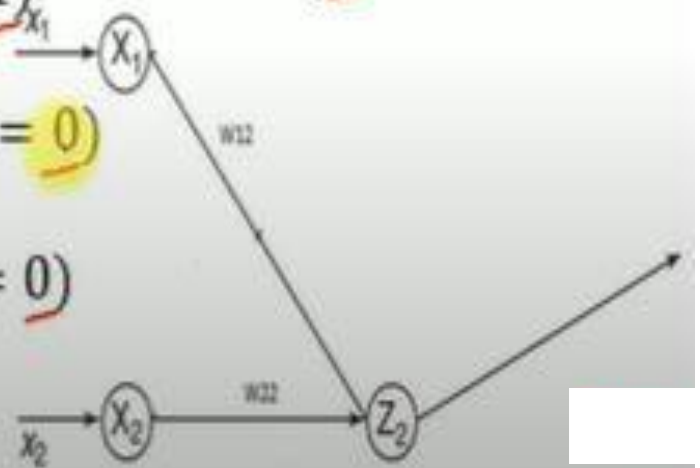


- First function $z_2 = \overline{x_1} x_2$
- Assume the Initial weights are $w_{12} = -0.5 \; and \; w_{22} = 1$
- Threshold = 1 and learning rate = 1.5

x_1	x_2	z_2
0	0	0
0	1	1
1	0	0
1	1	0

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq \theta \\ 0 & \text{if } y_{in} < \theta \end{cases}$$

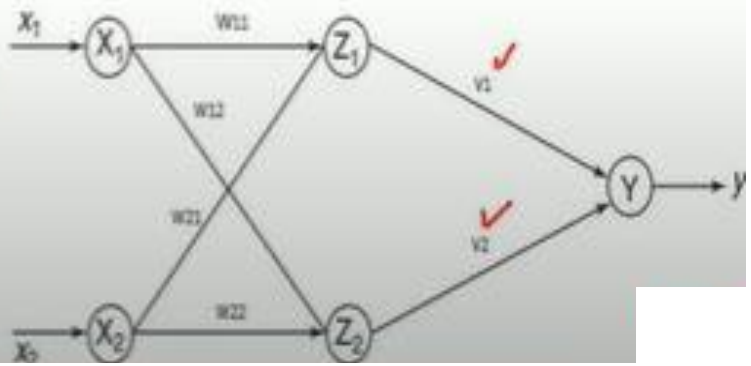
- $(0,0) \rightarrow Z_{2in} = w_{ij} * x_i = (-0.5) * 0 + 1 * 0 = 0 \quad (out = 0)$
- $(0,1) \rightarrow Z_{2in} = w_{ij} * x_i = (-0.5) * 0 + 1 * 1 = 1 \quad (out = 1)$
- $(1,0) \rightarrow Z_{2in} = w_{ij} * x_i = (-0.5) * 1 + 1 * 0 = -0.5 \quad (out = 0)$
- $(1,1) \rightarrow Z_{2in} = w_{ij} * x_i = (-0.5) * 1 + 1 * 1 = 0.5 \quad (out = 0)$



- First function $y = z_1 \text{ (OR) } z_2 \qquad y_{in} = z_1 v_1 + z_2 v_2$
- Assume the Initial weights are $v_1 = v_2 = 1$
- Threshold = 1 and learning rate = 1.5
- $(0,0) \rightarrow y_{in} = v_i * x_i = 1 * 0 + 1 * 0 = 0 \quad (out = 0)$
- $(0,1) \rightarrow y_{in} = v_i * x_i = 1 * 0 + 1 * 1 = 1 \quad (out = 1)$
- $(1,0) \rightarrow y_{in} = v_i * x_i = 1 * 1 + 1 * 0 = 1 \quad (out = 1)$
- $(0,0) \rightarrow y_{in} = v_i * x_i = 1 * 0 + 1 * 0 = 0 \quad (out = 0)$

x_1	x_2	z_1	z_2	y
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

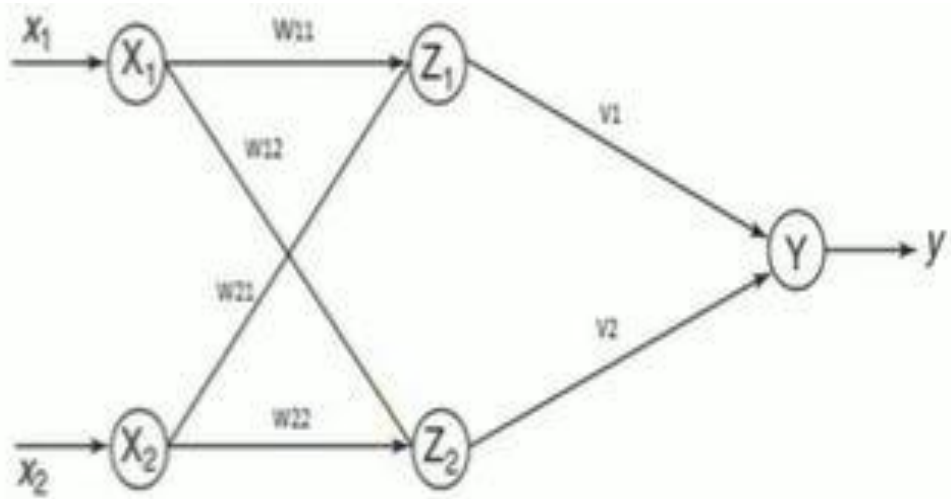
$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq \theta \\ 0 & \text{if } y_{in} < \theta \end{cases}$$

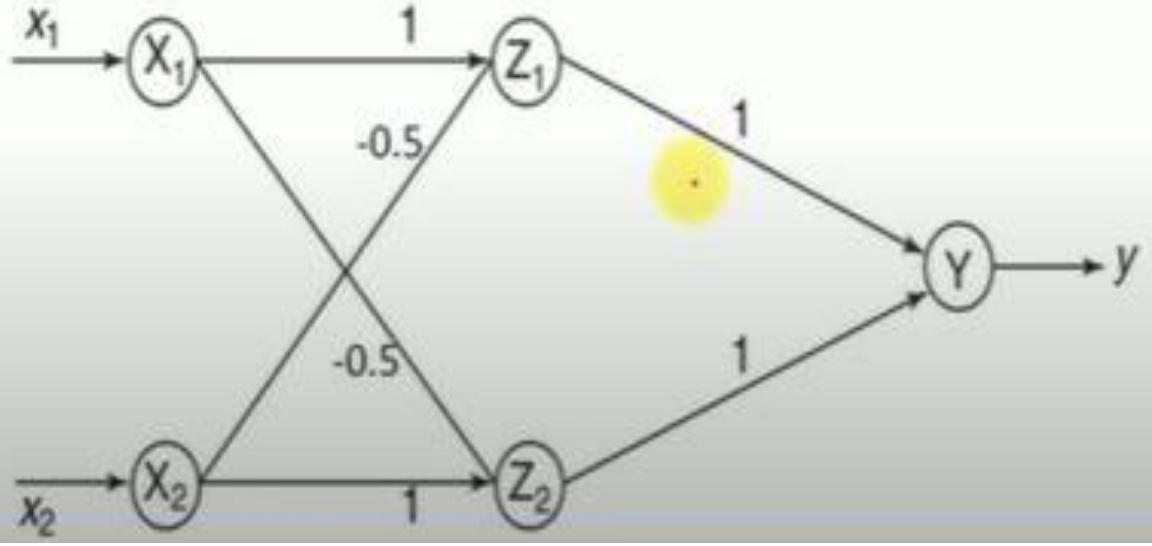


$w_{11} = 1 \; and \; w_{21} = -0.5$

$w_{12} = -0.5 \; and \; w_{22} = 1$

$v_1 = v_2 = 1$





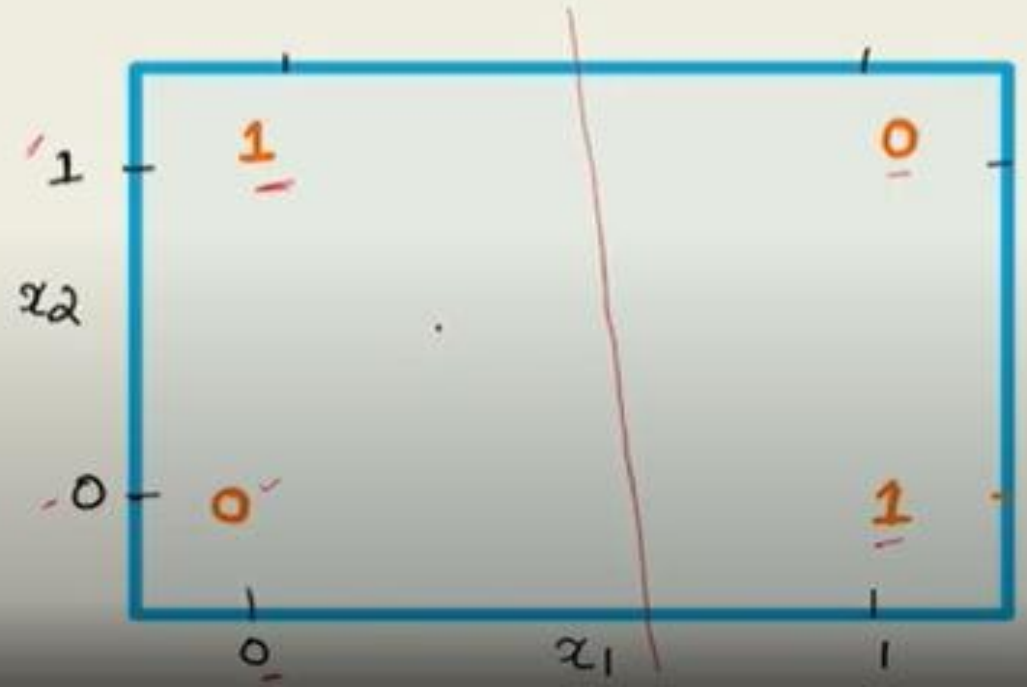
EXAMPLE: LEARNING XOR

Example: Learning XOR

- The XOR function ("exclusive or") is an operation on two binary values, x1 and x2
- When exactly one of these binary values is equal to 1, the XOR function returns 1.
- Otherwise, it returns 0
- these binary values is equal to 1, the XOR function returns 1.

XOR problem

XOR is not linearly Separable



we can not draw any line that Separates the 1s and 0s.

Solving XOR problem

Linear models cannot be used to solve XOR problem.

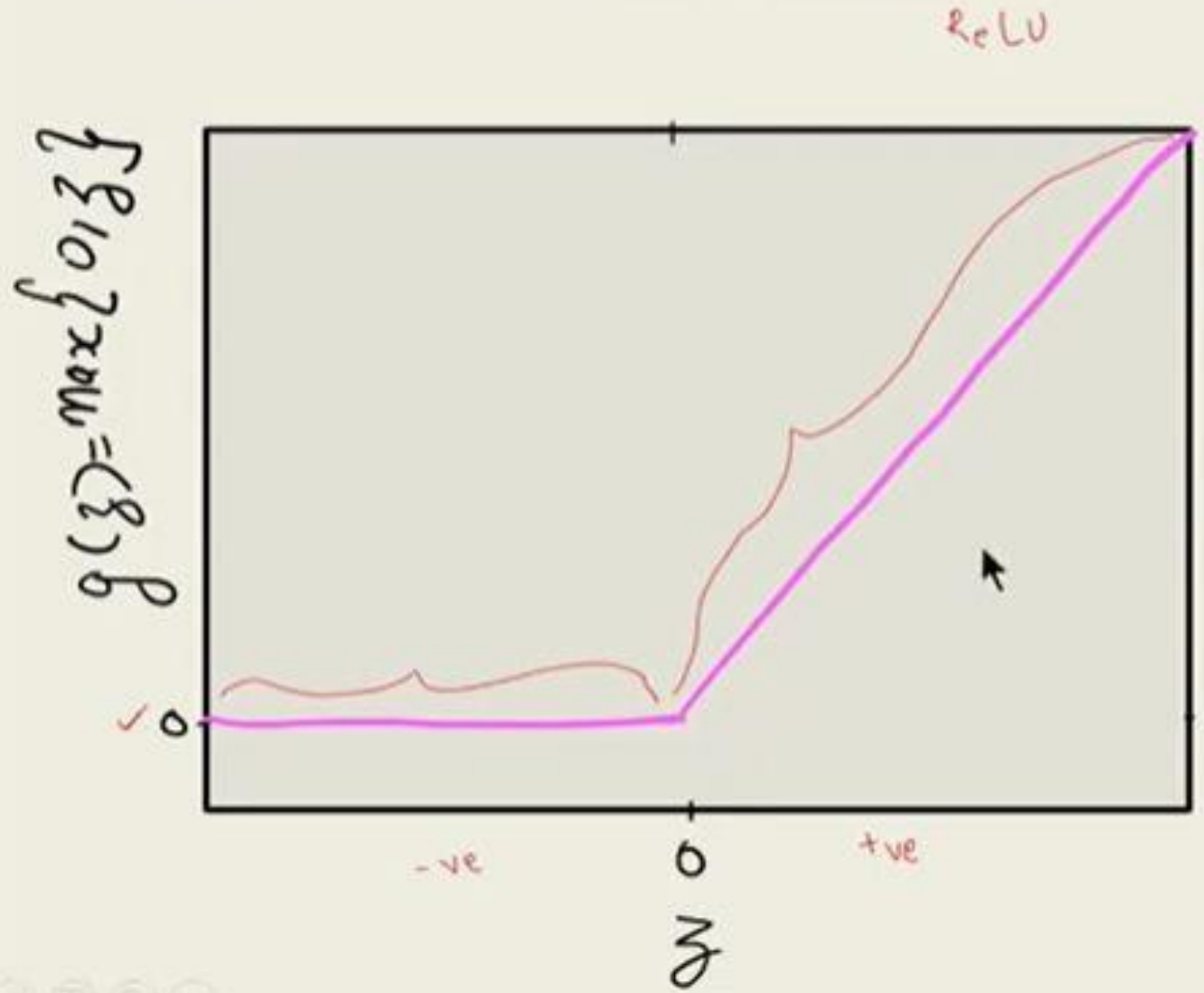
Hence we use activation function at hidden units.

We apply a linear transformation on data, and then apply non-linear activation function

↙

Rectified Linear function

The rectified linear activation function



g(z) gives 0 for -ve values

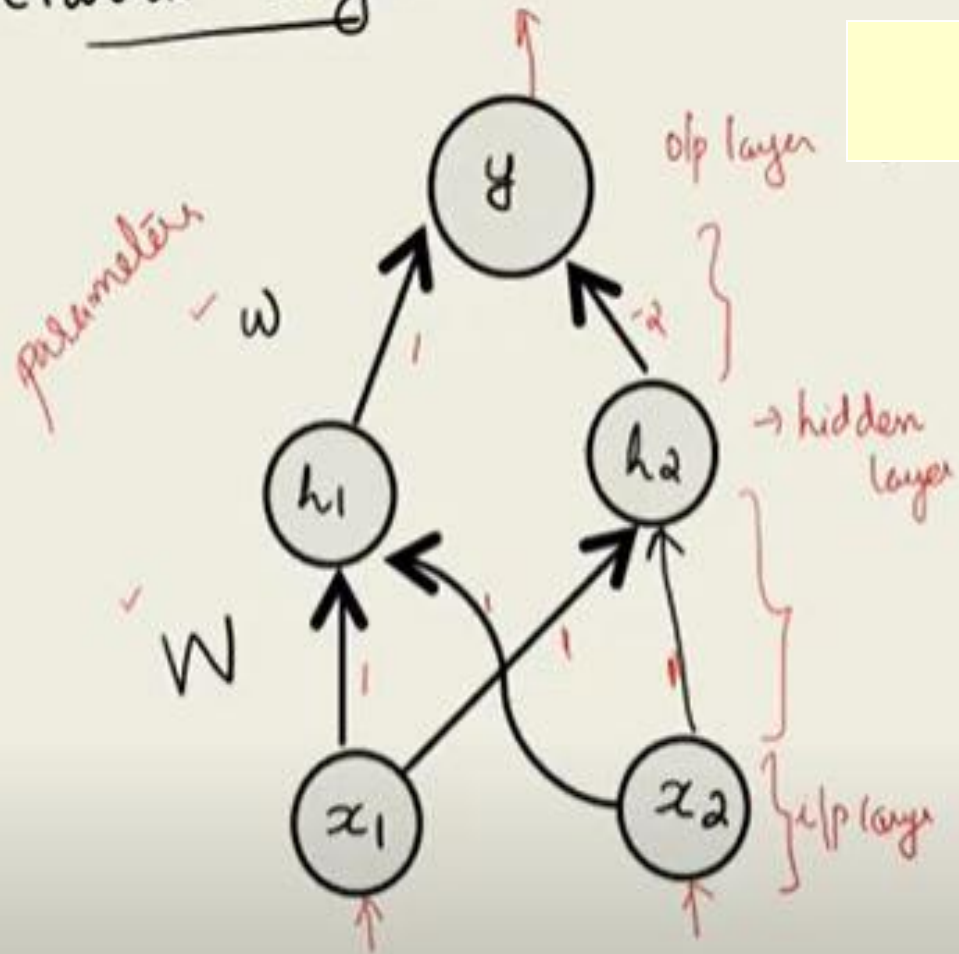
The rectified linear activation function

- This activation function is the default activation function recommended for use with most feedforward neural networks.
- Applying this function to the output of a linear transformation yields a nonlinear transformation.
- The function remains very close to linear, however, in the sense that is a piecewise linear function with two linear pieces.

The rectified linear activation function

- Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods.
- They also preserve many of the properties that make linear models generalize well.
- A common principle throughout computer science is that we can build complicated systems from minimal components.

Network Diagram



$w^T \max\{0, W^T x + c\}$

Solution

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

$$XW = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

add the bias vector c

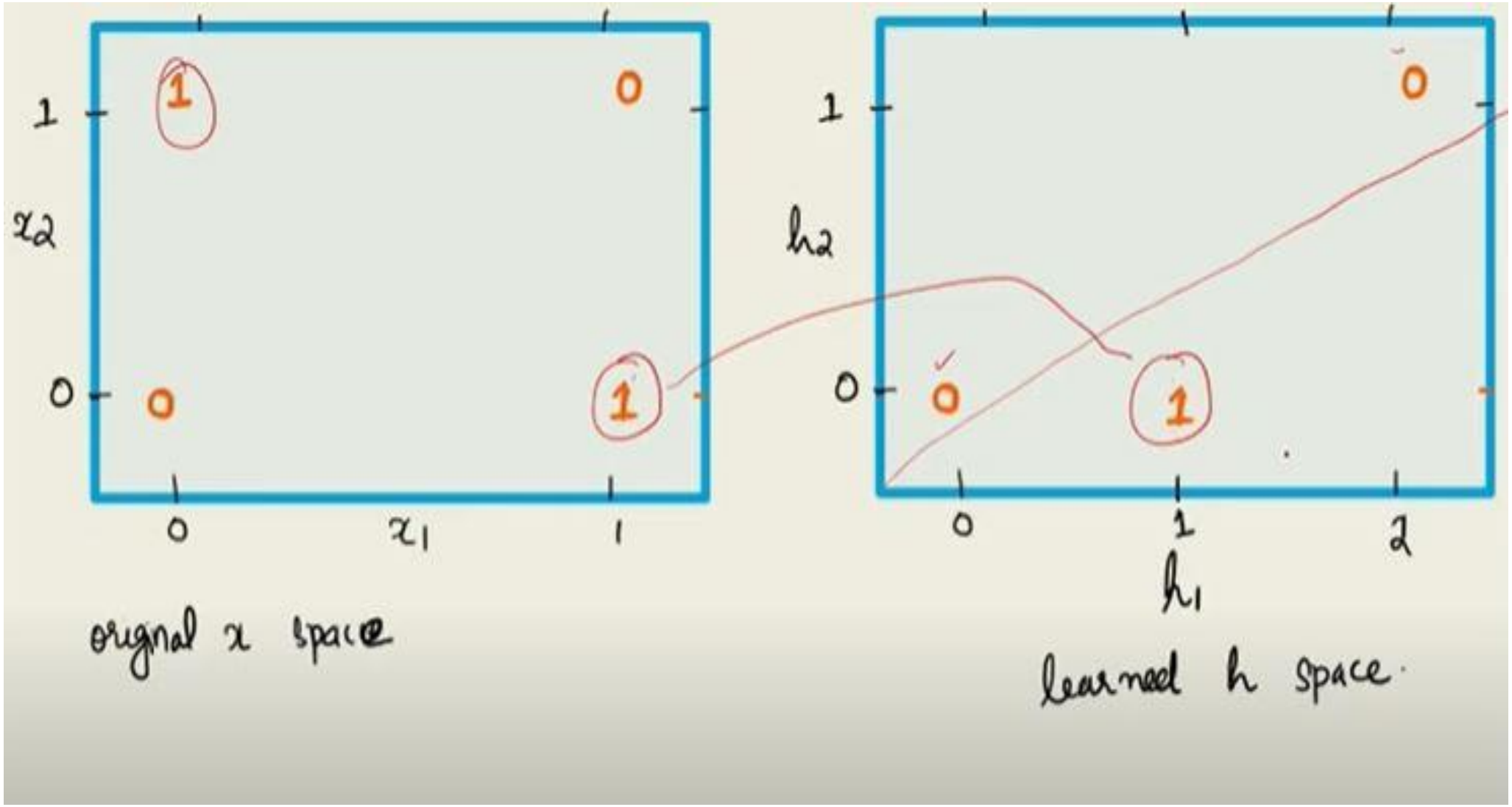
$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

apply the rectified linear transformation

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

multiplying by the weight vector w

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$


1
0
x_2
0
1
0
x_1
1
original x space
1
0
h_2
0
1
0
1
2
h_1
learned h space.

Solving XOR problem

- (Left) A linear model applied directly to the original input cannot implement the XOR function.
- When $x_1 = 0$, the model's output must increase as x_2 increases.
- When $x_1 = 1$, the model's output must decrease as x_2 increases.
- A linear model must apply a fixed coefficient w_2 to x_2.
- The linear model therefore cannot use the value of x_1 to change the coefficient on x_2 and cannot solve this problem.

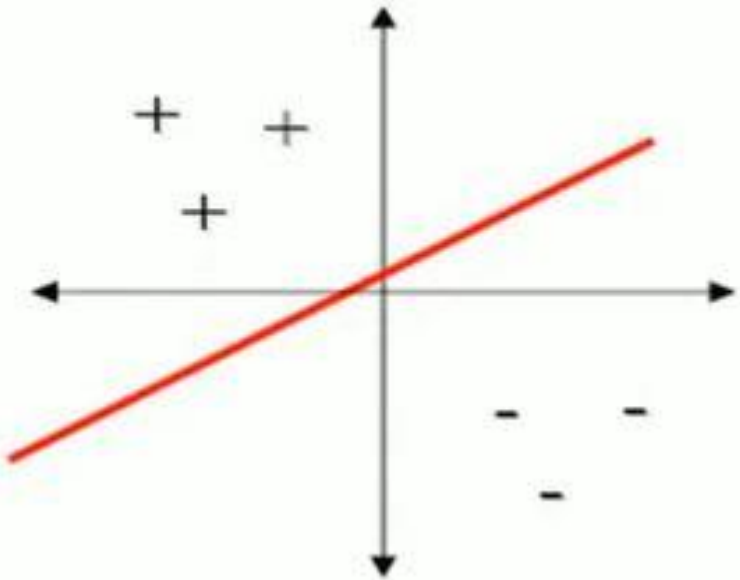
Solving XOR problem

- (Right) In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem.
- In our example solution, the two points that must have output 1 have been collapsed into a single point in feature space.
- In other words, the nonlinear features have mapped both $x = [1; 0]^T$ and $x = [0; 1]^T$ to a single point in feature space, $h = [1; 0]^T$.

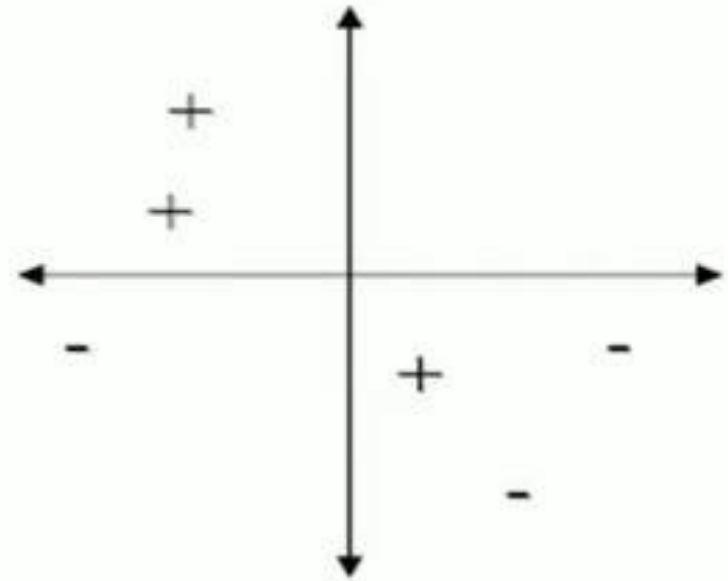
GRADIENT BASED LEARNING

- Machine learning approach that relies on optimization algorithms to adjust model parameters
- This process is typically implemented using backpropagation
- The gradients are computed recursively through layers of a neural network
- This process allows models to learn complex patterns and representation from data in training neural network

Gradient Descent and the Delta Rule



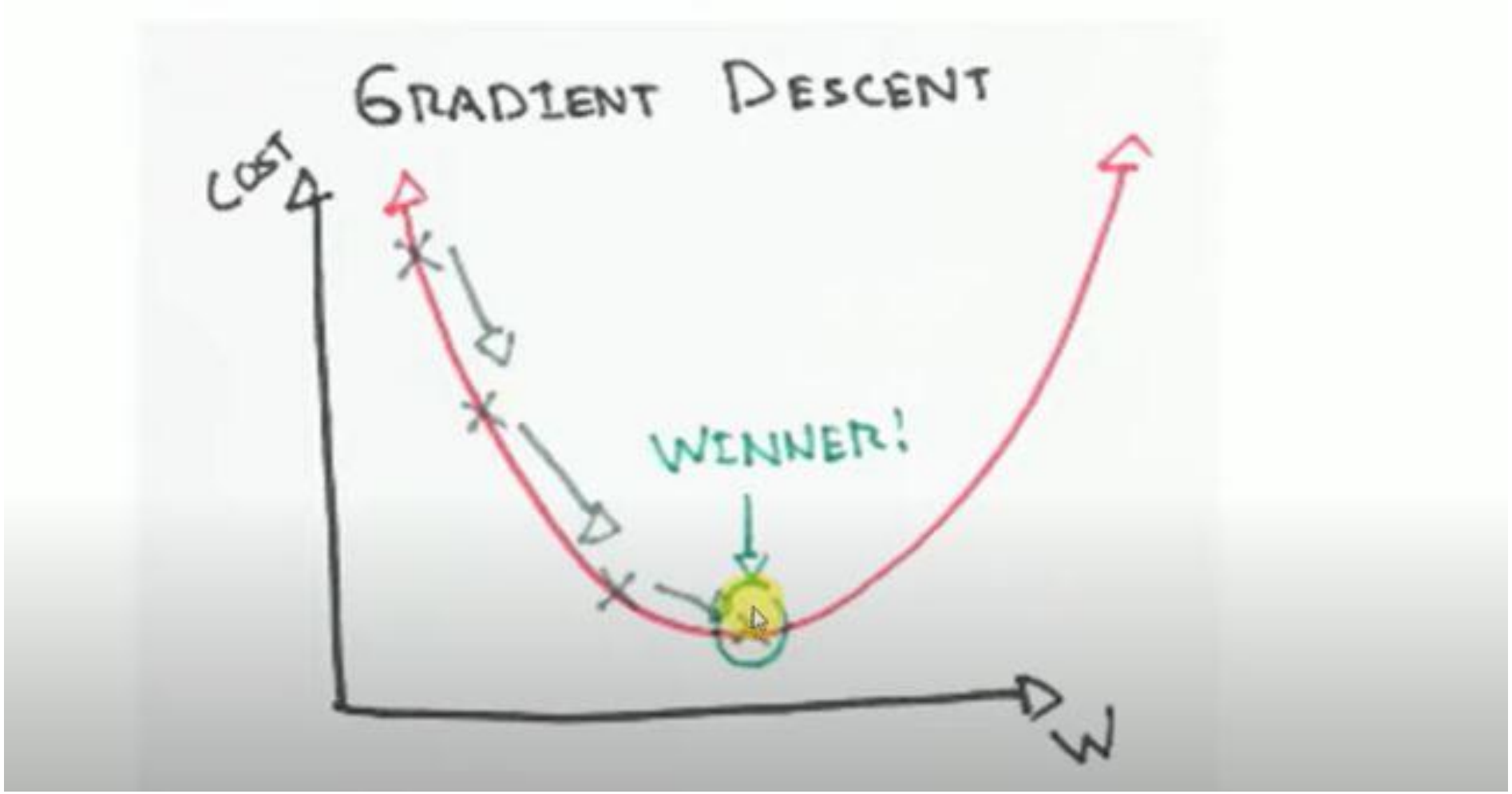
Linearly separable



Non-linearly separable

Gradient Descent and the Delta Rule

- Perceptron rule finds a successful weight vector when the training examples are linearly separable, but it can fail to converge if the examples are not linearly separable.
- A second training rule, called the ***delta rule,*** is designed to overcome this difficulty.
- If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.
- The key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.
- This rule is important because gradient descent provides the basis for the BACKPROPAGATON algorithm, which can learn networks with many interconnected units.
- It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.


GRADIENT DESCENT
COST
WINNER!
W

- The delta training rule is best understood by considering the task of training an ***unthresholded*** perceptron; that is, a ***linear unit*** for which the output ***o*** is given by

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

- Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.
- In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples.
- Although there are many ways to define this error, one common measure is

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- where D is the set of training examples, ***td*** is the target output for training example *d*, and ***od*** is the output of the linear unit for training example d.

Gradient Descent and the Delta Rule

- How can we calculate the direction of steepest descent along the error surface?
- This direction can be found by computing the derivative of E with respect to each component of the vector $\vec{w}$.
- This vector derivative is called the ***gradient*** of E with respect to $\vec{w}$, written ***as*** $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right]$$

- Since the gradient specifies the direction of steepest increase of E, the training rule for gradient descent is,

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

where

$$\Delta\vec{w} = -\eta \nabla E(\vec{w})$$

Gradient Descent and the Delta Rule

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

where

$$\Delta\vec{w} = -\eta \nabla E(\vec{w})$$

- Here *n* is a positive constant called the learning rate, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that ***decreases*** E.
- This training rule can also be written in its component form

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

where

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right]$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent and the Delta Rule

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id})$$

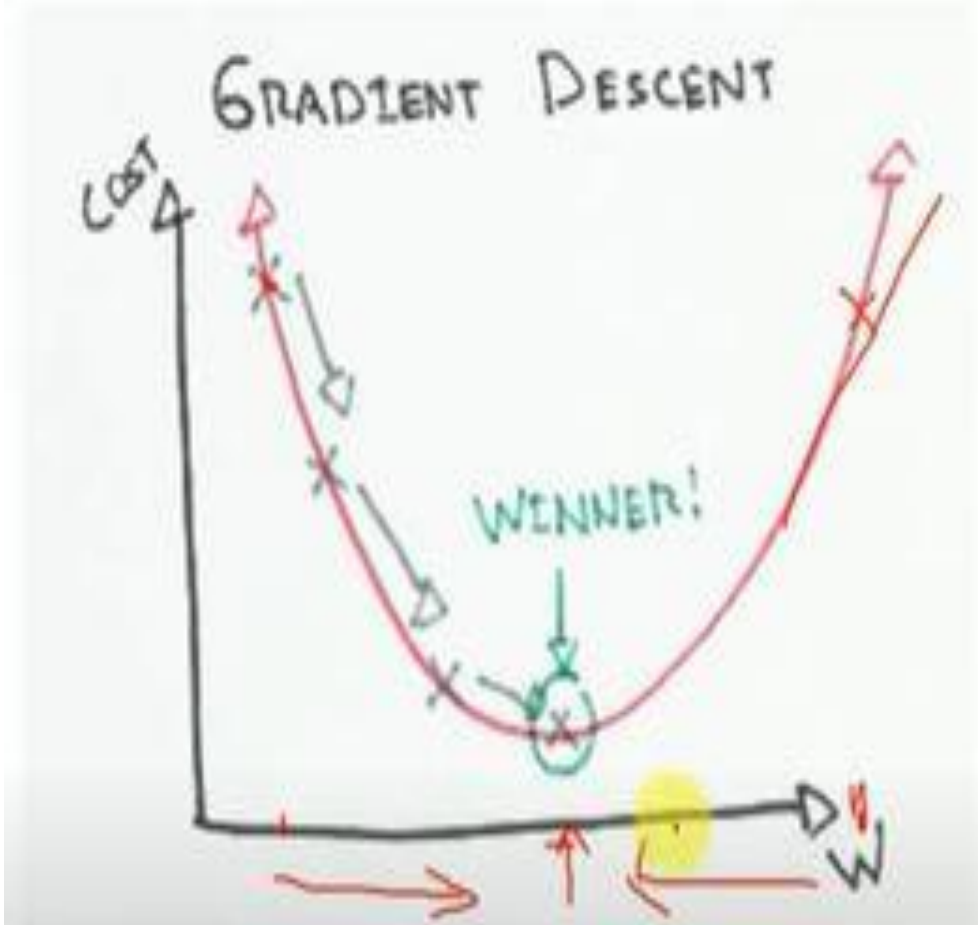
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id}$$

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



$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

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

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent - Delta Rule

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2 (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id})$$

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id}$$

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

Back Propagation Algorithm

BACKPROPAGATION (training_example, η, n_{in}, n_{out}, n_{hidden})

- Each training example is a pair of the form (x, t), where (x) is the vector of network input values, and (t) is the vector of target network output values.
- η is the learning rate (e.g., 0.05).
- n_i, is the number of network inputs,
- n_{hidden} the number of units in the hidden layer, and
- n_{out} the number of output units.
- The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}

Back Propagation Algorithm

- Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers
- Until the termination condition is met, Do
  - For each (x, t), in training examples, Do
    - Propagate the input forward through the network:

      1. Input the instance x, to the network and compute the output o_u of every unit u in the network.

    - Propagate the errors backward through the network

      2. For each network unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

      3. For each network unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

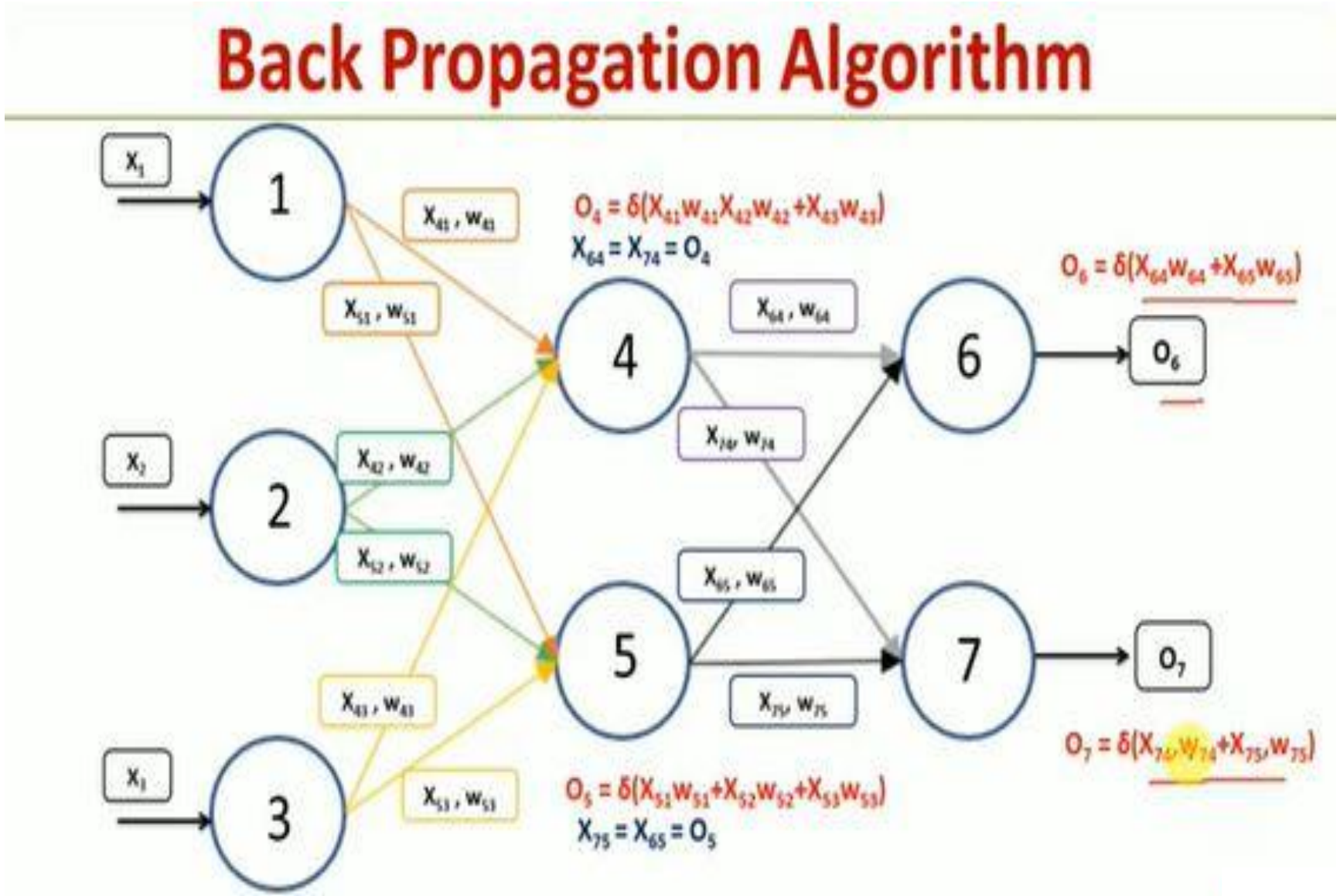
4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

Back Propagation Algorithm



DERIVATION OF BACKPROPOGATION

- To derive the equation for updating weights in back propagation algorithm, we use Stochastic gradient descent rule.
- Stochastic gradient descent involves iterating through the training examples one at a time, for each training example **d** descending the gradient of the error ***Ed*** with respect to this single example.
- In other words, for each training example **d** every weight ***wji*** is updated by adding to it Δw_{ij}.
- That is,

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Where

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Where

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

- where ***Ed*** is the error on training example **d**, that is half the squared difference between the target output and the actual output over all output units in the network,

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

- Here outputs is the set of output units in the network, $\mathbf{t_k}$ is the target value of unit k for training example **d**, and $\mathbf{o_k}$ is the output of unit k given training example **d**.

Notation Used:

x_{ji} = the i^{th} input to unit j

w_{ji} = the weight associated with the i^{th} input to unit j

$net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)

o_j = the output computed by unit j

t_j = the target output for unit j

σ = the sigmoid function

outputs = the set of units in the final layer of the network

Downstream(j) = the set of units whose immediate inputs include the output of unit j

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Where

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

- To begin, notice that weight ***wji*** can influence the rest of the network only through ***netj.*** Therefore, we can use the chain rule to write,

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

$$= \frac{\partial E_d}{\partial net_j} x_{ji}$$

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial net_j} x_{ji}$$

$$net_j = \sum_i w_{ji} X_{ji}$$

$$\frac{\partial net_j}{\partial w_{ji}} = x_{ji}$$

- Our remaining task is to derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$

To derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$

We consider two cases in turn:

- Case 1, where unit j is an output unit for the network, and
- Case 2, where unit j is an internal unit of the network.

Case 1: Training Rule for Output Unit Weights

- Just as ***wji*** can influence the rest of the network only through net_j, net_j can influence the network only through ***oj***. Therefore, we can invoke the chain rule again to write,

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2$$

$$\frac{\partial o_j}{\partial (net_j)} = \frac{\partial \sigma(net_j)}{\partial (net_j)}$$

$$\frac{\partial \sigma(x)}{\partial (x)} = \sigma(x)\,(1 - \sigma(x))$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2$$

$$= \frac{1}{2}2(t_j - o_j)\frac{\partial (t_j - o_j)}{\partial o_j}$$

$$= -(t_j - o_j)$$

$$= \sigma(net_j)\,(1 - \sigma(net_j))$$

$$= o_j\,(1 - o_j)$$

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\,o_j(1 - o_j)$$

Case 1: Training Rule for Output Unit Weights

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j(1 - o_j)$$

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial net_j} x_{ji}$$

$$\Delta w_{ji} = \eta\, (t_j - o_j)\, o_j(1 - o_j) x_{ji}$$

$$\delta_j = (t_j - o_j)\, o_j(1 - o_j)$$

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji}$$

Case 2: Training Rule for Hidden Unit Weights

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)$$

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j(1 - o_j)$$

$$\delta_j = (t_j - o_j)\, o_j(1 - o_j)$$

$$\frac{\partial net_k}{\partial o_j} = \frac{\partial x_{kj} w_{kj}}{\partial o_j} = \frac{\partial o_j w_{kj}}{\partial o_j}$$

$$\frac{\partial o_j}{\partial (net_j)} = \frac{\partial \sigma(net_j)}{\partial (net_j)}$$

$$= \sigma(net_j)\,(1 - \sigma(net_j))$$

$$= o_j\,(1 - o_j)$$

Case 2: Training Rule for Hidden Unit Weights

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial net_j} x_{ji}$$

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)$$

$$\Delta w_{ji} = \eta \, o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj} \, x_{ji}$$

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj}$$

HIDDEN UNITS

- Hidden Units In Deep Learning Refer To The Neuron Or Nodes With A Neural Networks Hidden Layer.
- They play a crucial role in learning complex patterns from data.
  - Fully connected (Dense) Layer

    All nodes are connected to every node in the previous and following layers .
  - Convolutional Layers

    Used for Image related tasks

    Neurons are connected in grid to local regions in input
  - Recurrent Layers

    Used in RNN'S for sequential data.

    They have connections that form cycles.

Architecture Design

- It Refers to the Creation and Configuration of the Neural Network Architectures
- This Involves Deciding the Number and type Of Layers ,the Connection Between these Layers and the Activation Function Used.
- Common Architectures Include CNN For Image Processing ,RNN For Sequential Data.
- It Is a Key Step in building effective Deep Learning Models

Differentiation algorithm

Differentiation algorithms are methods used to compute the derivative of a function.

- **Automatic Differentiation (AD)**

Automatic differentiation computes derivatives accurately to machine precision by applying the chain rule at the elementary operation level. It comes in two modes: forward mode and reverse mode.

- **Forward Mode**

  Evaluates derivatives along with the function evaluation.

- **Reverse Mode**

  Used primarily in machine learning for backpropagation, evaluates gradients of scalar-valued functions efficiently.

Automatic Differentiation: Highly efficient for computational tasks, especially in optimization problems.

- In deep learning, differentiation algorithms play a crucial role, particularly in the training process. The primary differentiation algorithm used in deep learning is backpropagation, which relies on automatic differentiation (AD), typically in its reverse mode.