MULTI-LAYERED PERCEPTRON AND BACKPROPAGATION MODULE 4

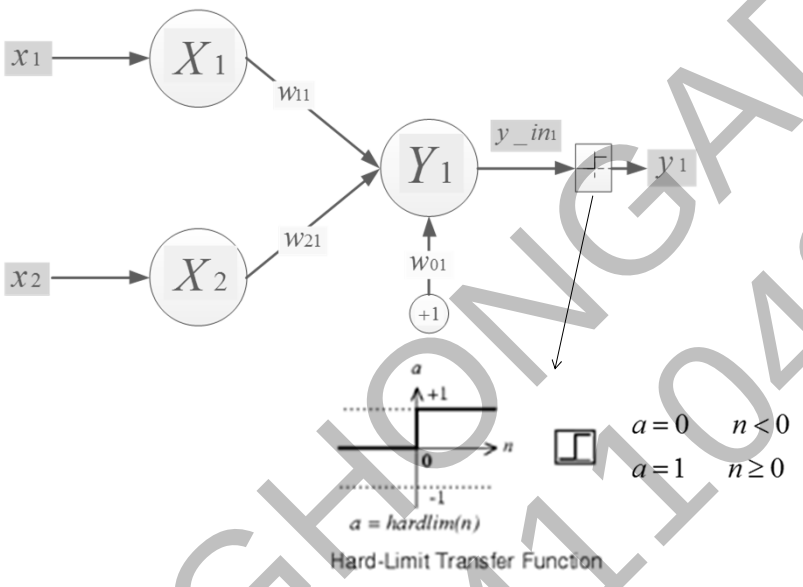
Dr. Rajesh B. Ghongade

Professor, Vishwakarma Institute of Information Technology, Pune-411048

Agenda

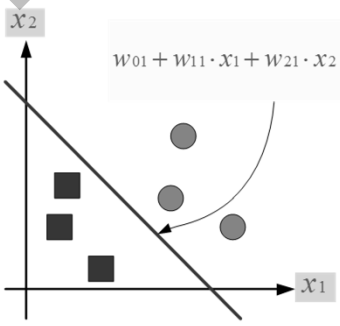
- Perceptron and its limitations
- MLP architecture
- Activation functions
- Gradient Descent Algorithm and Delta Rule
- Generalized Delta Rule(Backpropagation)
- Signal Flow
- Standard Backpropagation Algorithm
- XOR problem
- MATLAB Demo
- Some tips for net convergence
- Variations in standard backpropagation algorithm
- Applications of MLP trained with backpropagation algorithm

Perceptron and its limitation



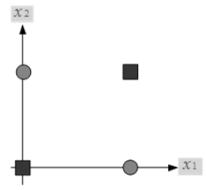
Hard-Limit Transfer Function

- Creates a linear separation boundary called as decision boundary
- Capable of classifying linearly separable objects only



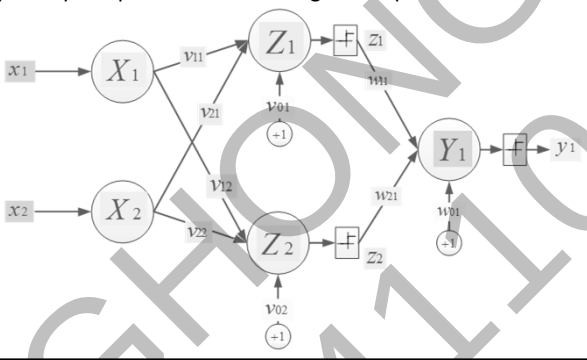
$$w_{01} + w_{11} \cdot x_1 + w_{21} \cdot x_2 = w_{01} + \sum_{j=1}^{2} w_{j1} \cdot x_j$$

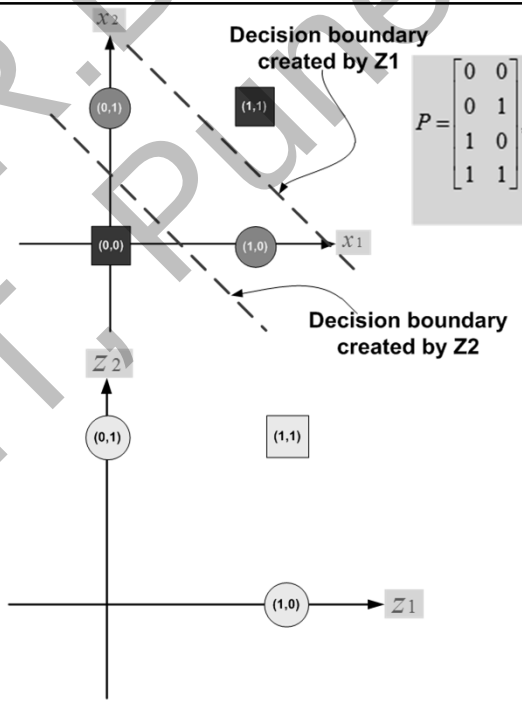
- Minsky and Papert (1969) showed that a perceptron is incapable of solving a simple XOR problem which is not linearly separable



x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

- Minsky and Papert (1969) also showed that such a problem can be solved by adding another layer of perceptron and combining the responses





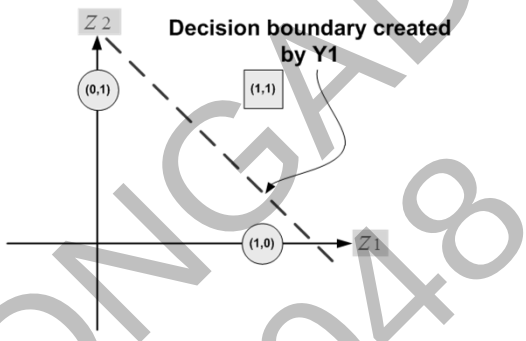
$$P=\begin{bmatrix}0&0\\0&1\\1&0\\1&1\end{bmatrix},\ x(1)=[0\ \ 0],\ x(2)=[0\ \ 1],\ x(3)=[1\ \ 0],\ x(4)=[1\ \ 1]$$

$$v=\begin{bmatrix}1&-1\\1&-1\end{bmatrix},\ v_0=\begin{bmatrix}-0.5\\1.5\end{bmatrix}$$

$$w=\begin{bmatrix}1\\1\end{bmatrix},\ w_0=[-1.5]$$

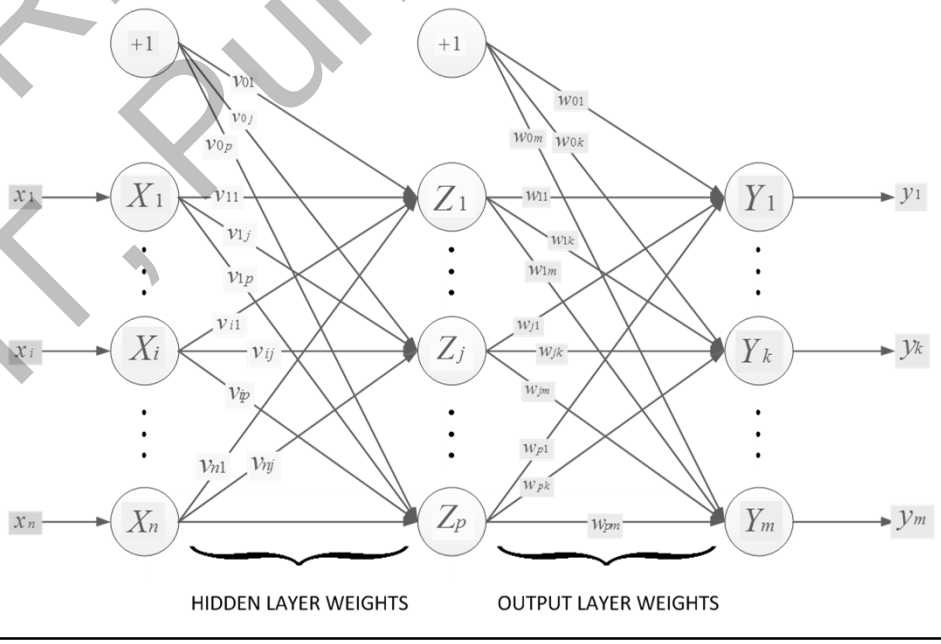
$$x(1)=[0\ \ 0]$$
$$z_1=\text{hardlim}\left([0\ \ 0]\bullet\begin{bmatrix}1&-1\\1&-1\end{bmatrix}+[-0.5\ \ 1.5]\right)=[0\ \ 1]$$
$$x(2)=[0\ \ 1]$$
$$z_1=\text{hardlim}\left([0\ \ 1]\bullet\begin{bmatrix}1&-1\\1&-1\end{bmatrix}+[-0.5\ \ 1.5]\right)=[1\ \ 1]$$
$$x(3)=[1\ \ 0]$$
$$z_1=\text{hardlim}\left([1\ \ 0]\bullet\begin{bmatrix}1&-1\\1&-1\end{bmatrix}+[-0.5\ \ 1.5]\right)=[1\ \ 1]$$
$$x(4)=[1\ \ 1]$$
$$z_1=\text{hardlim}\left([1\ \ 1]\bullet\begin{bmatrix}1&-1\\1&-1\end{bmatrix}+[-0.5\ \ 1.5]\right)=[1\ \ 0]$$

$$z(1)=[0\quad 1]$$
$$y_1=\text{hardlim}\left([0\quad 1]\bullet\begin{bmatrix}1\\1\end{bmatrix}+[-1.5]\right)=[0]$$
$$z(2)=[1\quad 1]$$
$$y_1=\text{hardlim}\left([1\quad 1]\bullet\begin{bmatrix}1\\1\end{bmatrix}+[-1.5]\right)=[1]$$
$$z(3)=[1\quad 1]$$
$$y_1=\text{hardlim}\left([1\quad 1]\bullet\begin{bmatrix}1\\1\end{bmatrix}+[-1.5]\right)=[1]$$
$$z(4)=[1\quad 0]$$
$$y_1=\text{hardlim}\left([1\quad 0]\bullet\begin{bmatrix}1\\1\end{bmatrix}+[-1.5]\right)=[0]$$

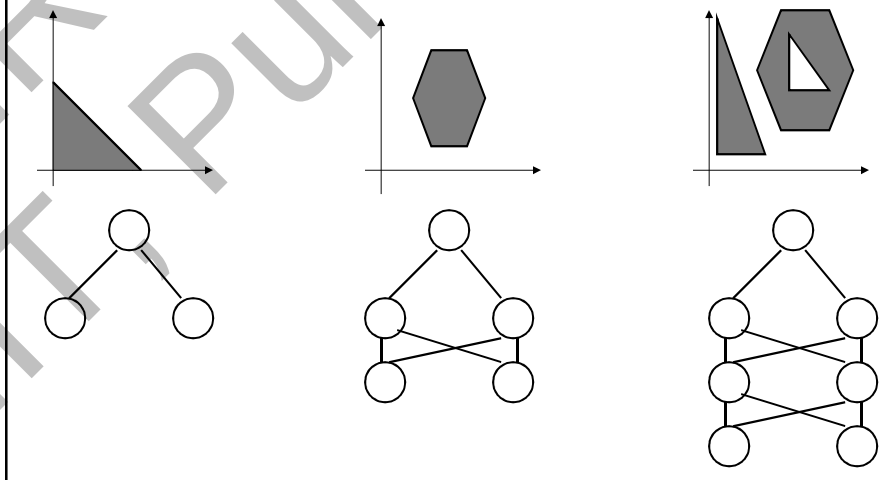


XOR problem can be solved, but how to find out all the weights and biases?

MLP Architecture



- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
- Often more than 2 layers
- Number of output units need not equal number of input units
- Number of hidden units per layer can be more or less than input or output units
- Can also view 1st layer as using local knowledge while 2nd layer does global
- With sigmoidal activation functions can show that a 2 layer net can approximate any function to arbitrary accuracy: property of Universal Approximation



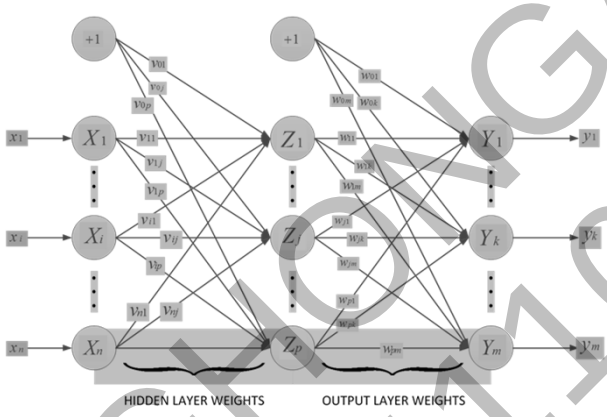
1st layer draws linear boundaries

2nd layer combines the boundaries

3rd layer can generate arbitrarily complex boundaries

Concept of layers!

Number of layers= layers of weights
or
Number of layers = layers of processing elements

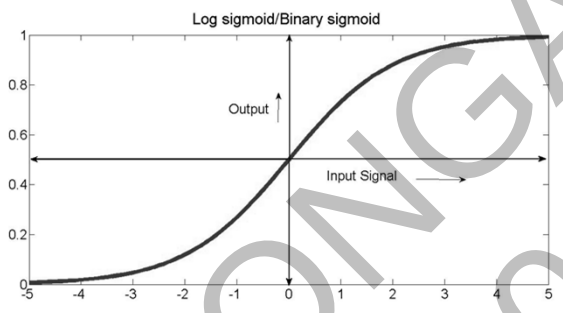


Activation Functions

- To map non-linear output functions we require non-linear activation functions
- The activation functions should be
  - Continuous
  - Monotonically increasing
  - Differentiable

Activation Functions

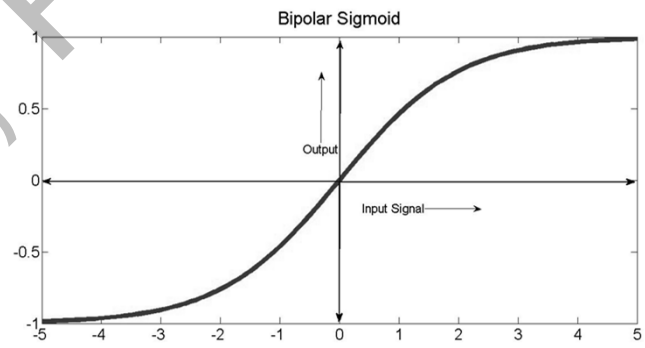
Log Sigmoid $f(x) = y = \dfrac{1}{1+e^{-x}}$ Output limits:[0,1]



Derivative $f'(x) = \dfrac{dy}{dx} = f(x)[1 - f(x)]$

Activation Functions

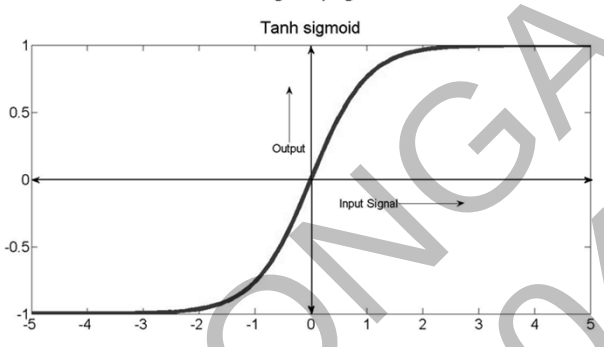
Bipolar Sigmoid $f(x) = y = \dfrac{2}{1+e^{-x}} - 1$ Output limits:[-1,1]



Derivative $f'(x) = \dfrac{dy}{dx} = \dfrac{1}{2}[1 + f(x)][1 - f(x)]$

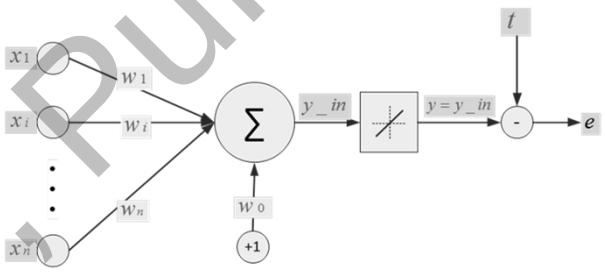
Activation Functions

Tanh Sigmoid $f(x) = y = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$ Output limits:[-1,1]



Derivative $f'(x) = \dfrac{dy}{dx} = \left[1 - \left(f(x)\right)^{2}\right]$

Gradient Descent Algorithm



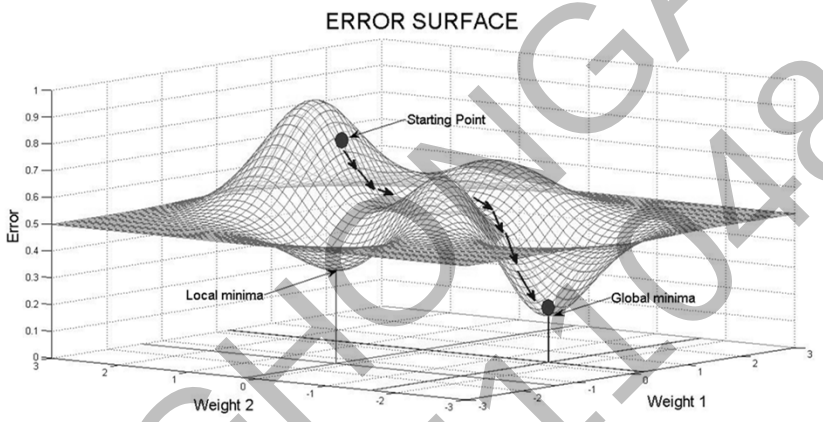
- Error e is defined as the deviation of the actual output y from the desired target t

$$e = t - y = t - y\_in = f(x_I, w_I)$$

where $y\_in = w_0 + \sum_{i=1}^{n} w_i \cdot x_i = \sum_{i=0}^{n} w_i \cdot x_i$

- Error energy is then: $E = e^{2} = \dfrac{1}{2}\left(t - y\_in\right)^{2}$

- In order to reduce squared error i.e, error energy E we have to find the partial derivative of E with respect to the weights w_I, and modify the weights in a direction opposite to the partial derivative, this is an optimization technique known as gradient descent.



- Hence we want to compute $\frac{\partial E}{\partial w_I}$ i.e. the gradient of error energy with respect to weight
- Once the gradient is obtained we can move along the opposite direction to the gradient in the *hope* of reaching the global minima
- Hence the weights have to be modified as: $\Delta w_I = -\frac{\partial E}{\partial w_I}$

But $$\frac{\partial E}{\partial w_I} = -(t - y\_in)\left(\frac{\partial y\_in}{\partial w_I}\right)$$

And $\frac{\partial y\_in}{\partial w_I} = x_I$ since $y\_in = w_0 + \sum_{i=1}^{n} w_i \cdot x_i = \sum_{i=0}^{n} w_i \cdot x_i$

hence $$\Delta w_I = -\frac{\partial E}{\partial w_I} = (t - y\_in) \cdot x_I$$

Thus the delta rule becomes $\Delta w_I = \alpha \cdot (t - y\_in) \cdot x_I$

where α is the learning rate (0< α>1)

Backpropagation

- In the perceptron/single layer nets, we used gradient descent on the error function to find the correct weights:

$$\Delta w_I = \alpha \cdot (t - y\_in) \cdot x_I$$

- We see that errors/updates are local to the node i.e. the change in the weight from node i to output j (w_{ij}) is controlled by the input that travels along the connection and the error signal from output j

- ***But with more layers how are the weights for the first 2 layers found when the error is computed for layer 3 only?***

- ***There is no direct error signal for the first layers!!!!!***

Derivation of Backpropagation Algorithm

- We shall denote the weight between hidden unit Z_j and the output unit Y_k by w_{jk}.
- The subscripts ij are used analogously for the weights between input X_i and the hidden unit Z_j.
- Let $f(x)$ be any arbitrary function with the derivative $f'(x)$.
- We desire to minimize the error E by modifying the weights w_{jk}
- Since $\frac{\partial E}{\partial w_{jk}}$ denotes the change of error E with respect to weights w_{jk} , we want to modify the weights in opposite direction to $\frac{\partial E}{\partial w_{jk}}$ hence

$$\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}}$$, where α =learning rate

- Let us find out the dependence of output error E on the weights in output and the hidden layers.
- We shall first define the error term to be minimized as:

$$e_k = t_k - y_k \quad \ldots\ldots\ldots\ldots\ldots\ldots (1)$$

$$E = \frac{1}{2}\sum_k \left(t_k - y_k\right)^2 = \frac{1}{2}\sum_k e^2 \quad \ldots\ldots\ldots\ldots\ldots\ldots (2)$$

$$E_{av} = \frac{1}{L}\sum_{l=1}^{L} E_l \quad \ldots\ldots\ldots\ldots\ldots\ldots (3)$$

- Output of neuron Y_k before applying activation function is

$$y\_in_k = w_{0k} + \sum_{j=1}^{p} z_j \cdot w_{jk} = \sum_{j=0}^{p} z_j \cdot w_{jk} \quad \ldots\ldots\ldots\ldots (4)$$

$$y_k = f(y\_in_k) \quad \ldots\ldots\ldots\ldots (5)$$

- We want to find out the contribution of weight w_{jk} to error E i.e; $\frac{\partial E}{\partial w_{jk}}$
- We can express it using chain rule as follows:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial e_k} \cdot \frac{\partial e_k}{\partial y_k} \cdot \frac{\partial y_k}{\partial y\_in_k} \cdot \frac{\partial y\_in_k}{\partial w_{jk}}$$

but $\frac{\partial E}{\partial e_k} = e_k \;, \frac{\partial e_k}{\partial y_k} = -1 \;, \frac{\partial y_k}{\partial y\_in_k} = f'(y\_in_k) \;, \frac{\partial y\_in_k}{\partial w_{jk}} = z_j$

hence $$\frac{\partial E}{\partial w_{jk}} = -e_k \cdot f'\left(y\_in_k\right) \cdot z_j$$

- Now we want to reduce the error E by changing weights w_{jk} proportionately as:

$$\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}}$$

where α =learning rate

hence $\Delta w_{jk} = \alpha \cdot e_k \cdot f'(y\_in_k) \cdot z_j$

- We define local gradient δ_k as:

$$\delta_k = e_k \cdot f'(y\_in_k) \quad \ldots\ldots\ldots\ldots\ldots\ldots (6)$$

- Hence weight update equation becomes:

$$\boxed{\Delta w_{jk} = \alpha \cdot \delta_k \cdot z_j}$$

Actually $\delta_k = -\frac{\partial E}{\partial y\_in_k}$, can shown by using chain rule again as follows:

$$\delta_k = -\frac{\partial E}{\partial y\_in_k} = \frac{\partial E}{\partial e_k} \cdot \frac{\partial e_k}{\partial y_k} \cdot \frac{\partial y_k}{\partial y\_in_k} = e_k \cdot (-1) \cdot f'(y\_in_k)$$

- ***Thus local gradient is the derivative of error energy with respect to its own induced field.***

Computing local gradient δ for hidden layer neuron:

- Computing local gradient is easy since the error value is directly available, but that for the hidden layer neurons requires more analysis
- We start again with the definition of the local gradient, now we want to compute the local gradient for the neuron j belonging to the hidden layer hence;

$$\delta_j = -\frac{\partial E}{\partial z\_in_j}$$

Using chain rule again to express the above equation:

$$\delta_j = -\frac{\partial E}{\partial z\_in_j} = -\frac{\partial E}{\partial z_j} \cdot \frac{\partial z_j}{\partial z\_in_j} = -\frac{\partial E}{\partial z_j} \cdot f'(z\_in_j)$$

$$\therefore \delta_j = -\frac{\partial E}{\partial z_j} \cdot f'(z\_in_j) \quad \ldots\ldots\ldots\ldots\ldots\ldots (7)$$

We have; $E = \frac{1}{2}\sum_k e^2$ ……(8)

We now compute $\frac{\partial E}{\partial z_j}$ as follows:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial e_k} \cdot \frac{\partial e_k}{\partial y_k} \cdot \frac{\partial y_k}{\partial y\_in_k} \cdot \frac{\partial y\_in_k}{\partial z_j}$$

$$\frac{\partial E}{\partial z_j} = \sum_k e_k \cdot \frac{\partial e_k}{\partial z_j} = \sum_k e_k \cdot \frac{\partial e_k}{\partial y_k} \cdot \frac{\partial y_k}{\partial y\_in_k} \cdot \frac{\partial y\_in_k}{\partial z_j} \quad \text{……(9)}$$

Note here that $\frac{\partial E}{\partial e_k}$ here is coming from **all output neurons** as against the previous case , hence we have to sum all the error terms.

Again

$$\frac{\partial e_k}{\partial y_k} = -1 \quad , \frac{\partial y_k}{\partial y\_in_k} = f'(y\_in_k) \quad , \frac{\partial y\_in_k}{\partial z_j} = w_{jk} \quad \text{……(10)}$$

Substituting Eq.10 in Eq.9, we get:

$$\frac{\partial E}{\partial z_j} = -\sum_k e_k \cdot f'(y\_in_k) \cdot w_{jk}$$

But $e_k \cdot f'(y\_in_k) = \delta_k$

hence we re-write the above equation as follows:

$$\frac{\partial E}{\partial z_j} = -\sum_k \delta_k \cdot w_{jk} \quad \text{……(11)}$$

Using Eq. 11 in Eq. 7 we get:

$$\delta_j = f'(z\_in_j) \cdot \sum_k \delta_k \cdot w_{jk} \quad \text{……(12)}$$

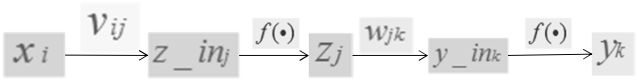
Again to reduce the error E by changing weights v_{ij} proportionately as:

$$\boxed{\Delta v_{ij} = \alpha \cdot \delta_j \cdot x_i}$$

Backpropagation Algorithm thus has THREE phases:

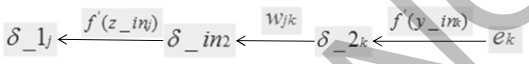
1.Forward phase

Where the input signal propagates in the forward direction



2.Error backpropagation phase

Where the error propagates in the reverse direction



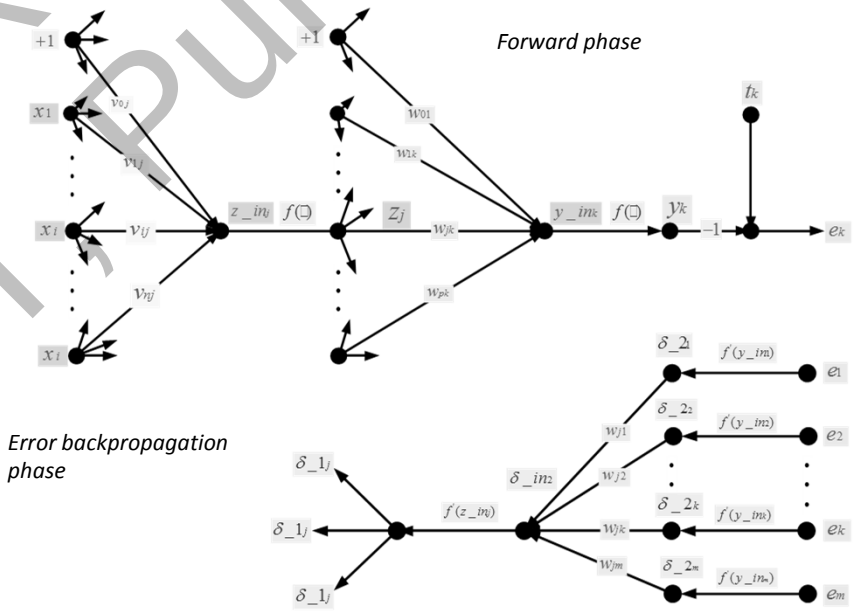
3.Weights and biases update phase

Where the local gradients are used to compute the weight and bias updates

$$\Delta w_{jk} = \alpha \cdot \delta\_2_k \cdot z_j \qquad \Delta w_{0k} = \alpha \cdot \delta\_2_k$$

$$\Delta v_{ij} = \alpha \cdot \delta\_1_j \cdot x_i \qquad \Delta v_{0j} = \alpha \cdot \delta\_1_j$$

Signal flow graph



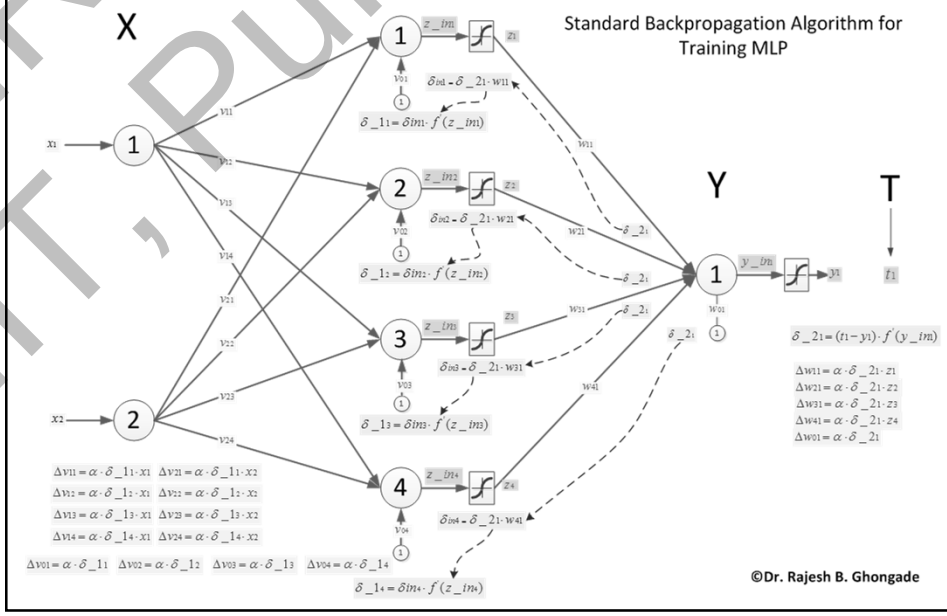
The standard backpropagation algorithm

STEP 0	Initialize weights. (Set to small random values).		
STEP 1	While stopping condition is false, do Steps 2-9.		
	STEP 2	For each training pair, do Steps 3-8.	
		FORWARD PHASE	
		STEP 3	Each input unit $(X_i,\ i=1,\ldots,n)$ receives input signal x_i and broadcasts this signal to all units in the hidden units layer).
		STEP 4	Each hidden unit $(Z_j,\ j=1,\ldots,p)$ sums its weighted input signals, $z\_in_j = v_{0j} + \sum_i x_i \cdot v_{ij}$ applies its activation function to compute its output signal, $z_j = f(z\_in_j)$ and sends this signal to all units in the output units layer
		STEP 5	Each output unit $(Y_k,\ k=1,\ldots,m)$ sums its weighted input signals, $y\_in_k = w_{0k} + \sum_j z_j \cdot w_{jk}$ applies its activation function to compute its output signal, $y_k = f(y\_in_k)$

BACKPROPAGATION PHASE	
STEP 6	Each output unit $(Y_k,\ k=1,\ldots,m)$ receives a target pattern corresponding to the input training pattern, computes its error information term, $\delta_k = (t_k - y_k) \cdot f'(y\_in_k)$ calculates its weight correction term (used to update w_{jk} later), $\Delta w_{jk} = \alpha \cdot \delta_k \cdot z_j$ calculates its bias correction term (used to update w_{0k} later), $\Delta w_{0k} = \alpha \cdot \delta_k$ and sends δ_k to units in the hidden layer .
STEP 7	Each hidden unit $(Z_j,\ j=1,\ldots,p)$ sums its delta inputs (from units in the output layer), $\delta\_in_j = \sum_k \delta_k \cdot w_{jk}$ multiplies by the derivative of its activation function to calculate its error information term, $\delta_j = \delta\_in_j \cdot f'(z\_in_j)$ calculates its weight correction term (used to update v_{ij} later), $\Delta v_{ij} = \alpha \cdot \delta_j \cdot x_i$ calculates its bias correction term (used to update v_{0j} later), $\Delta v_{0j} = \alpha \cdot \delta_j$

	WEIGHTS AND BIASES UPDATE PHASE	
	STEP 8	Each output unit $(Y_k,\ k=1,\ldots,m)$ updates its bias and weights $(j=0,\ldots,p)$ $w_{jk}(new)=w_{jk}(old)+\Delta w_{jk}$ Each hidden unit $(Z_j,\ j=1,\ldots,p)$ updates its bias and weights $(i=0,\ldots,n)$
STEP 9	Calculate mean square error $mse=\frac{0.5\sum_k (t_k-y_k)^2}{N}$ Test the stopping condition (max. iterations or if mse value is acceptable)	

XOR solution



MATLAB Demo

Some tips for net convergence

Choice of initial weights and biases

- Random Initialization**:** The choice of initial weights will influence whether the net reaches a global (or only a local) minimum of the error and; if so, how quickly it converges.
- The update of the weight between two units depends on both the derivative of the output unit's activation function and the activation of the hidden unit. ***For this reason, it is important to avoid choices of initial weights that would make it likely that either activations or derivatives of activations are zero***.
- The values for the initial weights must not be too large, or the initial input signals to each hidden or output unit will be likely to fall in the region where the derivative of the sigmoid function has a very small value (the so-called saturation region)
- On the other hand, if the initial weights are too small, the net input to a hidden or output unit will be close to zero, which also causes extremely slow learning.
- A common procedure is to initialize the weights (and biases) to random values between -0.5 and 0.5 (or between -1 and 1 or some other suitable interval)

How long to train the net ?

- Since the usual motivation for applying a backpropagation net is to achieve a balance between correct responses to training patterns and good responses to new input patterns (i.e., a balance between memorization and generalization), it is not necessarily advantageous to continue training until the total squared error actually reaches a minimum.
- Hecht-Nielsen (1990) suggests using two sets of data during training: a set of training patterns and a set cross-validation patterns. These two sets are disjoint.
- Weight adjustments are based on the training patterns; however, at intervals during training, the error is computed using the cross-validation patterns. As long as the error for the cross-validation patterns decreases, training continues. When the error begins to increase, the net is starting to memorize the training patterns too specifically (and starting to lose its ability to generalize). At this point, training is terminated.

How many training pairs there should be ?

- Under what circumstances can I be assured that a net which is trained to classify a given percentage of the training patterns correctly will also classify correctly testing patterns drawn from the same sample space?“
- Thumb rule dictates :

$$N = O\left(\frac{W}{e}\right)$$.....order of $\left(\frac{W}{e}\right)$

Where N= number of training exemplars, W=Number of free parameters to be adjusted (weights & biases), e=fraction of permissible classification error

- For example, with e = 0.1, a net with 80 weights will require 800 training patterns to be assured of classifying 90% of the testing patterns correctly, assuming that the net was trained to classify 95% of the training patterns correctly
- But experience suggests that the optimum number of training patterns is problem specific !

Data Representation

- It is recommended that the number of dimensions of the data be reduced by suitable methods , this process is called feature extraction
- Feature extraction methods like the Principal Component Analysis, Transforms like FFT, DCT, Wavelet have to be carefully chosen so that the intelligence in the data is preserved
- In general, it is easier for a neural net to learn a set of distinct responses than a continuous-valued response, therefore encoding the targets is important, we employ one-hot coding for the output neurons for classification problems

Number of Hidden Layer Neurons

- Researcher have attempted to find out optimal number of hidden layer neurons, but have failed so far!
- The number of hidden layer neurons is highly problem specific and has to be found out using brute-force technique.
- This technique is to simply start with a few hidden layer neurons and carrying out the training-testing phase over a larger number of neurons , to find out the maximum accuracy configuration

Number of Hidden Layers

- Generally one hidden layer is sufficient for a backpropagation net to approximate any continuous mapping from the input patterns to the output patterns to an arbitrary degree of accuracy.
- However, two hidden layers may make training easier in some situations

Choice of learning rate

- Since we use the gradient descent algorithm to reach a global minima , if it exists, learning rate plays an important role in the training of the net
- A very high learning rate can de-stabilize the net into producing oscillatory behavior
- A very low learning rate on the other hand slows down the learning and takes longer to reach the acceptable value of error
- Generally $0 < \alpha < 1$

Generalization

Affected by THREE factors:

1. Size of the training set
2. Architecture of the net
3. Physical complexity of the problem at hand

Variations in the standard backpropagation algorithm

Alternative Weight Update Procedures

Momentum: the weight change is in a direction that is a combination of the current gradient and the previous gradient.

$$w_{jk}(t+1) = w_{jk}(t) + \alpha \cdot \delta_k \cdot z_j + \mu\left[w_{jk}(t) - w_{jk}(t-1)\right]$$

where $\mu = momentum\ factor\ (0 < \mu < 1)$

Adaptive Learning Rates

Delta-Bar-Delta: Allow each weight to have its own learning rate, and to let the learning rates vary with time as training progresses

$$w_{jk}(t+1) = w_{jk}(t) - \alpha_{jk}(t+1) \cdot \frac{\partial E}{\partial w_{jk}} = w_{jk}(t) - \alpha_{jk}(t+1) \cdot \delta_k \cdot z_j$$

Applications of MLP trained with backpropagation algorithm

1. **Regression**
2. **Pattern Classification**
3. **Forecasting**

Thank you!