

## UNIT 2

Unit 2: Neural Networks- Concept, biological neural system,. Evolution of neural network, McCullochPitts neuron model, activation functions, feed forward and feedback networks, learning rules – Hebbian, Delta, Perceptron learning and Windrow-Hoff, winner-take-all. Supervised learning- Perceptron learning, single layer/multilayer perceptron, Adaptive resonance architecture, applications of neural networks to pattern recognition systems such as character recognition, face recognition, Application of Neural networks in Image processing.

### LECTURE-1

#### NEURAL NETWORK INTRODUCTION:

What is a neuron? A neuron is the basic processing unit in a neural network sitting on our brain. It consists of

1. Nucleus-
2. Axon- Output node
3. Dendrites-Input node
4. Synaptic junction

The dynamics of this synaptic junction is complex. We can see the signal inputs from the action of a neuron and through synaptic junction an output is actuated which is carried over through dendrites to another neuron. Here, these are the neurotransmitters. We learned from our experience that these synaptic junctions are either reinforced or in the sense they behave in such a way that the output of synaptic junction may excite a neuron or inhibit the neuron. This reinforcement of the synaptic weight is a concept that has been taken to artificial neural model.

The objective is to create artificial machine and this artificial neural networks are motivated by certain features that are observed in human brain, like as we said earlier, parallel distributed information processing.

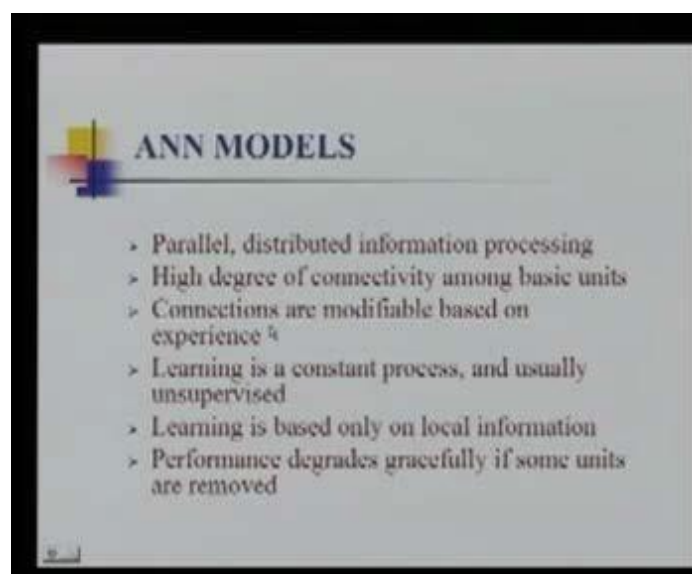


Fig. ANN model

Artificial neural networks are among the most powerful learning models. They have the versatility to approximate a wide range of complex functions representing multi-dimensional input-output maps. Neural networks also have inherent adaptability, and can perform robustly even in noisy environments.

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected simple processing elements (neurons) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well. ANNs can process information at a great speed owing to their highly massive parallelism.

A trained neural network can be thought of as an "expert" in the category of information it has been given to analyse. This expert can then be used to provide projections given new situations of interest and answer "what if" questions.

#### **Advantages of ANN:**

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.
3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

Table- Difference between the brain and a digital Computer

Property	Computer	Brain
Shape	2d Sheets of inorganic matter	3d volume of organic matter
Power	Powered by DC mains	Powered by ATP
Signal	Digital	pulsed
Clock	Centralized clock	No centralized clock
Clock speed	Gigahertz	100s of Hz
Fault tolerance	Highly fault-sensitive	Very fault-tolerant
Performance	By programming	By learning

#### **Differences human brain & ANN:**

1. Computer has such fast speed of GHz, a traditional computer, however, when it comes to certain processing like pattern recognition and language understanding, the brain is very fast.
2. Intelligence and self-awareness, are absent in an artificial machine.

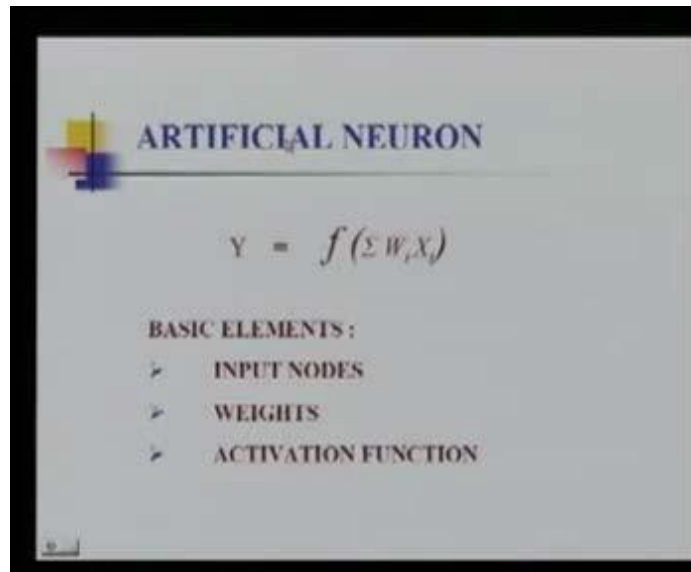


Fig. An artificial neuron

### **An Artificial Neuron:**

Basic computational unit in an artificial neural network is neuron. Obviously, it has to be an artificial neuron.

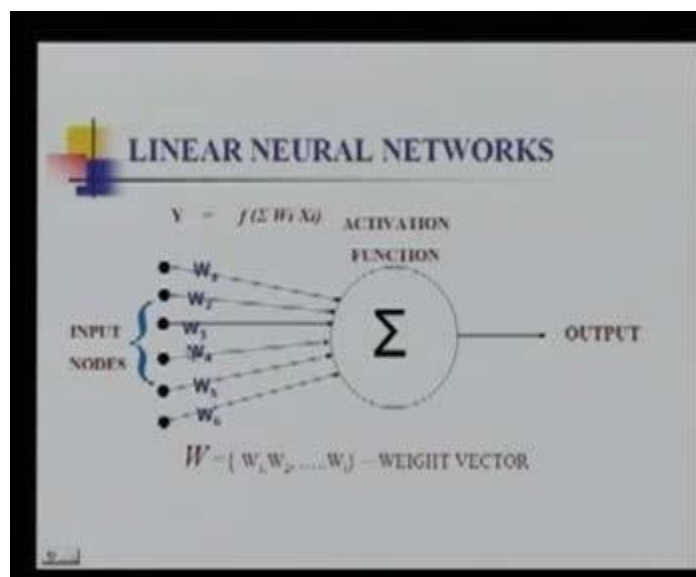


Fig. An artificial Neuron in linear NN

This artificial neuron has three basic elements:

1. Nodes,
2. Weights and
3. Activation function.

Between input nodes and output nodes, there are synaptic weights  $w_1, w_2, w_3, w_4, w_5$  and  $w_6$ . There can be as many weights and these weights are multiplied with the signal as they reach the output unit, where the output is simply sum of the signal multiplied with the weights and then this output goes to an activation function  $f$ .

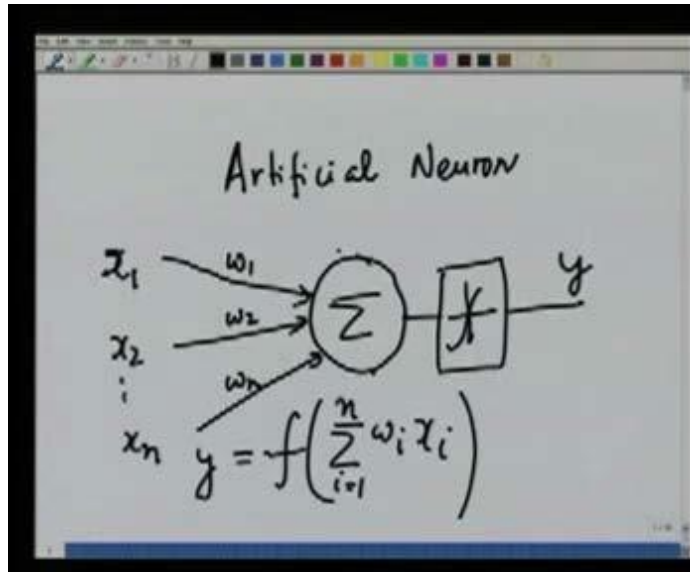


Fig. Basic processing unit- the neuron

In a simple neuron, if input signals be  $x_1, x_2, \dots, x_n$  with weights  $w_1, w_2, \dots, w_n$ . The weighted sum will activate this total output by an activation function  $f$ . That is your output. What you are seeing is actually a nonlinear map from input vector  $x$  to output  $y$ . A single neuron has single output but multiple inputs. Inputs are multiple for a single neuron and the output is unique,  $y$  and this output  $y$  and the input bear a nonlinear relationship, by  $f$ . Neural networks can be built using this single neuron. We can use the single neuron and build neural networks.

#### Analogy to brain:

Artificial Neural Network (ANN) is a system which performs information processing. An ANN resembles or it can be considered as a generalization of mathematical model of human brain assuming that

1. Information processing occurs at many simple elements called neurons.
2. Signals are passed between neurons over connection links.
3. Each connection link has an associated weight, which in a typical neural net multiplies the signal transmitted.

ANN is built with basic units called *neurons* which greatly resemble the neurons of human brain. A neural net consists of a large number of simple processing elements called neurons. Each neuron applies an activation function to its net input to determine its output signal. Every neuron is connected to other neurons by means of directed communication links, each with an associated weight. Each neuron has an internal state called its *activation level*, which is a function of the inputs it has received. As and when the neuron receives the signal, it gets added up and when the cumulative signal reaches the activation level the neuron sends an output. Till then it keeps receiving the input. So activation level can be considered as a threshold value for us to understand.

In general, a neural network is characterized by

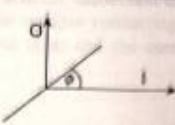
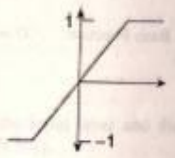
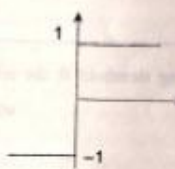
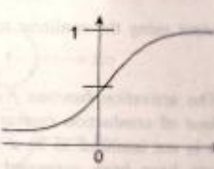
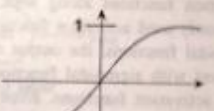
1. *Pattern of connections* between the neurons called its architecture
2. Method of *determining the weights* on the connections called its training or learning algorithm
3. Its *internal state* called its Activation function.

The arrangement of neurons into layers and the connection patterns within and between layers is called the net *architecture*. A neural net in which the signals flow from the input units to the output units in a forward direction is called *feed forward nets*.

Interconnected competitive net in which there are closed loop signal paths from a unit back to it is called a *recurrent network*. In addition to architecture, the method of setting the values of the weights called *training* is an important characteristic of neural nets. Based on the training methodology used neural nets can be distinguished into *supervised* or *unsupervised* neural nets. For a neural net with supervised training, the training is accomplished by presenting a sequence of training vectors or patterns each with an associated target output vector. The weights are then adjusted according to a learning algorithm. For neural nets with unsupervised training, a sequence of input vectors is provided, but no target vectors are specified. The net modifies the weights so that the most similar input vectors are assigned to the same output unit. The neural net will produce a representative vector for each cluster formed. Unsupervised learning is also used for other tasks, in addition to clustering.

## LECTURE-2

### Activation functions:

Type	Equation	Functional form
Linear	$O = \lambda I$ $\hat{g} = \tan \phi$	
Piecewise Linear	$O = \begin{cases} 1 & \text{if } \lambda I > 1 \\ \lambda I & \text{if }  \lambda I  < 1 \\ -1 & \text{if } \lambda I < -1 \end{cases}$	
Hard Limiter	$O = \text{sgn} [I]$	
Unipolar Sigmoidal	$O = \frac{1}{(1 + \exp(-\lambda I))}$	
Bipolar Sigmoidal	$O = \tanh [\lambda I]$	

Type	Equation	Functional form
Unipolar Multimodal	$O = \frac{1}{2} \left[ 1 + \frac{1}{M} \sum_{n=1}^M \tanh (g^n (I - W_0^n)) \right]$	
Radial Basis Function (RBF)	$O = \exp(I)$ $I = \left[ \frac{-\sum_{i=1}^N (W_i(t) - X_i(t))^2}{2\sigma^2} \right]$	

### Architecture:

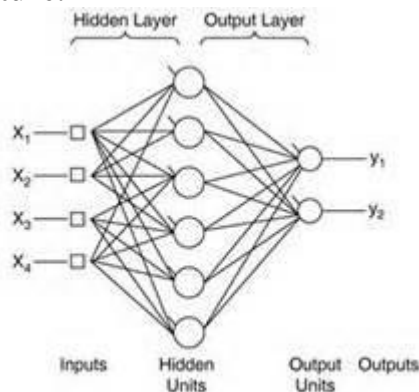


Fig. Architecture of multilayer Neural network

Artificial neural networks are represented by a set of nodes, often arranged in layers, and a set of weighted directed links connecting them. The nodes are equivalent to neurons, while the links denote synapses. The nodes are the information processing units and the links acts as communicating media.

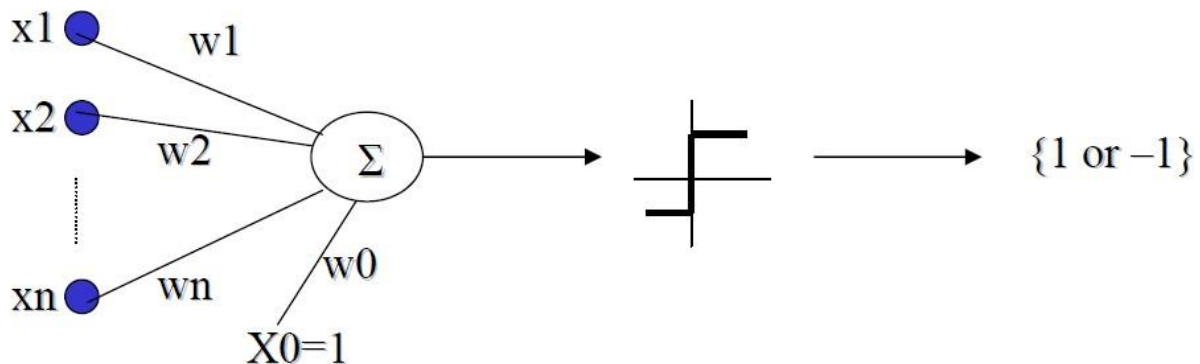
A neural network may have different layers of neurons like

1. input layer,
2. hidden layer,
3. output layer.

The input layer receives input data from the user and propagates a signal to the next layer called the hidden layer. While doing so it multiplies the weight along with the input signal. The hidden layer is a middle layer which lies between the input and the output layers. The hidden layer with non linear activation function increases the ability of the neural network to solve many problems than the case without the hidden layer. The output layer sends its calculated output to the user from which decision can be made. Neural nets can also be classified based on the above stated properties.

There are a wide variety of networks depending on the nature of information processing carried out at individual nodes, the topology of the links, and the algorithm for adaptation of link weights. Some of the popular among them include:

**Perceptron:** *Definition:* It's a step function based on a linear combination of real-valued inputs. If the combination is above a threshold it outputs a 1, otherwise it outputs a -1. This consists of a single neuron with multiple inputs and a single output. It has restricted information processing capability. The information processing is done through a transfer function which is either linear or non-linear.



$$O(x_1, x_2, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

Fig. A perceptron

A perceptron can learn only examples that are called “linearly separable”. These are examples that can be perfectly separated by a hyperplane.

Perceptrons can learn many boolean functions: AND, OR, NAND, NOR, but not XOR

However, every boolean function can be represented with a perceptron network that has two levels of depth or more.

The weights of a perceptron implementing the AND function is shown below.

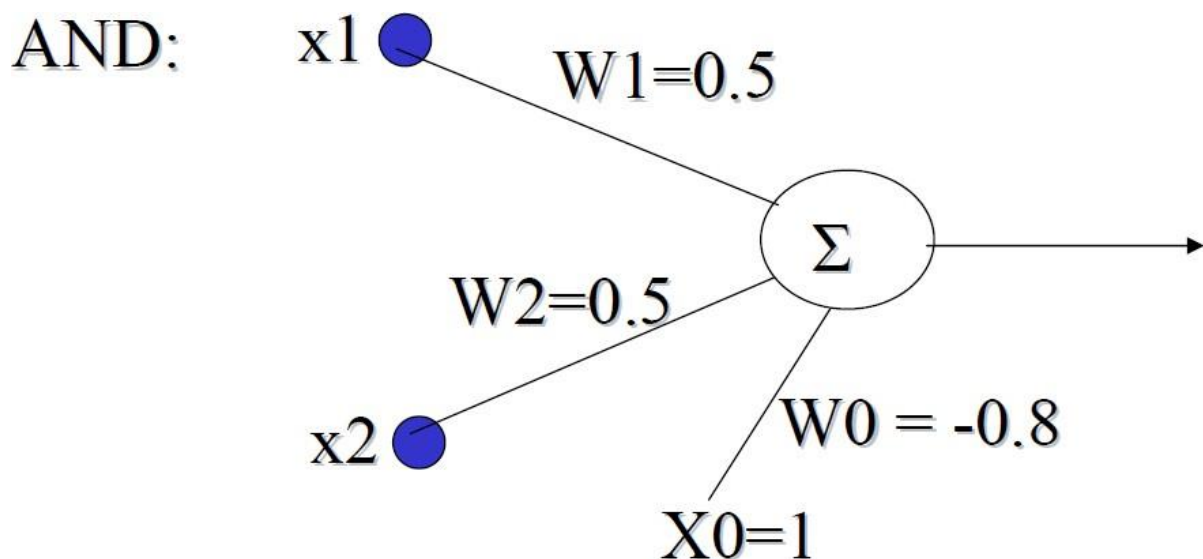


Fig. AND operation on inputs by a single perceptron

**Multi-layered Perceptron (MLP):** It has a layered architecture consisting of input, hidden and output layers. Each layer consists of a number of perceptrons. The output of each layer is



transmitted to the input of nodes in other layers through weighted links. Usually, this transmission is done only to nodes of the next layer, leading to what are known as feed forward networks. MLPs were proposed to extend the limited information processing capabilities of simple perceptrons, and are highly versatile in terms of their approximation ability. Training or weight adaptation is done in MLPs using supervised backpropagation learning.

#### **Adding a hidden layer:**

The perceptron, which has no hidden layers, can classify only linearly separable patterns.

The MLP, with at least 1 hidden layer can classify *any* linearly non-separable classes also.

An MLP can approximate any continuous multivariate function to any degree of accuracy, provided there are sufficiently many hidden neurons (Cybenko, 1988; Hornik et al, 1989). A more precise formulation is given below.

A serious limitation disappears suddenly by adding a single hidden layer.

It can easily be shown that the XOR problem which was not solvable by a Perceptron can be solved by a MLP with a single hidden layer containing two neurons.

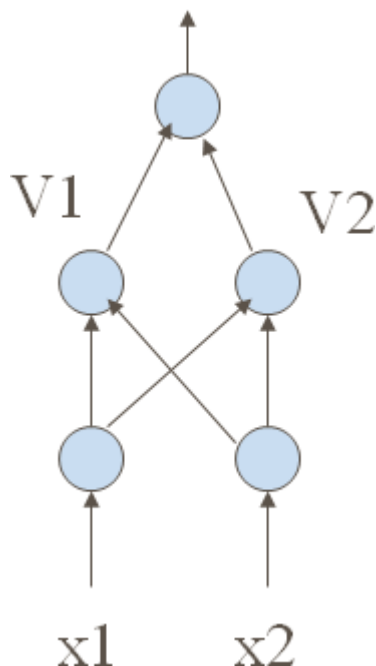


Figure 6.2.1.1: MLP for solving Xor

**Recurrent Neural Networks:** RNN topology involves backward links from output to the input and hidden layers. The notion of time is encoded in the RNN information processing scheme. They are thus used in applications like speech processing where inputs are time sequences data.



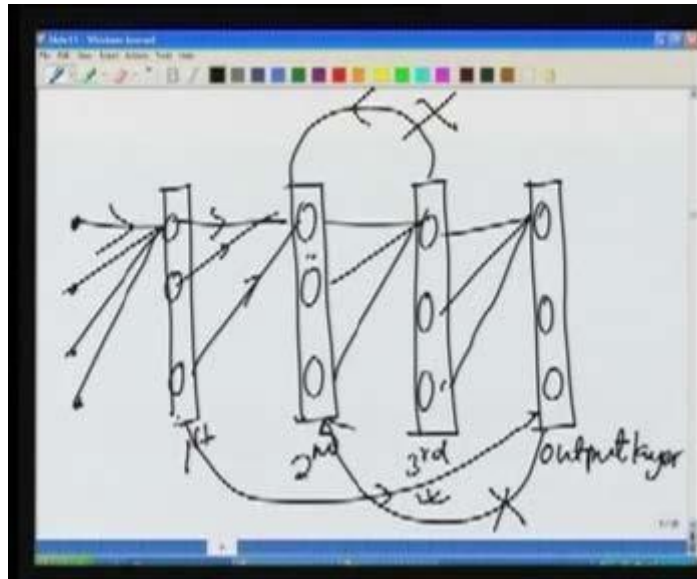


Fig. Multilayer feed back network (Recurrent Neural Network)

**Self-Organizing Maps:** SOMs or Kohonen networks have a grid topology, with unequal grid weights. The topology of the grid provides a low dimensional visualization of the data distribution. These are thus used in applications which typically involve organization and human browsing of a large volume of data. Learning is performed using a winner take all strategy in an unsupervised mode. It is described in detail later.

#### Single layer Network:

A neural net with only input layer and output layer is called single layer neural network. A neural network with input layer, one or more hidden layers and an output layer is called a multilayer neural network. A single layer network has limited capabilities when compared to the multilayer neural networks.

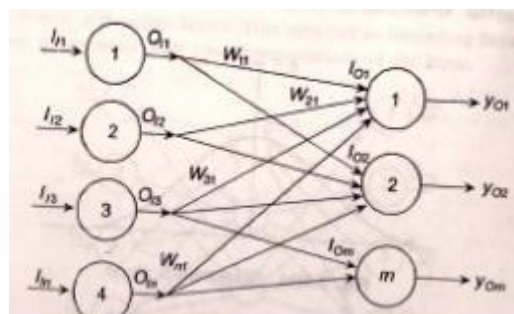


Fig. Single Layer feed forward Neural Network

### LECTURE-3

#### Steps in developing NN:

##### *Network formation*

Neural network consists of an input layer, an output layer and a hidden layer. While a neural network is constructed, the number of neurons in each layer has to be fixed. The input layer will have neurons whose number will be equal to the number of features extracted. The number of neurons in the output layer will be equal to the number of pattern classes. The

number of neurons in the hidden layer is decided by trial and error basis. With a minimum number of neurons in the hidden layer, the neural network will be constructed and the convergence will be checked for. Then the error will be noted. The number of neurons for which the error is minimum, can be taken and will be checked for reduced error criterion.

#### *Data preprocessing and normalization*

Data selection and pre processing can be a demanding and intricate task. Neural net is as good as the input data used to train it. If important data inputs are missing, then the effect on the neural network's performance can be significant. The most appropriate raw input data must be preprocessed. Otherwise the neural network will not produce accurate results. Transformation and normalization are two widely used preprocessing methods. Transformation involves manipulating raw data inputs to create a single input to a net, while normalization is a transformation performed on a single data input to distribute the data evenly and scale it into an acceptable range for the network. Knowledge of the domain is important in choosing preprocessing methods to highlight the features in the data, which can increase the ability of the network to learn the association between inputs and outputs. Data normalization is the final preprocessing step. In normalizing data, the goal is to ensure that the statistical distribution of values should be scaled to match the range of the input neurons. The simplest method of normalization can be done using the formula

$X \text{ normalized} = (X - \mu) / \sigma$  where  $\mu$  and  $\sigma$  are the mean and standard deviation of the input data.

#### **Perceptron Learning**

Learning a perceptron means finding the right values for  $W$ . The hypothesis space of a perceptron is the space of all weight vectors.

The perceptron learning algorithm can be stated as below.

1. Assign random values to the weight vector
2. Apply the *weight update rule* to every training example
3. Are all training examples correctly classified?
  - a. Yes. Quit
  - b. No. Go back to Step 2.

There are two popular weight update rules.

- i) The perceptron rule, and
- ii) Delta rule

#### **The Perceptron Rule**

For a new training example  $X = (x_1, x_2, \dots, x_n)$ , update each weight according to this rule:

$$w_i = w_i + \Delta w_i$$

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

$t$ : target output

$o$ : output generated by the perceptron

$\eta$ : constant called the learning rate (e.g., 0.1)

Comments about the perceptron training rule:

Example means training data.

- If the example is correctly classified the term  $(t - o)$  equals zero, and no update on the weight is necessary.
- If the perceptron outputs  $-1$  and the real answer is  $1$ , the weight is increased.
- If the perceptron outputs a  $1$  and the real answer is  $-1$ , the weight is decreased.
- Provided the examples are linearly separable and a small value for  $\eta$  is used, the rule is proved to classify all training examples correctly (i.e, is consistent with the training data).

#### **The Delta Rule**

What happens if the examples are not linearly separable?

To address this situation we try to approximate the real concept using the delta rule.

The key idea is to use a *gradient descent search*. We will try to minimize the following error:

$$E = \frac{1}{2} \sum_i (t_i - o_i)^2$$

where the sum goes over all training examples. Here  $o_i$  is the inner product  $WX$  and not  $\text{sgn}(WX)$  as with the perceptron rule. The idea is to find a minimum in the space of weights and the error function  $E$ .

**The delta rule is as follows:**

For a new training example  $X = (x_1, x_2, \dots, x_n)$ , update each weight according to this rule:

$$w_i = w_i + \Delta w_i$$

$$\text{Where } \Delta w_i = -\eta E'(W)/w_i$$

$\eta$ : learning rate (e.g., 0.1)

It is easy to see that

$$E'(W)/w_i = \sum_i (t_i - o_i) (-x_i)$$

So that gives us the following equation:

$$w_i = \eta \sum_i (t_i - o_i) x_i$$

There are two differences between the perceptron and the delta rule. The perceptron is based on an output from a step function, whereas the delta rule uses the linear combination of inputs directly. The perceptron is guaranteed to converge to a consistent hypothesis assuming the data is linearly separable. The delta rule converges in the limit but it does not need the condition of linearly separable data.

There are two main **difficulties with the gradient descent method**:

1. Convergence to a minimum may take a long time.
  2. There is no guarantee we will find the global minimum.
- These are handled by using momentum terms and random perturbations to the weight vectors.

## LECTURE-4

### ADALINE & MADALINE:

The Adaline networks (ADaptive LINEar Element) and Madaline (Multiple Adaline) were developed by Widrow. The structures use neurons and step/ sigmoidal activation function. ADALINE has one output neuron but MADALINE has many. The learning is different from a perceptron. It is here by Widrow-Hoff or LMS (Least Mean Square error) rule. Analogical input or output can be found by this network as minimum error function is searched before applying activation function.

#### ADALINE:

The structure includes an adaptive linear combiner (ALC) to obtain linear response that can be applied to other elements of bipolar commutation. IF O/P of ALC is +ve response of ADALINE is +1, and if -ve result of ADALINE is -1. It is represented by:

$$y(t+1) = \begin{cases} +1 & s > 0 \\ y(t) & s = 0 \\ -1 & s < 0 \end{cases}$$

The binary answer corresponding to the Adaline network is:

$$s = \sum_{j=0}^N w_j x_j = \mathbf{w}^T \mathbf{X} \quad (3.11)$$

The Adaline Network can be used to generate an analogical response using a sigmoid commuter, instead of a binary one, in such a case the output  $y$  will be obtained applying a sigmoidal function.

The Adaline and Madaline Networks use a supervised learning rule, off-line, denominated Least Mean Squared (LMS), through which the vector of weight,  $\mathbf{w}$ , should associate with success each input pattern with its corresponding value of desired output,  $d_k$ . Particularly, the network training consists in modifying the weights when the training patterns and desired outputs are presented. With each combination input-output an automatic process is made of little adjustments in the weights values until the correct outputs are obtained.

Concretely, the learning rule LMS minimizes the mean squared error, defined as:

$$E = \frac{1}{2P} \sum_{k=1}^P \varepsilon_k^2 \quad (3.13)$$

Where  $P$  is the quantity of input vectors (patterns) that form the training group, and  $\varepsilon_k$  is the difference between the desired output and the obtained when is introduced the  $k$ -th pattern, in the case of the Adaline network, it expressed as  $\varepsilon_k = (d_k - s_k)$ , where  $s_k$  corresponds to the exit of the ALC (expression 3.11), that is to say:

$$s_k = \sum_{j=0}^N w_j x_{jk} = \mathbf{w}^T \mathbf{X}_k \quad (3.14)$$

Even when the error surface is unknown, equation (3.13), the gradient descent method gets to obtain the local information of it. With this information it is decided what direction to take to arrive to the global minimum. Therefore, based in the gradient descent method, the rule known as delta rule or LMS rule is obtained. With this the weight are modified so that the new point, in the space of weights, is closer to the minimum point. In other words, the modification of the weights is proportional to the gradient descent of the error function  $\Delta w_j = -\eta(\partial E_k / \partial w_j)$ . Using the chain rule for the calculus of the derivative of expression (3.13), we obtain:

$$w_j(t+1) = w_j(t) + \eta \varepsilon_k x_{kj} \quad (3.15)$$

Based in this, the learning algorithm of an Adaline network contains the following steps:

1. A pattern is introduced  $X_k = (x_{k1}, x_{k2}, \dots, x_{kN})$  in the entrances of Adaline
2. The linear output is obtained  $s_k = \sum_{j=0}^N w_j x_{jk} = \mathbf{w}^T \mathbf{X}_k$  and it is calculated the difference with respect from what is expected  $\varepsilon_k = (d_k - s_k)$ .
3. The weights are actualized.

$$\begin{aligned} \mathbf{w}(t+1) &= \mathbf{w}(t) + \eta \varepsilon_k \mathbf{X}_k \\ w_j(t+1) &= w_j(t) + \eta \varepsilon_k x_{kj} \end{aligned}$$

4. Steps from 1 to 3 are repeated, with all the entrance vectors.
5. If the mean squared error

$$E = \frac{1}{2P} \sum_{k=1}^P \varepsilon_k^2$$

Has a reduced value, the learning process ends; if not, the process is repeated from step one with all the patterns.

### Madaline network

The Madaline network (Multiple Adaline) is formed as a combination of Adaline modules, which are structured in layers (figure 3.4). Madaline overcomes some of the limitations of Adaline networks.

Given the differences between Madaline and Adaline, the training of these networks is not the same. The LMS algorithm can be applied to the output layer because its exit is already known for each one of the entrance patterns. However, the expected exit is unknown for the nodes of the hidden layer. Besides, the algorithm LMS works for the linear outputs (analogical), of the adaptive combiner and not for the digital of the Adaline.

To employ the algorithm LMS in the Madaline training is required to modify the activation function for a derivable continuous function (the step function is discontinuous in zero and therefore not derivable in this point).

Because of its similarities with the Multilayer Perceptron, the description of the Madaline learning rule will not be described, this will be handled lately.

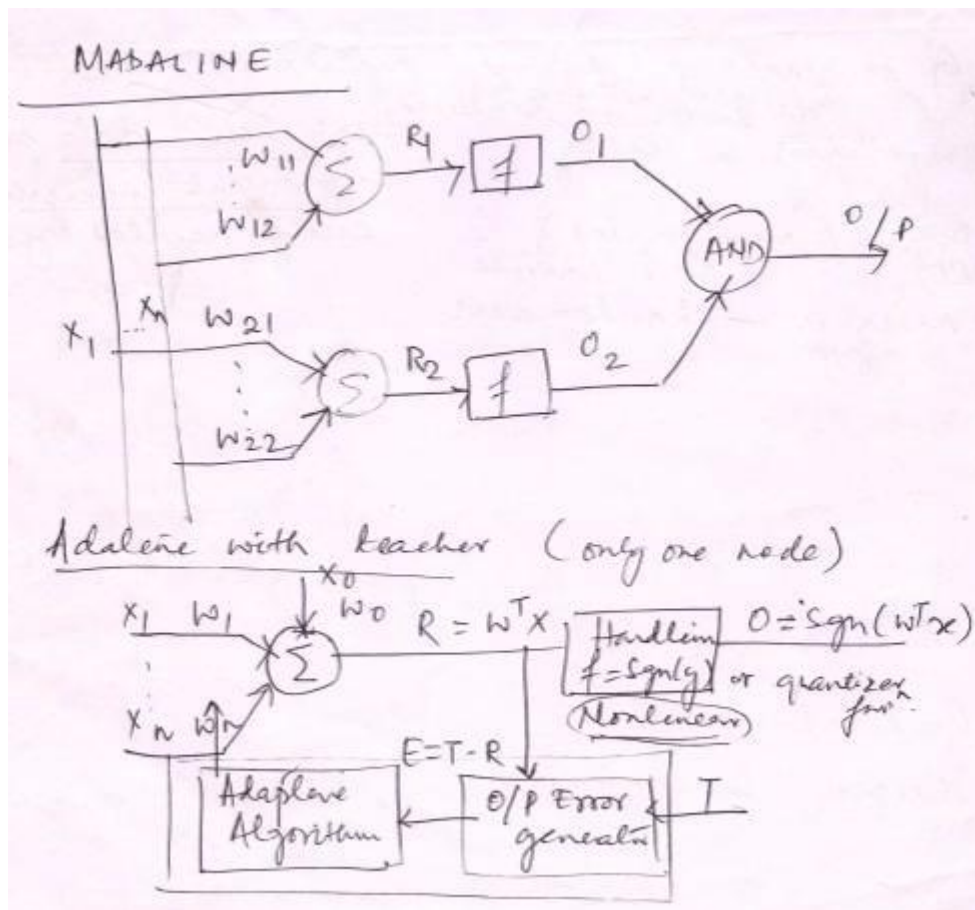


Fig. MADALINE network

## LECTURE-5

### The Multi-layered Perceptron training:

Improvements over Perceptron:

- 1) Smooth nonlinearity - sigmoid
- 2) 1 or more hidden layers

### Training the hidden layer:

Not obvious how to train the hidden layer parameters.

The error term is meaningful only to the weights connected to the output layer. How to adjust hidden layer connections so as to reduce output error? – *credit assignment* problem.

Any connection can be adapted by taking a full partial derivative over the error function, but then to update a single weight in the first stage we need information about distant neurons/connections close to the output layer (locality rule is violated). In a large network with many layers, this implies that information is exchanged over distant elements of the network though they are not directly connected. Such an algorithm may be mathematically valid, but is biologically unrealistic.

### The Backpropagation Algorithm:



As in Perceptron, this training algorithm involves 2 passes:  
 The forward pass – outputs of various layers are computed  
 The backward pass – weight corrections are computed  
 Consider a simple 3-layer network with a single neuron in each layer.

Total output error over all patterns:  $E = \sum_p E_p$

Squared Output error for the p'th pattern:  $E_p = \frac{1}{2} \sum_i e_i^2$

Output error for the p'th pattern:  $e_i = d - y_i$

Network output:  $y_i = g(h_i^s)$

Net input to the output layer:  $h_i^s = \sum_j w_{ij}^s V_j - \theta_i^s$

Output of the hidden layer:  $V_j^f = g(h_j^f)$

Net input of the hidden layer:  $h_j^f = \sum_k w_{jk}^f x_k - \theta_j^f$

Update rule for the weights using gradient descent:

$$\Delta w_{ij}^s = -\eta \frac{\partial E_p}{\partial w_{ij}^s}; \quad \Delta \theta_i^s = -\eta \frac{\partial E_p}{\partial \theta_i^s}$$

$$\Delta w_{jk}^f = -\eta \frac{\partial E_p}{\partial w_{jk}^f}; \quad \Delta \theta_j^f = -\eta \frac{\partial E_p}{\partial \theta_j^f}$$

Updating  $w_{ij}^s$ :

$$\frac{\partial E_p}{\partial w_{ij}^s} = \frac{\partial E_p}{\partial e_i} \frac{\partial e_i}{\partial y_i} \frac{\partial y_i}{\partial h_i^s} \frac{\partial h_i^s}{\partial w_{ij}^s}$$

$$= e(-1)g'(h_i^s)V_j$$



The delta at the output layer,  $\delta_i^s$ , is defined as,

$$\delta_i^s = e_i g'(h_i^s) \quad (6.2.2.1.11)$$

Therefore,

$$\Delta w_{ij}^s = -\eta \frac{\partial E_p}{\partial w_{ij}^s} = \eta \delta_i^s V_j \quad (6.2.2.1.12)$$

By similar arguments, it can be easily be shown that the update rule for the threshold term is,

$$\Delta \theta_i^s = -\eta \delta_i^s \quad (6.2.2.1.13)$$

Updating  $w_{jk}^f$ :

$$\begin{aligned} \Delta w_{jk}^f &= -\eta \frac{\partial E_p}{\partial w_{jk}^f} \\ \frac{\partial E_p}{\partial w_{jk}^f} &= \sum_i \frac{\partial E_p}{\partial e_i} \frac{\partial e_i}{\partial y_i} \frac{\partial y_i}{\partial h_i^s} \frac{\partial h_i^s}{\partial V_j} \frac{\partial V_j}{\partial w_{jk}^f} \\ &= \sum_i e_i (-1) g'(h_i^s) w_{ij}^s \frac{\partial V_j}{\partial w_{jk}^f} \\ &= \sum_i e_i (-1) g'(h_i^s) w_{ij}^s g'(h_j^f) x_k \\ &= \sum_i \delta_i^s w_{ij}^s g'(h_j^f) x_k \end{aligned} \quad (6.2.2.1.14)$$

Define an error term at the hidden layer as,

$$\delta_j^f = \sum_i \delta_i^s w_{ij}^s g'(h_j^f) \quad (6.2.2.1.15)$$

Therefore,

$$\Delta w_{jk}^f = -\eta \delta_j^f x_k \quad (6.2.2.1.16)$$

Similarly the update rule for the threshold term is,

$$\Delta \theta_j^f = -\eta \delta_j^f \quad (6.2.2.1.17)$$

**weight correction = (learning rate) \* (local  $\delta$  from ‘top’) \* (activation from ‘bottom’)**

### General formulation of the Backpropagation Algorithm:

Notation:

Input (at k'th input neuron)	-	$x_k$
Actual Output (at i'th output neuron)	-	$y_i$
Target output (at i'th output neuron)	-	$d_i$
Hidden neuron activation	-	$V_j^l$ (of j'th neuron in l'th layer)
Layer number	-	$l=0$ (input layer) to $L$ (output layer)
Net input	-	$h_{jl}$ (for j'th neuron in l'th layer)
$g(h)$	-	sigmoid nonlinearity $= 1/(1+\exp(-\beta h))$

### Steps:

1. Initialize weights with small random values
2. (Loop over training data)
3. Choose a pattern and apply it to the input layer

$$V_k^0 = x_k^p \quad \text{for all } k.$$

4. Propagate the signal forwards thro' the network using:

$$V_j^l = g(h_j^l) = g\left(\sum_k w_{jk}^l V_k^{l-1} - b_j^l\right).$$

for each j, k and 'l' until final outputs  $V_i^L$  have all been calculated.

5. Compute errors,  $\delta$ 's, for the output layer.

$$\delta_i^L = g'(h_i^L)[d_i(p) - V_i^L]$$

6. Compute  $\delta$ 's for preceding layers by backpropagation of error:

$$\delta_i^{l-1} = g'(h_i^{l-1}) \left[ \sum_j w_{ji}^l \delta_j^l \right]$$

For  $l = L, L-1, \dots, 1$

7. Update weights using the following:

$$\Delta w_{ij}^l = \eta \delta_i^l V_j^{l-1};$$

$$\Delta \theta_i^l = -\eta \delta_i^l$$

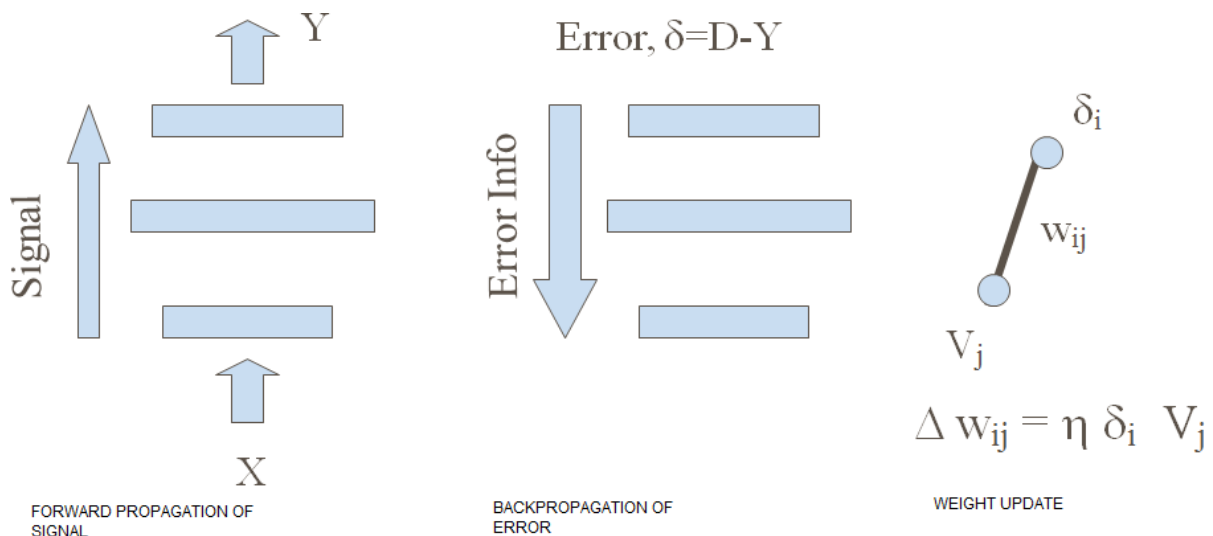


Fig. Training using Back propagation algorithm

#### Training:

Randomly initialize weights.

Train network using backprop eqns.

Stop training when error is sufficiently low and freeze the weights.

Testing

Start using the network.

#### Merits of MLP trained by BP:

- A general solution to a large class of problems.
- With sufficient number of hidden layer nodes, MLP can approximate arbitrary target functions.
- Backprop applies for arbitrary number of layers, partial connectivity (no loops).
- Training is local both in time and space – parallel implementation made easy.
- Hidden units act as “feature detectors.”
- Good when no model is available

#### Problems with MLP trained by BP:

- Blackbox approach
- Limits of generalization not clear
- Hard to incorporate prior knowledge of the model into the network
- slow training

e) local minima

## LECTURE-6

### Architectures of MLP:

If there is no nonlinearity then an MLP can be reduced to a linear neuron.

#### 1. Universal Approximator:

Theorem:

Let  $g(\cdot)$  be a nonconstant, bounded, and monotone-increasing continuous function. Let  $I_m$  denote the  $m$ -dimensional hypercube  $[0,1]^m$ . The space of continuous functions on  $I_m$  is denoted by  $C(I_m)$ . Then given any function  $f$  in  $C(I_m)$  and  $\varepsilon > 0$ , there exist an integer  $M$  and sets of real constants  $a_i$ ,  $b_i$  and  $w_{ij}$ , where  $i = 1, \dots, n$ , and  $j = 1, \dots, m$  such that we may define:

$$F(x_1, \dots, x_m) = \sum_{i=1}^n a_i g\left(\sum_{j=1}^m w_{ij} x_j + b_i\right)$$

As an approximate realization of the function  $f(\cdot)$ ; that is,

$$|F(x_1, \dots, x_m) - f(x_1, \dots, x_m)| < \varepsilon$$

for all  $x_1, \dots, x_m$  that lie in the input space.

For the above theorem to be valid, the sigmoid function  $g(\cdot)$  has to satisfy some conditions. It must be: 1) non-constant, 2) bounded, 3) monotone-increasing and 4) continuous.

All the four transfer functions described in the section on Perceptrons satisfy conditions #1,2 and 3. But the hardlimiting nonlinearities are not continuous. Therefore, the logistic function or the tanh function are suitable for use as sigmoids in MLPs.

#### 2. In general more layers/nodes greater network complexity

Although 3 hidden layers with full connectivity are enough to learn any function often more hidden layers and/or special architectures are used.

More hidden layers and/or hidden nodes:

#### 3-layer network:

Arbitrary continuous function over a finite domain

#### 4-layer network

Neurons in a 3-layer architecture tend to interact globally.

In a complex situation it is hard to improve the approximation at one point without worsening it at another.

So in a 4-layer architecture:

<sup>st</sup> 1 hidden layer nodes are combined to construct locally sensitive neurons in the second hidden layer.

#### Discontinuous functions:

learns discontinuous (inverse function of continuous function) functions also (Sontag, 1992)

#### For hard-limiting threshold functions:

<sup>st</sup> 1 hidden layer: semi-infinite regions separated by a hyper-plane

<sup>nd</sup> 2 hidden layer: convex regions

<sup>rd</sup> 3 hidden layer: non-convex regions also

#### Training MLP:

1. **Initialization:** is VERY important.

$g''(.)$  appears on the right side of all weight update rules (see sections 6.1.1, 6.1.2, 6.2.1). Note that  $g''(.)$  is high at the origin and falls on both sides. Therefore most learning happens when the net input ( $h$ ) to the neurons is close to 0. Hence it is desirable to make initial weights small. A general rule for initialization of input weights for a given neuron is:

$$\text{Mean}(w(0)) = 0.$$

$$\text{std}(w(0)) = \frac{1}{\sqrt{m}} \text{ where } m \text{ is the number of inputs going into a neuron.}$$

## 2. Batch mode and Sequential mode:

**Epoch:** presentation of all training patterns is called an epoch.

### Batch mode:

Updating network weights once every epoch is called batch mode update.

- memory intensive
- greater chance of getting stuck in local minima

### Sequential mode:

Updating the network weights after every presentation of a data point is sequential mode of update.

- lesser memory requirement
- The random order of presentation of input patterns acts as a noise source lesser chance of local minima

### Rate of learning:

We have already seen the tradeoffs involved in choice of a learning rate.

Small learning rate  $\eta$ , approximate original continuous domain equations more closely but slows down learning.

Large learning rate  $\eta$ , poorer approximation of original equations. Error may not decrease monotonically and may even oscillate. But learning is faster..

A good thumb rule for choosing  $\eta$ :

$$\eta = 1/m$$

Where „ $m$ “ is the number of inputs to a neuron. This rule assumes that there are different  $\eta$  s for different neurons.

## 3. Important tip relating learning rate and error surface:

Rough error surface, slow down, low  $\eta$

Smooth (flat) error surface, speed up, high  $\eta$

### i) Momentum:

a) If  $|a| < 1$ , the above time-series is convergent.

b) If the sign of the gradient remains the same over consecutive iterations the weighted sum  $\Delta w_{ji}$  grows exponentially i.e., accelerate when the terrain is clear.

c) If the gradient changes sign in consecutive iterations,  $\Delta w_{ji}$  shrinks in magnitude i.e., slow down when the terrain is rough.

i) Momentum:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

Action of momentum:

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^n \alpha^{n-t} \delta_j(t) y_i(t) = -\eta \sum_{t=0}^n \alpha^{n-t} \frac{\partial E(t)}{\partial w_{ji}(t)}$$

ii) Separate eta for each weight:

- a) Separate  $\eta$  for each weight
- b) Every eta varies with time
- c) If  $\delta(w)$  changes sign several time in the past few iters, decrease  $\eta$
- d) If  $\delta(w)$  doesn't change sign in the past few iters, increase  $\eta$

**Stopping Criteria:** when do we stop training?

- a) Error < a minimum.
- b) Rate of change in error averaged over an epoch < a minimum.
- c) Magnitude of gradient  $\|g(w)\| < \text{a minimum}$ .
- d) When performance over a test set has peaked.

**Premature Saturation:**

All the weight modification activity happens only when  $|h|$  is within certain limits.  
 $g''(h) \approx 0$ , or  $\delta(w) = 0$ , for large  $|h|$ .

**NN gets stuck in a shallow local minimum.**

**Solutions:**

- 1) - Keep a copy of weights
  - Retract to pre-saturation state
  - Perturb weights, decrease  $\eta$  and proceed
- 2) - Reduce sigmoid gain ( $\lambda$ ) initially
- e) Increase  $\lambda$  gradually as error is minimized

Network doesn't get stuck, but never settles either.

**Testing/generalization:**

Idea of overfitting or overtraining:

Using too many hidden nodes, may cause overtraining. The network might just learn noise and generalize poorly.

**Example of polynomial interpolation:**

Consider a data set generated from a quadratic function with noise added. A linear fit is likely to give a large error. Best fit is obtained with a quadratic function. Fit 10<sup>th</sup> degree might give a low error but is likely to learn the variations due to noise also. Such a fit is likely to do poorly on a test data set. This is called overfitting or poor generalization.

This happens because there are many ways of generalizing from a given training data set.

The above Venn diagram illustrates the possibility of generalizing in multiple ways from a given training data set. U is the universe of all possible input-output patterns. F (the ellipse)



represents the set of I/O pairs that define the function to be learnt by the mlp. T (circle) denotes the training data set which is a subset of F. X denotes the test data set. The dotted rectangle denotes the actual function learnt by the NN, which is consistent with the training set T, but is completely non-overlapping with the test set X, and very different from the unknown function F.

A simple calculation from (Hertz et al 1991).

Boolean function

N-inputs, 1-output

$2^N$  patterns and  $2^{2^N}$  rules totally.

Assume,

p – training patterns say, represent the rule T.

Then there are  $(2^N - p)$  test patterns and there are  $2^{(2^N - p)}$  rules, R, consistent with rule T.

$N = 30$ ,  $p = 1000$  patterns.

You have  $2^{10^9}$  generalizations exist for the same training set T.

### Applications of MLP

Three applications of MLPs that simulate aspects of sensory, motor or cognitive functions are described.

1. Nettalk
2. Past tense learning
3. Autonomous Land Vehicle in a Neural Network (ALVINN)

### LECTURE-6

### Multilayer Feed-Forward Network:

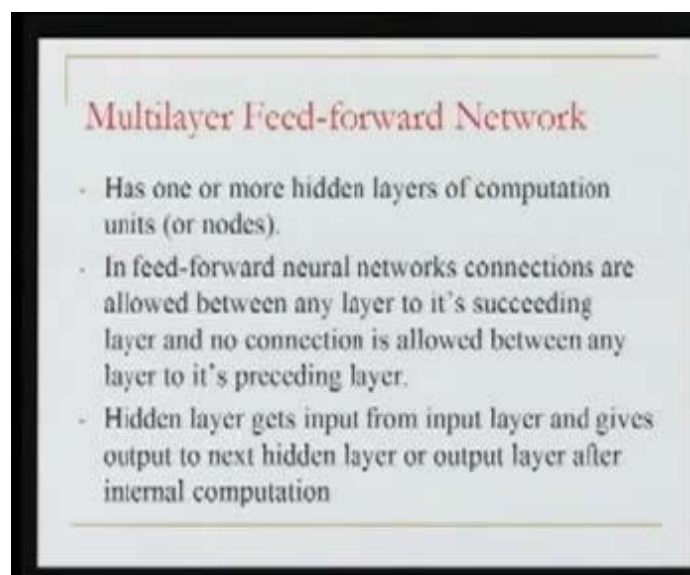


Fig. Characteristics of Multilayer feed-forward network



The algorithm that was derived using gradient descent for nonlinear neural networks with nonlinear activation function is popularly known as back propagation learning algorithm, although the learning algorithm still is derived using gradient descent rule.

Multilayer feed forward network has more hidden layers and again, when I say feed forward network, the connections are all allowed only from any layer to its succeeding layer, but the connections are not allowed from any layer to its preceding layer. The example is you see here there are four layers. These are all inputs. First hidden layer, second hidden layer, third hidden layer and this is output layer. When we say the number of layers, we do not count the input layer as one of the layers. When I say two layered network, then I have only one hidden layer and next layer becomes output layer.

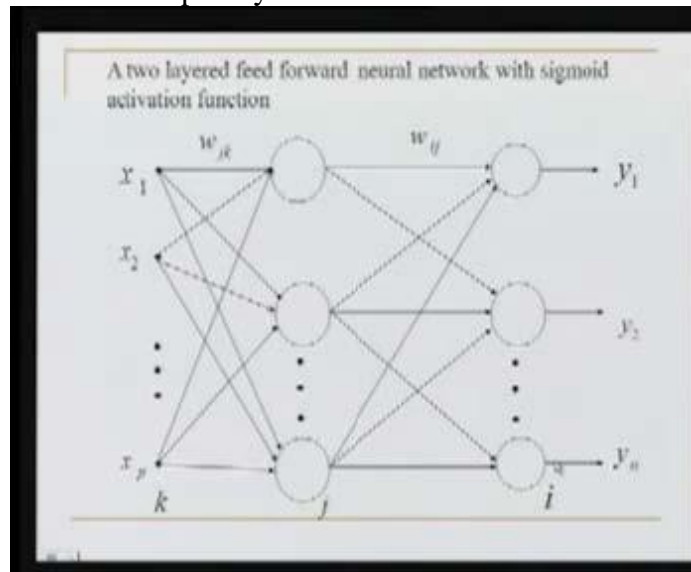


Fig. Multilayer feed foreward network

This particular configuration means there are sub-units, sub-neurons here and this particular configuration, if I connect you will see why I say feed forward network, because I am able to connect any layer from its preceding layer. That means connections are allowed from the preceding layer to any layer, but cannot allow the feedback connection. (Refer Slide Time: 30:54) This is called feedback connection; this is not allowed. This is allowed. From this layer, I can connect to this layer. This is allowed, but I cannot allow from this layer to connect to this layer. These are called feedback connections. They are not allowed and that is why this is known as feed forward network.

Today, we will derive a two-layered feed forward neural network with sigmoid activation function. We can very easily see that this is 1 layer; this is the only hidden layer and this is the only output layer; output layer is always only one.

We have a certain convention that we will put while deriving a back propagation learning algorithm for this. The same simple principle; given training data, we allow the input to pass through the network, compute the error here, use the gradient descent rule and the back propagated error are used to modify the weights here that is between output layer and hidden layer and again another form of back propagated error here has to be used for modification of the weights between input layer and hidden layer. This is again the convention that we will use.

## Derivation of Back propagation algorithm

Compute the response

$$v_j = \frac{1}{1 + e^{-h_j}}, \quad h_j = \sum w_{jk} x_k$$

$$y_i = \frac{1}{1 + e^{-s_i}}, \quad s_i = \sum w_{ij} v_j$$

Compute the cost function (Instantaneous update)

$$E = \frac{1}{2} \sum_i (y_i^e - y_i)^2$$

9

## The gradient descent rule

Weight update rule for weights between the hidden layer and the output layer

$$w_{ij}(t+1) = w_{ij}(t) - \eta \frac{\partial E}{\partial w_{ij}}$$

Weight update rule for weights between the input layer and the hidden layer

$$w_{jk}(t+1) = w_{jk}(t) - \eta \frac{\partial E}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{ij}} \text{ and } \frac{\partial E}{\partial w_{jk}} \text{ have to be derived}$$

Fig. The Gradient descent rule

After choosing the weights of the network randomly, the backpropagation algorithm is used to compute the necessary corrections. The algorithm can be decomposed in the following four steps:

- i) Feed-forward computation
- ii) Backpropagation to the output layer
- iii) Backpropagation to the hidden layer
- iv) Weight updates

The algorithm is stopped when the value of the error function has become sufficiently small.

In the case of  $p > 1$  input-output patterns, an extended network is used to compute the error function for each of them separately. The weight corrections The Backpropagation Algorithm are computed for each pattern and so we get, for example, for weight  $w_{ij}^{(1)}$  the corrections

$$\Delta_1 w_{ij}^{(1)}, \Delta_2 w_{ij}^{(1)}, \dots, \Delta_p w_{ij}^{(1)}$$

The necessary update in the gradient direction is then

$$\Delta w_{ij}^{(1)} = \Delta_1 w_{ij}^{(1)} + \Delta_2 w_{ij}^{(1)} + \dots + \Delta_p w_{ij}^{(1)}$$

We speak of batch or off-line updates when the weight corrections are made in this way. Often, however, the weight updates are made sequentially after each pattern presentation (this is called on-line training). In this case the corrections do not exactly follow the negative gradient direction, but if the training patterns are selected randomly the search direction oscillates around the exact gradient direction and, on average, the algorithm implements a form of descent in the error function. The rationale for using on-line training is that adding some noise to the gradient direction can help to avoid falling into shallow local minima of the error function. Also, when the training set consists of thousands of training patterns, it is very expensive to compute the exact gradient direction since each epoch (one round of presentation of all patterns to the network) consists of many feed-forward passes and on-line training becomes more efficient.

### **Back Propagation Neural Network**

Backpropagation is a training method used for a multi layer neural network. It is also called the generalized delta rule. It is a gradient descent method which minimizes the total squared error of the output computed by the net. Any neural network is expected to respond correctly to the input patterns that are used for training which is termed as memorization and it should respond reasonably to input that is similar to but not the same as the samples used for training which is called generalization. The training of a neural network by back propagation takes place in three stages 1. Feedforward of the input pattern 2. Calculation and Back propagation of the associated error 3. Adjustments of the weights After the neural network is trained, the neural network has to compute the feedforward phase only. Even if the training is slow, the trained net can produce its output immediately.

#### **Architecture**

A multi layer neural network with one layer of hidden units is shown in the figure. The output units and the hidden units can have biases. These bias terms are like weights on connections from units whose output is always 1. During feedforward the signals flow in the forward direction i.e. from input unit to hidden unit and finally to the output unit. During back propagation phase of learning, the signals flow in the reverse direction.

#### **Algorithm**

The training involves three stages 1. Feedforward of the input training pattern 2. Back propagation of the associated error 3. Adjustments of the weights. During feedforward, each input unit ( $X_i$ ) receives an input signal and sends this signal to each of the hidden units  $Z_1, Z_2, \dots, Z_n$ . Each hidden unit computes its activation and sends its signal to each output unit. Each output unit computes its activation to compute the output or the response of the neural net for the given input pattern.

During training, each output unit compares its computed activation  $y_k$ , with its target value  $t_k$  to determine the associated error for the particular pattern. Based on this error the factor  $\delta_k$  for all  $m$  values are computed. This computed  $\delta_k$  is used to propagate the error at the output unit  $Y_k$  back to all units in the hidden layer. At a later stage it is also used for updation of weights between the output and the hidden layer. In the same way  $\delta_j$  for all  $p$  values are computed for each hidden unit  $Z_j$ . The values of  $\delta_j$  are not sent back to the input units but are used to update the weights between the hidden layer and the input layer. Once all the  $\delta$  factors are known, the weights for all layers are changed simultaneously. The adjustment to all weights  $w_{jk}$  is based on the factor  $\delta_k$  and the activation  $z_j$  of the hidden unit  $Z_j$ . The change in weight to the connection between the input layer and the hidden layer is based on  $\delta_j$  and the activation  $x_i$  of the input unit.

### Activation Function

An activation function for a back propagation net should have important characteristics. It should be continuous, Differentiable and monotonically non- decreasing. For computational efficiency, it is better if the derivative is easy to calculate. For the commonly used activation function, the derivative can be expressed in terms of the value of the function itself. The function is expected to saturate asymptotically. The commonly used activation function is the binary sigmoidal function.

### Training Algorithm

The activation function used for a back propagation neural network can be either a bipolar sigmoid or a binary sigmoid. The form of data plays an important role in choosing the type of the activation function. Because of the relationship between the value of the function and its derivative, additional evaluations of exponential functions are not required to be computed.

#### Algorithm

Step 0: Initialize weights

Step 1: While stopping condition is false, do steps 2 to 9

Step 2: For each training pair, do steps 3 - 8 *Feed forward*

Step 3: Input unit receives input signal and propagates it to all units in the hidden layer

Step 4: Each hidden unit sums its weighted input signals

Step 5: Each output unit sums its weighted input signals and applied its activation function to compute its output signal.

*Backpropagation* Step 6: Each output unit receives a target pattern corresponding to the input training pattern, computes its error information term  $\delta_k = (t_k - y_k) f'(y_{ink})$  Calculates its bias correction term  $\Delta W_{ok} = \alpha \delta_k$  And sends  $\delta_k$  to units in the layer below

Step 7: Each hidden unit sums its delta inputs Multiplies by the derivative of its activation function to calculate its error information term Calculates its weight correction term  $\Delta v_{ij} = \alpha \delta_j x_i$  And calculates its bias correction term  $\Delta v_{oj} = \alpha \delta_j$  *Update weights and biases*

Step 8: Each output unit updates its bias and weights  $W_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}$  Each hidden unit updates its bias and weights  $V_{ij}(\text{new}) = v_{ij}(\text{old}) + \Delta v_{ij}$

Step9:Test stopping condition

## LECTURE-7

### Radial Basis Function Networks:

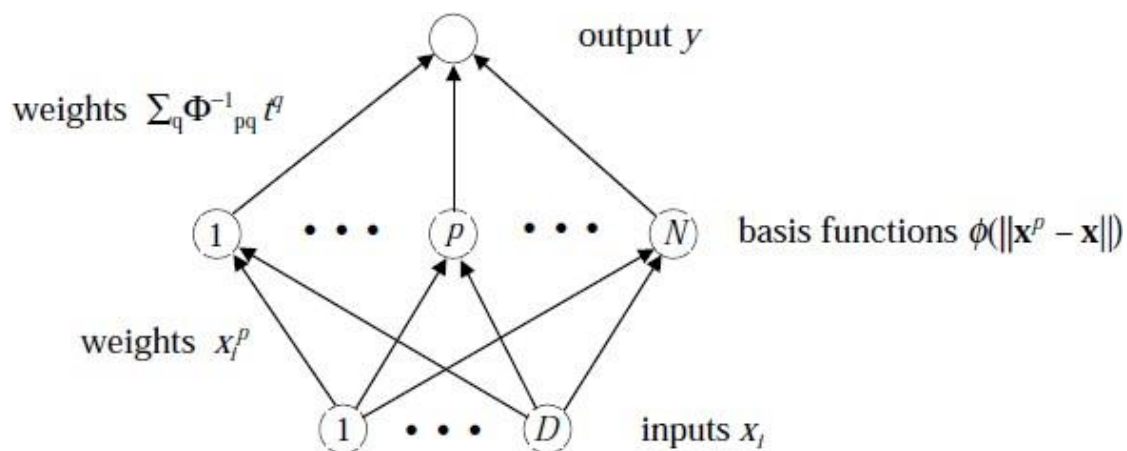


Fig. RBF network

- These are 3-layer networks that can approximate any continuous function through a basis function expansion.
- The basis functions here (which are data dependent as earlier) exhibit some radial symmetry.
- These networks have the so called perfect interpolation property.

The function represented by an RBF network with  $p$  hidden nodes can be written as

$$y = \sum_{j=1}^p w_j \phi(\|X - \theta_j\|),$$

$X$  is the input to the network.

- $w_j$  is weight from  $j^{\text{th}}$  hidden node to the output.
  - $\phi(\|X - \theta_j\|)$  is the output of the  $j^{\text{th}}$  hidden node
- and  $\theta_j$  is the parameter vector associated with  $j^{\text{th}}$

hidden node,  $j = 1, \dots, p$ .

A very popular model is the Gaussian RBF network.

- Here the output is written as

$$y = \sum_{j=1}^p w_j \exp\left(-\frac{\|X - \theta_j\|^2}{2\sigma^2}\right)$$

- The  $\theta_j$  is called the center of the  $j^{\text{th}}$  hidden or RBF node and  $\sigma$  is called the width.
- We can have different  $\sigma$  for different hidden nodes.

We next consider learning the parameters of a RBF network from training samples.

- Let  $\{(X_i, d_i), i = 1, \dots, N\}$  be the training set.
- Suppose we are using the Gaussian RBF.
- Then we need to learn the centers ( $\theta_j$ ) and widths ( $\sigma$ ) of the hidden nodes and the weights into the output node ( $w_j$ ).

Like earlier, we can find parameters to minimize empirical risk under squared error loss function.

- Same as minimizing sum of squares of errors. Let

$$J = \sum_{i=1}^N \left( \sum_{j=1}^p w_j \exp\left(-\frac{\|X^i - \theta_j\|^2}{2\sigma^2}\right) - d^i \right)^2$$

$J$  is a function of  $\sigma$ ,  $w_j$ ,  $\theta_j$ ,  $j = 1, \dots, p$ .

We can find the weights/parameters of the network to minimize  $J$ .

- To minimize  $J$ , we can use the standard iterative algorithm of gradient descent.
- This needs computation of gradient which can be done directly from the expression for  $J$ .

- For this network structure there are no special methods to evaluate all the needed partial derivatives. Such a gradient descent algorithm is certainly one method of learning an RBF network from given training data.
- This is a general-purpose method for learning an RBF network.
- Like in the earlier case, we have to fix  $p$ , the number of hidden nodes.
- Such procedure would have the usual problems of converging to a local minimum of the error function.
- There are also other methods of learning an RBF network.
- If we have the basis functions,  $\phi_j$ , then it is exactly same as a linear model and we can use standard linear least squares method to learn  $w_j$ .
- To fix  $\phi_j$ , we need to essentially fix  $\theta_j$  (and may be  $\sigma$ ).
- So, if we can somehow fix centers and widths of the RBF nodes, then we can learn the  $w_j$  very easily.

As we have discussed earlier, these RBF networks use „local“ representations.

- What this means is that  $\theta_j$  should be „representative“ points of the feature space and they

should „cover“ the feature space.

- Essentially, the proof that these networks can represent any continuous function is based on having such centers for RBF nodes.
- We can use such ideas to formulate methods for fixing centers of RBF nodes.

One simple method of choosing centers,  $\theta_j$ , is to randomly choose  $p$  of the training examples.

- We know that with  $N$  hidden nodes and centers same as training examples, we get perfect interpolation.
- Hence we can take some of the training examples as centers.
- There can be some variations on this theme.
- However, such a method does not, in general, ensure that we have representative points in the feature space as centers.

When we have  $p$  hidden nodes, we need  $p$  „centers“.

- Hence we are looking for  $p$  number of „representative“ points in the feature space.
- The only information we have are the  $N$  training examples.
- Hence the problem is:

given  $N$  points,  $X_i, i = 1, \dots, N$  in  $\mathbb{R}^m$ , find  $p$  „representative“ points in  $\mathbb{R}^m$ .

- This is the „clustering problem“ This is a problem of forming the data into  $p$  clusters.
- We can take the „cluster centers“ to be the representative points.
- The kind of clusters we get depends on how we want to formalize the notion of the  $p$  points being representative of the  $N$  data points.
- We now look at one notion of clustering that is popular.

Let  $\theta_1, \dots, \theta_p$  represent the  $p$  cluster centers.

- Now we need an objective function that specifies how representative these are of the data  $X_i, i = 1, \dots, N$ .

Now we can define a cost function as

$$J = \sum_{j=1}^p \sum_{X^i \in S_j} ||X^i - \mu_j||^2$$

- The  $J$  is a function of  $\theta_j, j = 1, \dots, p$ . (Note that  $S_j$  are also functions of the  $\theta_j$ ’s).



- For a given set of centers,  $\{\theta_j\}$ ,  $J$  gives us the total error in approximating each of the training data by its nearest cluster center.
  - Hence we want to choose centers to minimize  $J$ . We now discuss a simple algorithm to find centers to minimize  $J$ .
  - This is known as K-means clustering algorithm.  
(Originally proposed by Lloyd in the context of vector quantization).
  - We are given  $N$  data points,  $X_i, i = 1, \dots, N$ .
- We want to find  $p$  cluster centers  $\theta_j, j = 1, \dots, p$ , to minimize  $J$ .

• We first rewrite  $J$  in a different form to motivate our algorithm.  
We think of the problem as finding the centers  $\theta_1, \dots, \theta_p$  and assigning  $X_i$  to these clusters.

- Let  $\mu_j, n = 1, \dots, N, j = 1, \dots, p$  be indicators of the cluster assignment.
- That is, if we assign  $X^n$  to cluster  $j$ , then we would have  $\rho_{nj} = 1$  and  $\rho_{np} = 0$ ,
- Now we can rewrite  $J$  as

$$J = \sum_{n=1}^N \sum_{j=1}^p \rho_{nj} \|X^n - \mu_j\|^2$$

- Since we are taking the Euclidian norm, we have

$$J = \sum_{n=1}^N \sum_{j=1}^p \rho_{nj} (X^n - \mu_j)^T (X^n - \mu_j)$$

- Note (for later use) that gradient of  $J$  w.r.t.  $\mu_j$  is

$$2 \sum_{n=1}^N \rho_{nj} (X^n - \mu_j)$$

We now have to find a way of minimizing  $J$  wrt all  $\rho_{nj}$  and  $\mu_j$ .

- First consider finding optimal values for  $\rho_{nk}$  with all the  $\mu_j$  fixed.
- We know that the variables  $\{\rho_{nk}\}$  have to satisfy  $\rho_{nk} \in \{0, 1\}$  and  $\sum_k \rho_{nk} = 1, \forall n$ . Recall

$$J = \sum_{n=1}^N \sum_{j=1}^p \rho_{nj} \|X^n - \mu_j\|^2$$

- Given the form of  $J$ , we can decouple optimization with respect to  $\rho_{nj}$  for different  $n$ .



- For a specific  $n$  we need to find  $\rho_{nk}$ ,  $k = 1, \dots, p$ , to minimize

$$\sum_{j=1}^p \rho_{nj} \|X^n - \mu_j\|^2$$

- Given the constraints on  $\rho_{nj}$ , the optimum would be

$$\begin{aligned} \rho_{nj} &= 1 && \text{if } \|X^n - \mu_j\|^2 \leq \|X^n - \mu_k\|^2, \forall k \\ &= 0 && \text{otherwise} \end{aligned}$$

- Now fixing the  $\rho_{nj}$ , consider optimizing  $J$  w.r.t.  $\mu_j$ .
- For a specific  $j$ , equating the gradient of  $J$  w.r.t.  $\mu_j$  to zero, we get

$$\sum_{n=1}^N \rho_{nj} (X^n - \mu_j) = 0$$

- This gives us the optimum value for  $\mu_j$  as

$$\mu_j = \frac{\sum_{n=1}^N \rho_{nj} X^n}{\sum_{n=1}^N \rho_{nj}}$$

---

Note that for a given  $n$ ,  $\rho_{nj}$  is 1 for exactly one  $j$  (and it is zero otherwise).

- Thus the  $\mu_j$  would be the mean of all data vectors assigned to the  $j^{\text{th}}$  cluster.
- This is the reason for the name K-means clustering.
- What we derived are optimum values for  $\rho_{nj}$  keeping  $\mu_j$  fixed and optimum values for  $\mu_j$  keeping  $\rho_{nj}$  fixed.
- Hence, in an algorithm we do this repeatedly.
- This is like the EM algorithm.

## Commonly Used Radial Basis Functions

A range of theoretical and empirical studies have indicated that many properties of the interpolating function are relatively insensitive to the precise form of the basis functions  $\phi(r)$ . Some of the most commonly used basis functions are:

### 1. Gaussian Functions:

$$\phi(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right) \quad \text{width parameter } \sigma > 0$$

### 2. Multi-Quadric Functions:

$$\phi(r) = (r^2 + \sigma^2)^{1/2} \quad \text{parameter } \sigma > 0$$

### 3. Generalized Multi-Quadric Functions:

$$\phi(r) = (r^2 + \sigma^2)^\beta \quad \text{parameters } \sigma > 0, 1 > \beta > 0$$

### 4. Inverse Multi-Quadric Functions:

$$\phi(r) = (r^2 + \sigma^2)^{-1/2} \quad \text{parameter } \sigma > 0$$

### 5. Generalized Inverse Multi-Quadric Functions:

$$\phi(r) = (r^2 + \sigma^2)^{-\alpha} \quad \text{parameters } \sigma > 0, \alpha > 0$$

### 6. Thin Plate Spline Function:

$$\phi(r) = r^2 \ln(r)$$

### 7. Cubic Function:

$$\phi(r) = r^3$$

### 8. Linear Function:

$$\phi(r) = r$$

### Experiences or learning:

Learning algorithms use experiences in the form of perceptions or perception action pairs to improve their performance. The nature of experiences available varies with applications. Some common situations are described below.

**Supervised learning:** In supervised learning a teacher or oracle is available which provides the desired action corresponding to a perception. A set of perception action pair provides what is called a training set. Examples include an automated vehicle where a set of vision inputs and the corresponding steering actions are available to the learner.

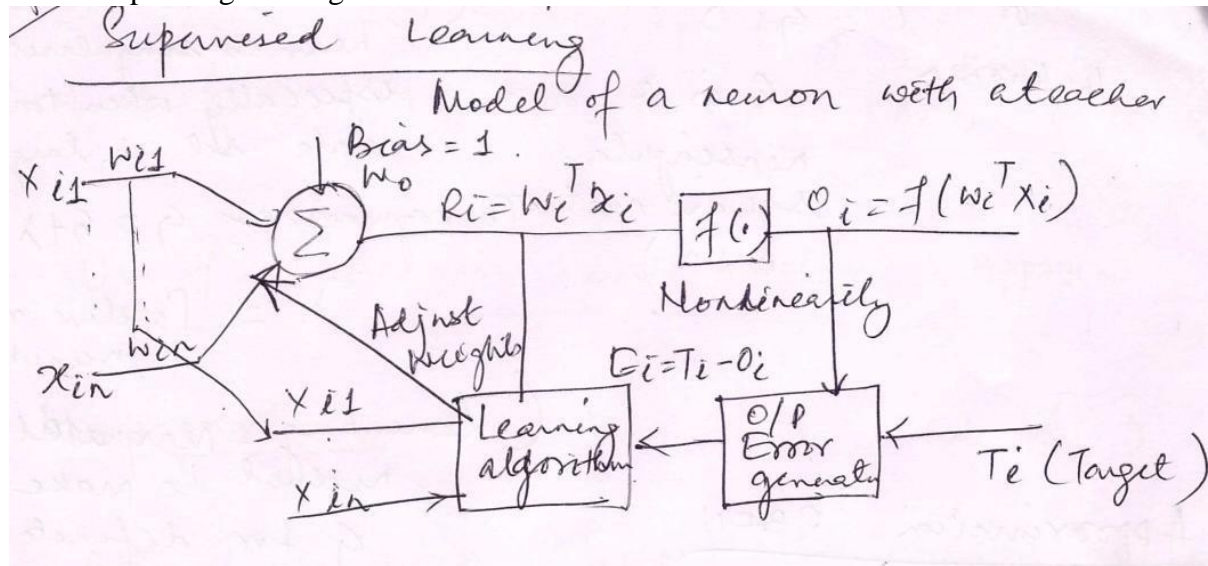


Fig. Supervised learning

**Unsupervised learning:** In unsupervised learning no teacher is available. The learner only discovers persistent patterns in the data consisting of a collection of perceptions. This is also called exploratory learning. Finding out malicious network attacks from a sequence of anomalous data packets is an example of unsupervised learning.

**Active learning:** Here not only a teacher is available, the learner has the freedom to ask the teacher for suitable perception-action example pairs which will help the learner to improve its performance. Consider a news recommender system which tries to learn an users preferences and categorize news articles as interesting or uninteresting to the user. The system may present a particular article (of which it is not sure) to the user and ask whether it is interesting or not.

**Reinforcement learning:** In reinforcement learning a teacher is available, but the teacher instead of directly providing the desired action corresponding to a perception, return reward and punishment to the learner for its action corresponding to a perception. Examples include a robot in a unknown terrain where its get a punishment when its hits an obstacle and reward when it moves smoothly.

In order to design a learning system the designer has to make the following choices based on the application.

## LECTURE-9

### Unsupervised Learning in Neural Networks:

Unsupervised learning mechanisms differ from supervised learning in that there is no "teacher" to instruct the network.

### Competitive Learning:

Competitive learning is a form of unsupervised learning which performs clustering over the input data. In a competitive learning network with  $n$ -output neurons, each output neuron is associated with a cluster. When a data point from a cluster is presented to the network, only the neuron corresponding to that cluster responds, while all other neurons remain silent. The single neuron that responds is often called a “winner” and therefore a competitive learning network of the kind just described is also known as a “winner-take-all” network.

It is easiest to introduce CL mechanism as a slight variation of Hebb’s rule.

### Kohonen Self-organizing Map:

It is also known as Kohonen feature map or topology-preserving map or Kohonen Self-organizing .

Information is often represented spatially in the two-dimensional neuronal sheets in the brain, in both the cortex and subcortical structures. We have learnt about the somatosensory, motor and visual maps in the corresponding sensory cortices in the brain. A map, in its ordinary sense, denotes a two-dimensional representation of a real-world domain, such that nearby points in the domain are mapped onto nearby points in the map.

Due to this “adjacency-preserving” property, these maps are also called *topographic* maps.

Self-organizing maps (SOM) are models of the topographic maps of the brain, first proposed by Teuvo Kohonen.

The SOM model can be presented as an extension of the competitive learning model described in the previous section. It is constructed by adding a biologically-relevant feature that is not originally present in the competitive learning network.

A key property of the SOM is that nearby or similar inputs activate nearby neurons in the map. The competitive learning network does not have this property.

Consider a hypothetical competitive learning network with 3 output neurons. The input space is two-dimensional. The weight vectors  $w_1$ ,  $w_2$ ,  $w_3$  lie on a line as shown in Fig., with  $w_1$  in between  $w_2$  and  $w_3$ . Note that such an arrangement is possible since there is no relation between the spatial position of the weight vectors and their indices.

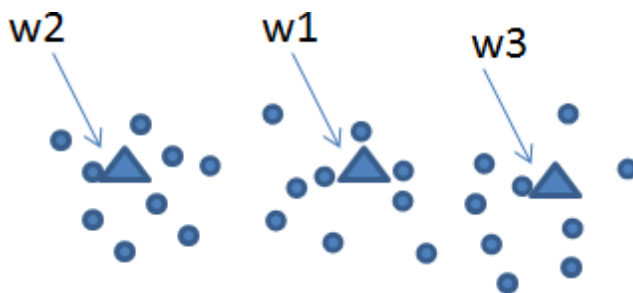


Fig. weight vectors and their indices when not related

The essence of the modification proposed in the SOM model, is a mechanism that ensures that the weight vectors remain spatially ordered, while they also move towards the data points that activate them maximally.

Unlike a competitive learning network, which consists of a single row of output neurons, a SOM consists of a  $m$ -dimensional grid of neurons. Usually two-dimensional SOMs are studied since SOMs were originally inspired by the two-dimensional maps in the brain. The topology of the grid is usually rectangular, though sometimes hexagonal topologies (Fig.) are also considered.

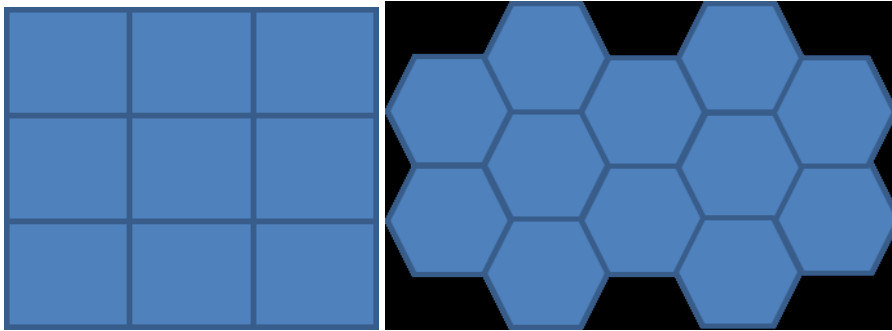


Figure: Rectangular and hexagonal trajectories of Kohonen's network

As in the case of competitive learning, the weight vector of the winner is moved towards the input,  $x$ . But addition, neurons close to the winner in the SOM are also moved towards the input,  $x$ , but with a lesser learning rate. Neurons that are nearby in the SOM are defined by a neighborhood  $N$ .

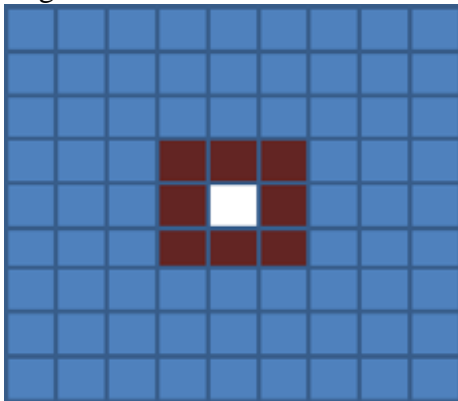


Fig. For the neuron in white (center) the neurons in red represent the neighborhood if we consider the neighborhood radius to be 1

Neighborhood size is large in the early stages, and is decreased gradually as training progresses.

### Learning Vector Quantization(LVQ):

Vector quantization is nothing but clustering, where Given a set of vectors  $\{x\}$ , find a set of representative vectors  $\{w_m; 1 \leq m \leq M\}$  such that each  $x$  is *quantized* into a particular  $w_m$ .  $\{w_m\}$  locate at the mean (centroid) of the density distribution of each cluster. LVQ is an unsupervised pattern classifier where the actual class membership information is not used.

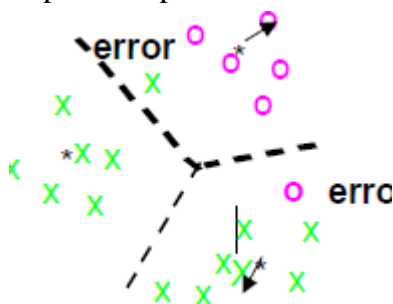


Fig. Clusters of data

## Applications of LVQ:

Speech Recognition

- Robot Arm control
- Industrial process control
- automated synthesis of digital systems
- channel equalization for telecommunication
- image compression
- radar classification of sea-ice
- optimization problems
- sentence understanding
- classification of insect courtship songs

## LECTURE-10

### Linear neuron model: (Hebbian Learning)

Hebb described a simple learning method of synaptic weight change. In Hebbian learning, when 2 cells have strong responses, and fire simultaneously, their connection strength or weight increases. The weight increase is proportional to the frequency at which they fire together.

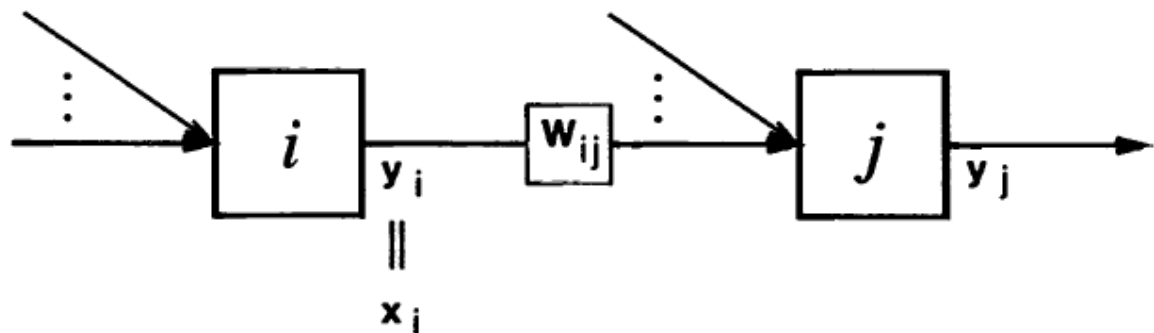


Fig. A simple network topology for Hebbian Learning, where  $W_{ij}$  resides between two neurons

$$\Delta w_{ij} = \eta f(\mathbf{w}_j^T \mathbf{x}) x_i$$

Where  $\eta$  is the learning rate,  $f(\cdot)$  is the neuron function,  $x$  is the input to the  $j$ th neuron.

Since the weights are adjusted according to the correlation formula is a type of correlational learning rule.

A sequence of learning patterns indexed by  $p$  is presented to the network. Initial weights are taken zero. So updated weight after entire data set is:

$$w_{ij} = \eta \sum_p y_{ip} y_{jp}$$

Frequent input patterns have more impact on weights, giving largest output at end.

The objective function is maximized to maximize output.

$$J = \frac{1}{2} \sum_p y_p^2 = \frac{1}{2} \sum_p (\mathbf{x}_p^T \mathbf{w})^2$$

This rule causes unconstrained growth of weights. Hebbian rule was modified by Oja by normalization.



### Modified Hebbian Learning:

$$w_i(t+1) = \frac{w_i(t) + \eta y(t) x_i(t)}{\sqrt{\sum_{i=1}^n [w_i(t) + \eta y(t) x_i(t)]^2}}$$

For small learning rate expanding in Taylor's series weight update rule becomes

$$\Delta w_i = \eta y x_i - \eta y^2 w_i = \eta y (x_i - y w_i)$$

Here, a weight decay proportional to the squared output is added to maintain weight vector unit length automatically.

### LECTURE-11

#### ANFIS: Adaptive Neuro-Fuzzy Inference Systems:

ANFIS are a class of adaptive networks that are functionally equivalent to fuzzy inference systems.

- ANFIS represent Sugeno & Tsukamoto fuzzymodels.
- ANFIS uses a hybrid learning algorithm

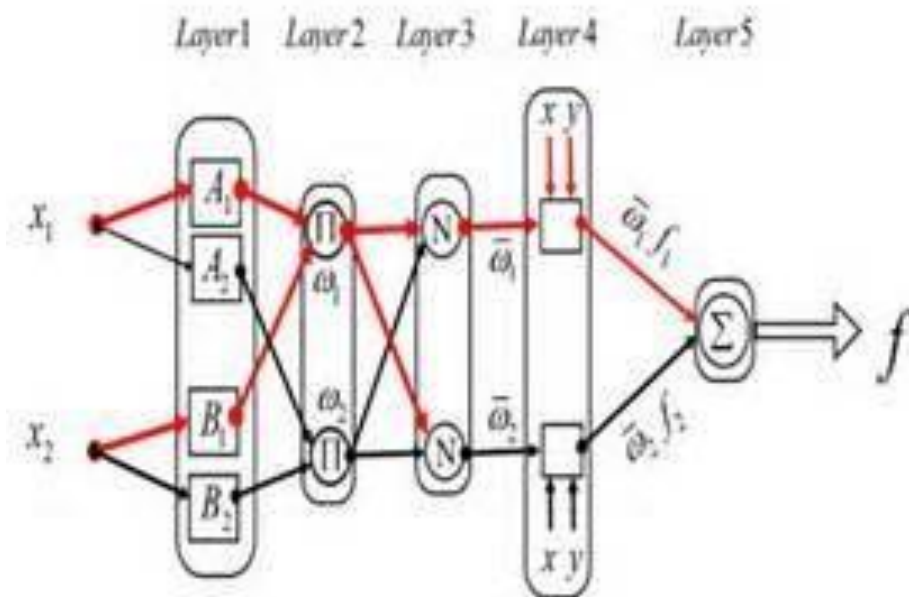


Fig. Architecture of ANFIS

$O_{1,i}$  is the output of the  $i^{\text{th}}$  node of the layer 1.

- Every node  $i$  in this layer is an adaptive node with a node function

$O_{1,i} = \mu_{A_i}(x)$  for  $i = 1, 2$ , or  $O_{1,i} = \mu_{B_{i-2}}(x)$  for  $i = 3, 4$

- $x$  (or  $y$ ) is the input node  $i$  and  $A_i$  (or  $B_{i-2}$ ) is a linguistic label associated with this node
- Therefore  $O_{1,i}$  is the membership grade of a fuzzy set ( $A_1, A_2, B_1, B_2$ ).

Typical membership function is Gaussian.

Every node in this layer is a fixed node labelled Prod.

- The output is the product of all the incoming signals.

$O_{2,i} = w_i = \mu_{A_i}(x) \cdot \mu_{B_i}(y)$ ,  $i = 1, 2$

- Each node represents the fire strength of the rule

- Any other T-norm operator that perform the AND operator can be used



Every node in this layer is a fixed node labelled Norm.

- The  $i$ th node calculates the ratio of the  $i$ th rule's firing strength to the sum of all rule's firing strengths.

$$O_{3,i} = w_i = \frac{w_i}{w_1 + w_2}, i = 1, 2$$

- Outputs are called normalized firing strengths.

Every node  $i$  in this layer is an adaptive node with a node function:

$$O_{4,i} = w_i f_i = w_i(p_i x + q_i y + r_i)$$

- $w_i$  is the normalized firing strength from layer 3.
- $\{p_i, q_i, r_i\}$  is the parameter set of this node.
- These are referred to as consequent parameters.

The single node in this layer is a fixed node labeled sum, which computes the overall output as the summation of all incoming signals:

$$\text{overall output} = O_{5,1} = \sum p_i w_i = \sum p_i w_i$$

### Hybrid Learning Algorithm:

The ANFIS can be trained by a hybrid learning algorithm presented by Jang in the chapter 8 of the book.

- In the forward pass the algorithm uses least-squares method to identify the consequent parameters on the layer 4.
- In the backward pass the errors are propagated backward and the premise parameters are updated by gradient descent.

	Forward Pass	Backward Pass
Premise Parameters	Fixed	Gradient Descent
Consequent Parameters	Least-squares estimator	Fixed
Signals	Node outputs	Error signals

Fig. Two passes in the hybrid learning algorithm for ANFIS.

Suppose that an adaptive network has  $L$  layers and the  $k$ th layer has  $\#(k)$  nodes.

- We can denote the node in the  $i$ th position of the  $k$ th layer by  $(k, i)$ .
- The node function is denoted by  $O_{ki}$ .
- Since the node output depends on its incoming signals and its parameter set  $(a, b, c)$ , we have

$$O_{ki}^k = O_{ki}^k(O_{i1}^{k-1}, \dots, O_{\#(k-1)}^{k-1}, a, b, c)$$

- Notice that  $O_{ki}$  is used as both node output and node function. Assume that a training data set has  $P$  entries.
- The error measure for the  $p$ th entry can be defined as the sum of the squared error

$$E_p = \sum_{m=1}^{\#(L)} (T_{m,p} - O_{m,p}^L)^2$$

$T_{m,p}$  is the  $m$ th component of the  $p$ th target.

- $O_{m,p}^L$  is the  $m$ th component the actual output vector.
- The overall error is

$$E = \sum_{p=1}^P E_p$$

In order to implement the gradient descent in E we calculate the error rate  $\frac{\partial E}{\partial O}$  for the pth training data for each node output O.

- The error rate for the output node at (L, i) is

$$\frac{\partial E_p}{\partial O_{i,p}^L} = -2(T_{i,p} - O_{i,p}^L)$$

For the internal node at (k, i), the error rate can be derived by the chain rule:

$$\frac{\partial E_p}{\partial O_{i,p}^k} = \sum_{m=1}^{\#(k+1)} \frac{\partial E_p}{\partial O_{m,p}^{k+1}} \frac{\partial O_{m,p}^{k+1}}{\partial O_{i,p}^k}$$

where  $1 \leq k \leq L - 1$

- The error rate of an internal node is a linear combination of the error rates of the nodes in the next layer.

Consider  $\alpha$  one of the parameters.

- Therefore

$$\frac{\partial E_p}{\partial \alpha} = \sum_{O^* \in S} \frac{\partial E_p}{\partial O^*} \frac{\partial O^*}{\partial \alpha}$$

where S is the set of nodes

- The derivative of the overall error with respect to  $\alpha$  is

$$\frac{\partial E}{\partial \alpha} = \sum_{p=1}^P \frac{\partial E_p}{\partial \alpha}$$

The update formula for  $\Delta \alpha$  is

$$\Delta \alpha = \eta \frac{\partial E}{\partial \alpha}$$

If the parameters are to be updated after each input-output pair (on-line training) then the update formula is:

$$\frac{\partial E_p}{\partial \alpha} = \sum_{O^* \in S} \frac{\partial E_p}{\partial O^*} \frac{\partial O^*}{\partial \alpha}$$

With the batch learning (off-line learning) the update formula is based on the derivative of the overall error with respect to  $\alpha$ :

$$\frac{\partial E}{\partial \alpha} = \sum_{p=1}^P \frac{\partial E_p}{\partial \alpha}$$

Problems of the gradient descent are:

The method is slow.

- It is likely to be trapped in local minima.

### Hybrid Learning Rule:

Combines:

- the gradient rule;
- the least squares estimate.

Consider that the adaptive network has only one output.

- output =  $F(I, S)$
- $I$  is the vector of input variables.
- $S$  is the set of parameters.
- $F$  is the function implemented by the ANFIS.
- If there exists a function  $H$  such that the composite function  $H \circ F$  is linear in some elements of  $S$  then these elements can be identified by LSM.

More formally, if the parameter set  $S$  can be decomposed into two sets  $S = S1 \oplus S2$  ( $\oplus$  direct sum), such that  $H \circ F$  is linear in the elements of  $S2$

- then applying  $H$  to output =  $F(I, S)$  we have  $H(\text{output}) = H \circ F(I, S)$  (7) which is linear in the elements of  $S2$ .
- Given values of elements of  $S1$ , it is possible to plug  $P$  training data in equation 7.
- As a result we obtain a matrix equation  $A_{-} = y$  where  $-$  is the unknown vector whose elements are parameters in  $S2$ .
- This is the standard linear least-square problem.

### Combining LSE and gradient descent:

#### - forward pass

In batch mode, each epoch is composed of a forward pass and a backward pass.

- In the forward pass an input vector is presented and the output is calculated creating a row in the matrices  $A$  and  $y$ .
- The process is repeated for all training data and the parameters  $S2$  are identified by BLS or RLS.
- After  $S2$  is identified the error for each pair is computed.

### Combining LSE and gradient descent:

#### - backward pass

The derivative of the error measure with respect to each node output propagate from the output toward the input.

- The derivatives are:

$$\frac{\partial E_p}{\partial O_{i,p}^L} = -2(T_{i,p} - O_{i,p}^L)$$

$$\frac{\partial E_p}{\partial O_{i,p}^k} = \sum_{m=1}^{\#(k+1)} \frac{\partial E_p}{\partial O_{m,p}^{k+1}} \frac{\partial O_{m,p}^{k+1}}{\partial O_{i,p}^k}$$

The parameters in  $S2$  are updated by the gradient method

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

### Applications of ANFIS:

1. Printed Character recognition
2. Inverse Kinematics
3. Nonlinear System identification
4. Channel Equalization

5.     Feed back control system
6.     Adaptive noise cancellation

