MACHINE LEARNING

Artificial Neural Networks (ANN)

Dr G.Kalyani
Department of Information Technology
Velagapudi Ramakrishna Siddhartha Engineering College

Topics

- **Motivation to ANN**
- **Perceptron**
- **Gradient Descent**
- **ANN with Backpropagation**

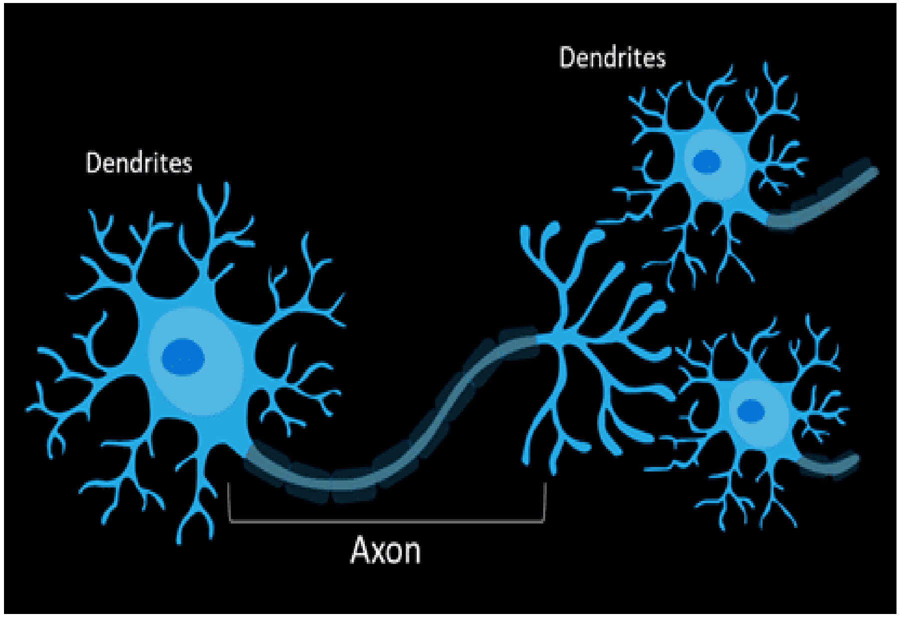
Introduction

- Neural network learning methods provide a robust approach to predict real-valued, discrete-valued, and vector-valued target functions.

- For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known.

Motivation to ANN

- The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems.
- To develop a feel for this analogy, let us consider a few facts from neurobiology.
  - The human brain, for example, is estimated to contain a densely interconnected network of approximately 10^{11} neurons, each connected, on average, to 10^{4} others.
  - Neuron activity is typically excited or inhibited through connections to other neurons.
  - The fastest neuron switching times are known to be on the order of 10^{-3} seconds--quite slow compared to computer switching speeds of 10^{-10} seconds.
  - Yet humans are able to make surprisingly complex decisions, surprisingly quickly.
  - For example, it requires approximately 10^{-1} seconds to visually recognize your mother.

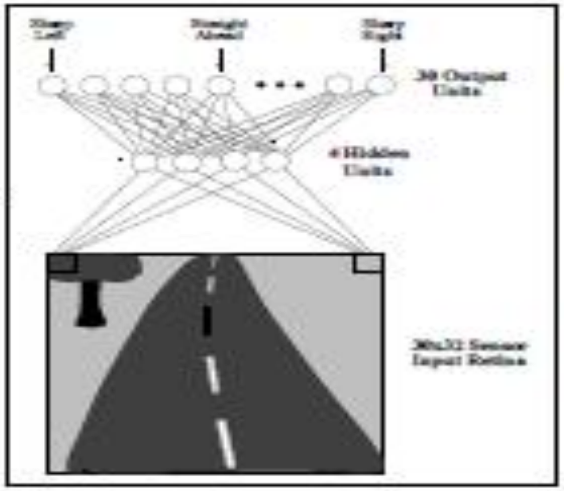


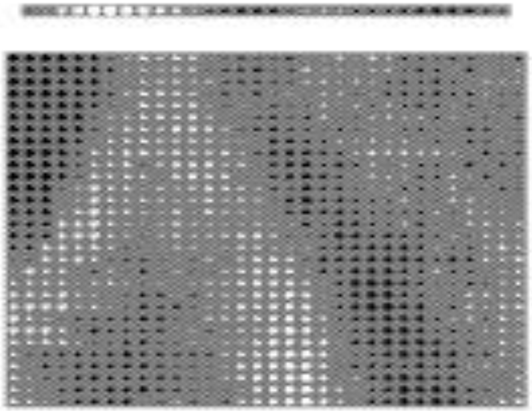
When to Consider Neural Networks

- Instances are represented by many attribute-value pairs.
- The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes
- The training examples may contain errors.
- Long training times are acceptable.
- Fast evaluation of the learned target function may be required.
- The ability of humans to understand the learned target function is not important.

ALVINN: An example ANN

ALVINN drives 70 mph on highways

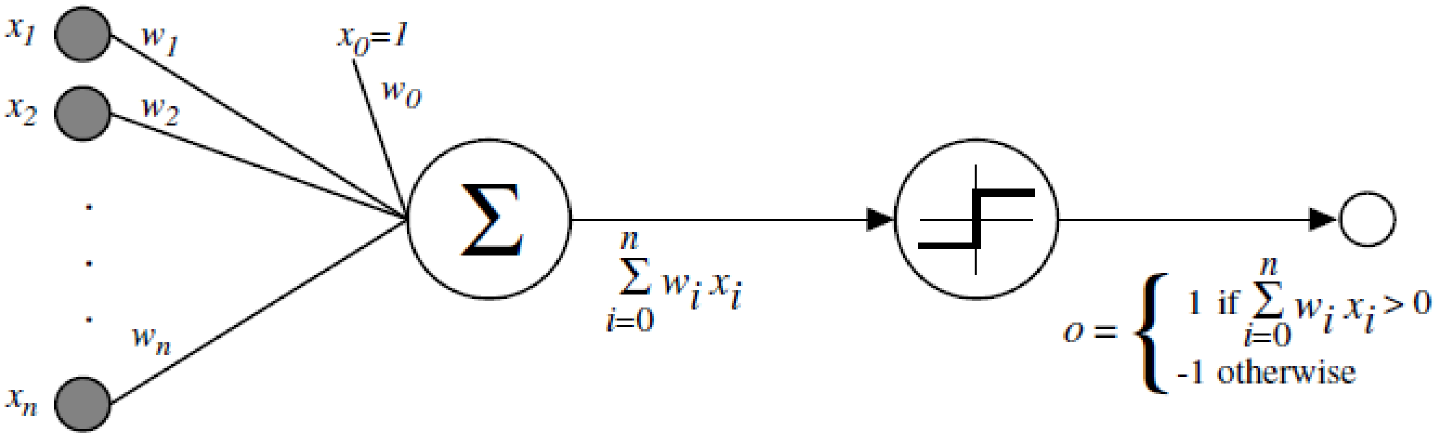




Topics

- **Motivation to ANN**
- **Perceptron**
- **Gradient Descent**
- **ANN with Backpropagation**

Perceptron



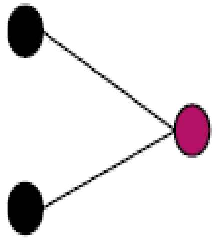
$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

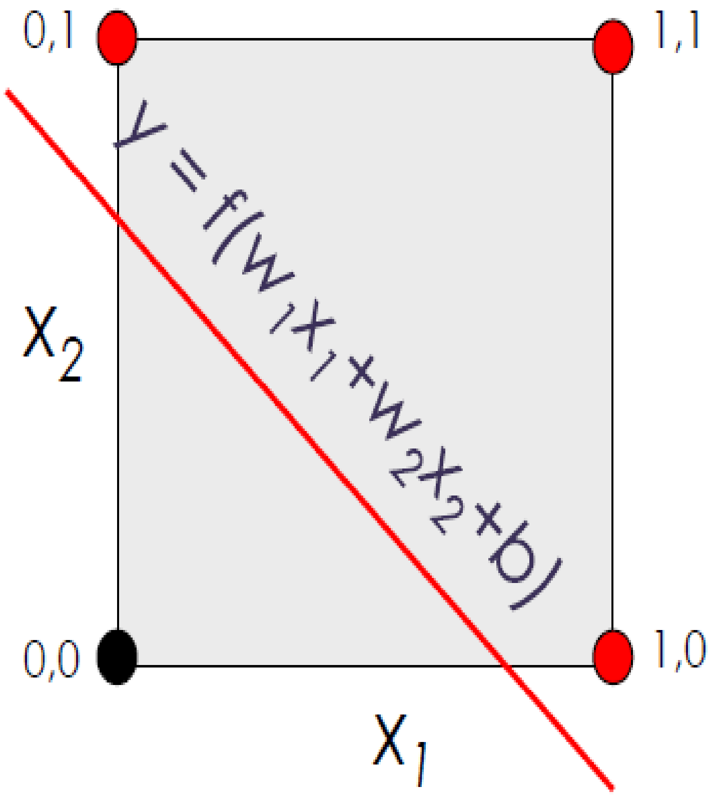
Perceptron Ex: Logical OR

- "w_1=1.0"
- "w_2=1.0"
- "b=-0.5"



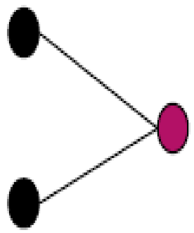
Logical OR Function

X_1	X_2	y
0	0	0
0	1	1
1	0	1
1	1	1



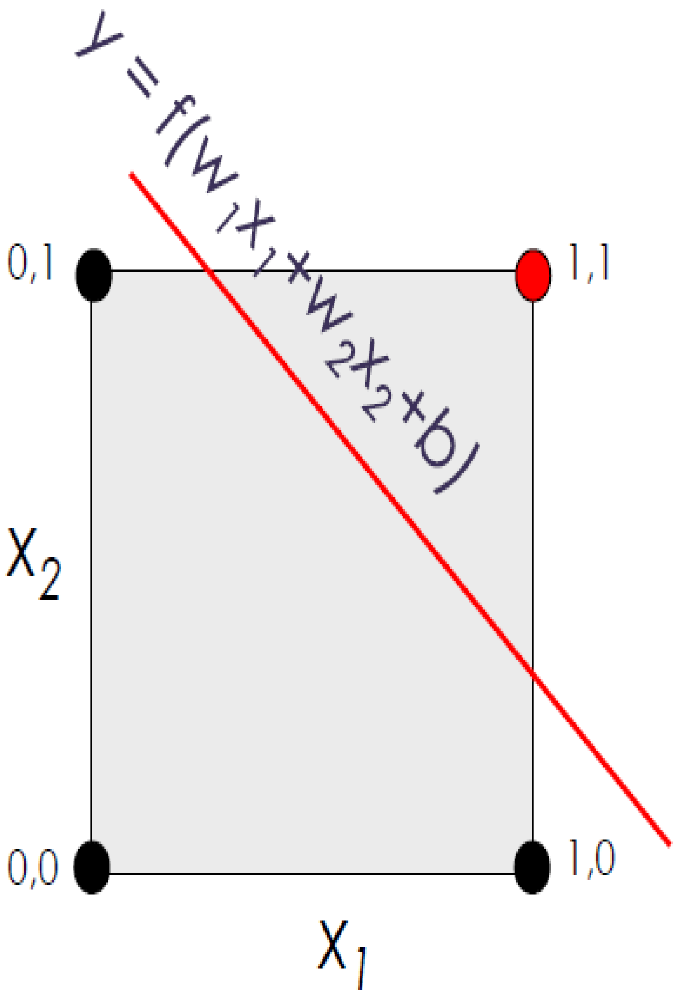
Perceptron Ex: Logical AND

- "w_1=1.0"
- "w_2=1.0"
- "b=-1.5"



Logical AND Function

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



Limitations of Perceptron

- Perceptron able to form only linear discriminate functions
  - i.e. classes which can be **divided by a line or hyperplane**

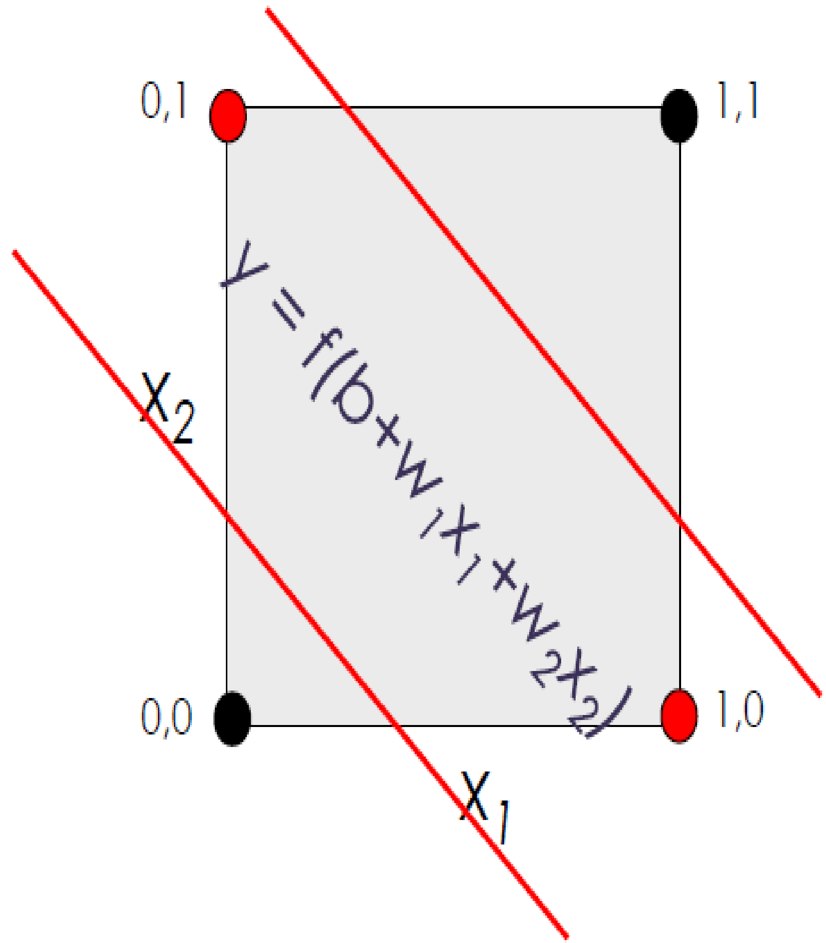
- Most functions are **more complex**
  - i.e. they are **non-linear or not linearly separable**
  - **Ex: Ex-OR**

Logical Ex-OR Operation

- Their combined results can produce good classification
- How to classify linearly?

Logical Ex-OR Function

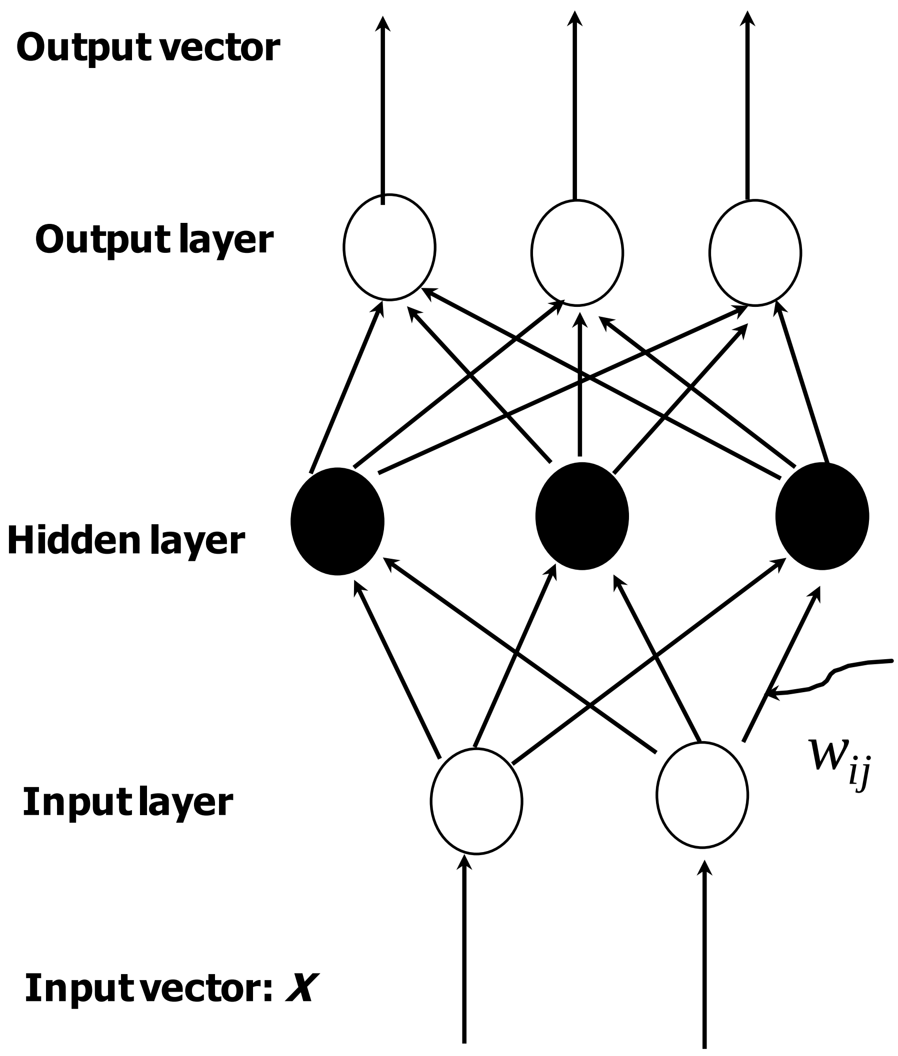
X_1	X_2	y
0	0	0
0	1	1
1	0	1
1	1	0

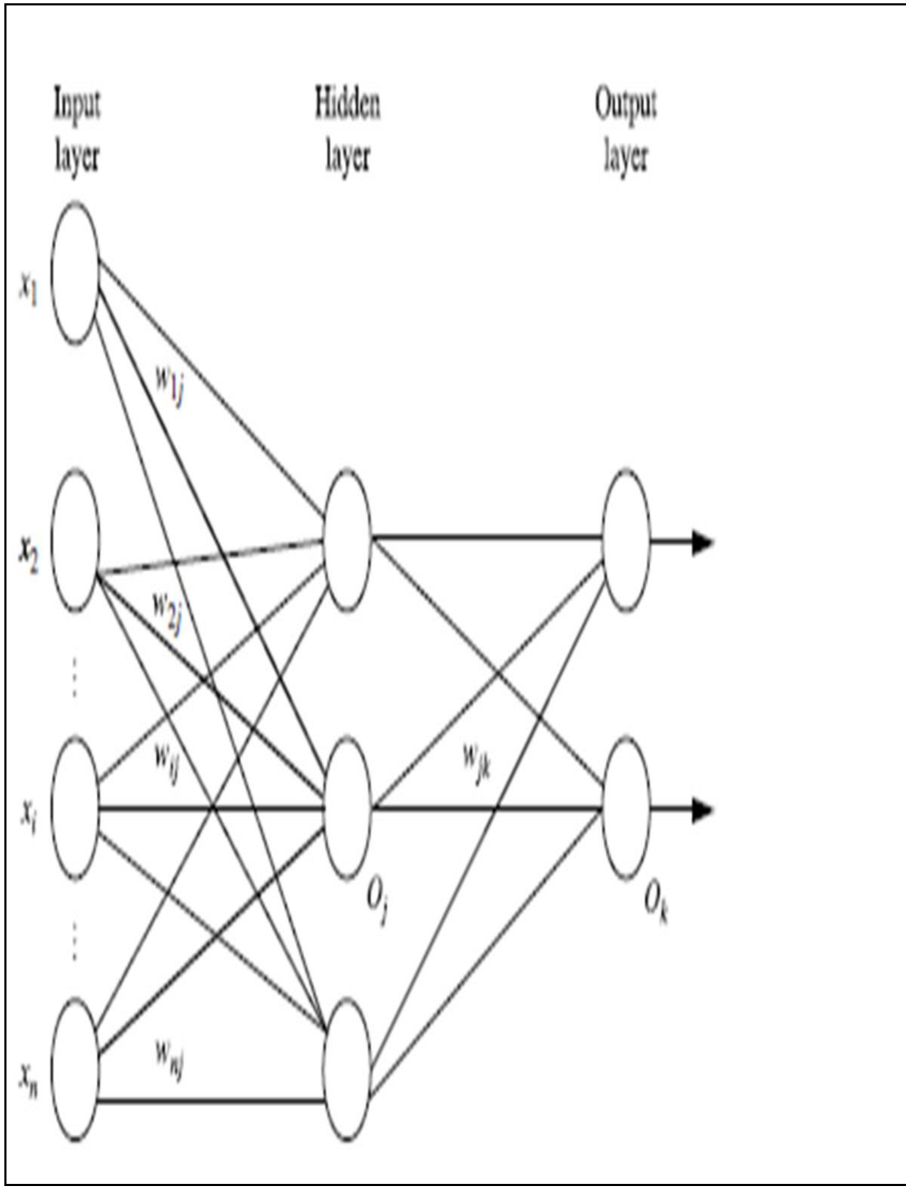


Artificial Neural Network/Multi-Layer Perceptron

- **A neural network:** A set of connected input/output units where each connection has a **weight** associated with it
- During the learning phase, the **network learns by adjusting the weights** so as to be able to predict the correct class label of the input tuples
- Also referred to as **connectionist learning** due to the connections between units

A Multi-Layer Feed-Forward Neural Network





How A Multi-Layer Neural Network Works

- The **inputs** to the network correspond to the attributes measured for each training tuple
- Inputs are fed simultaneously into the units making up the **input layer**
- They are then weighted and fed simultaneously to a **hidden layer**
- The number of hidden layers is arbitrary, although usually only one
- The weighted outputs of the last hidden layer are input to units making up the **output layer**, which emits the network's prediction
- The network is **feed-forward**: None of the weights cycles back to an input unit or to the previous layer

Defining a Network Topology

- Decide the **network topology or Structure:**
  - # of units in the *input layer*,
  - # of *hidden layers* (if > 1),
  - # of units in *each hidden layer*, and
  - # of units in the *output layer*
- Normalize the input values for each attribute measured in the training tuples to [0.0—1.0]
- One **input** unit per domain value, each initialized to 0
- **Output**, if for classification and more than two classes, one output unit per class is used
- Once a network has been trained and its **accuracy is unacceptable, repeat the training process with a *different network topology* or a *different set of initial weights***

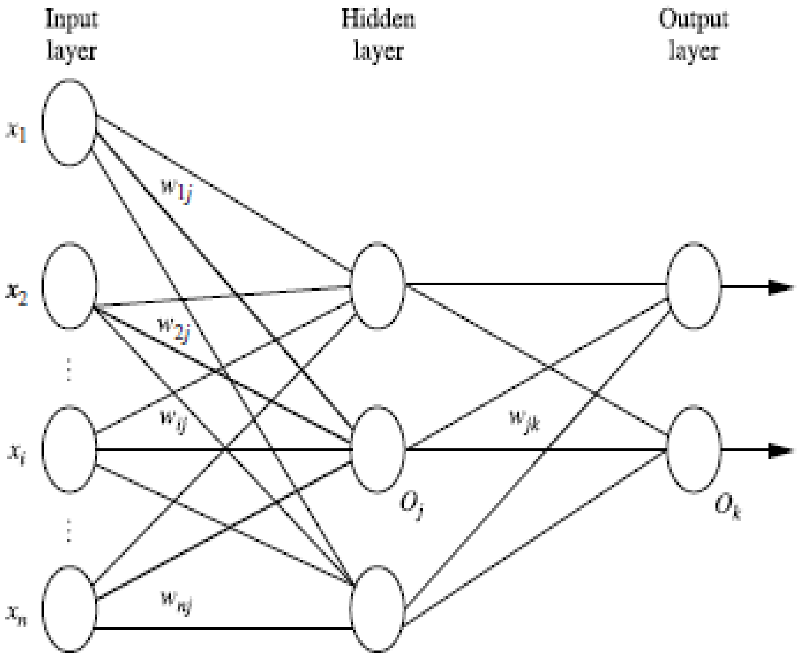
Backpropagation

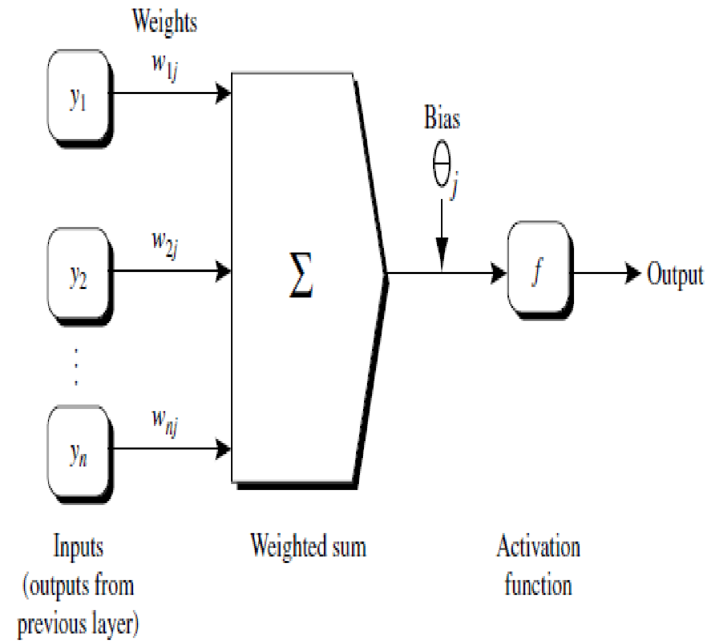
- Iteratively process a set of training tuples & compare the network's prediction with the actual known target value
- For each training tuple, the weights are modified to **minimize the mean squared error** between the network's prediction and the actual target value
- Modifications are made in the "**backwards**" direction: from the output layer, through each hidden layer down to the first hidden layer, hence "**backpropagation**"
- **Steps**
  - Initialize weights to small random numbers, associated with biases
  - Propagate the inputs forward (by applying activation function)
  - Backpropagate the error (by updating weights and biases)
  - Terminating condition (when error is very small, etc.)

Step1: Initialize the Weights

- **Initialize the weights:**
  - The weights in the network are initialized to small random numbers
  - e.g., ranging from -1.0 to 1.0, or -0.5 to 0.5.
  - Each unit has a *bias* associated with it.
  - The biases are similarly initialized to small random numbers.

Step 2: Propagating the inputs forward



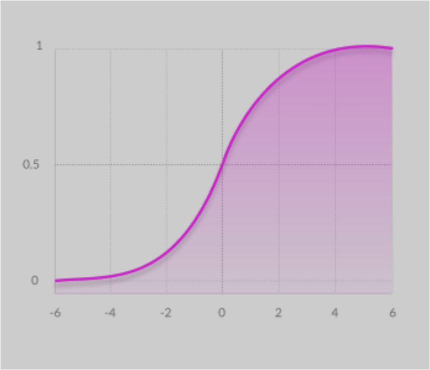


- An *n*-dimensional input vector **x** is mapped into variable y by means of the scalar product and a nonlinear function mapping
- The inputs to unit are outputs from the previous layer. They are multiplied by their corresponding weights to form a weighted sum, which is added to the bias associated with unit. Then a nonlinear activation function is applied to it.

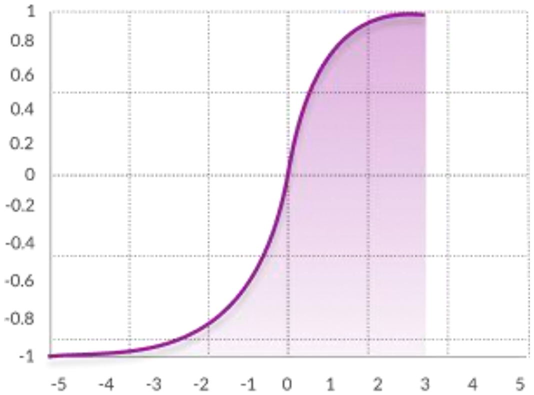
Step 2: Propagating the inputs forward

Different Activation Functions:

- **Sigmoid Function :**
  - $A = 1/(1 + e^{-x})$
  - **Value Range :** 0 to 1

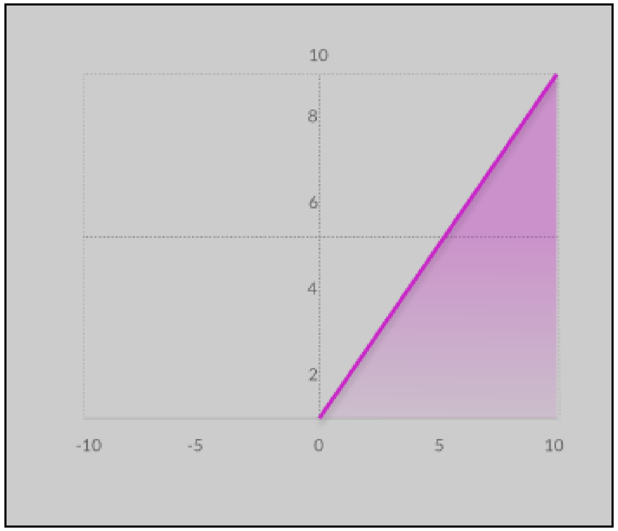


- **Tanh Function :-**
  - The activation that works almost always better than sigmoid function
  - $\tanh(x) = 2/(1 + e^{-2x}) - 1$ OR
  - $\tanh(x) = 2 * \text{sigmoid}(2x) - 1$
  - **Value Range :-** -1 to +1



Step 2: Propagating the inputs forward

- **RELU :-** Stands for *Rectified linear unit*.
  - It is the most widely used activation function.
  - ***A(x) = max(0,x)***.
  - It gives an output x if x is positive and 0 otherwise.
  - **Value Range :-** [0, inf)
  - In simple words, RELU learns *much faster* than sigmoid and Tanh function.



Step 2: Propagating the inputs forward

- **Given a unit, *j* in a hidden or output layer**, the net input, *Ij* , to unit *j* is

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

- where *wij* is the weight of the connection from unit *i* in the previous layer to unit *j*; *Oi* is the output of unit *i* from the previous layer; and *j* is the **bias** of the unit.
- Applies an **activation** function to it. The function symbolizes the activation of the neuron represented by the unit. The **ReLu or Tanh**, or **sigmoid**, function is used.
- Given the net input *Ij* to unit *j*, then *Oj* , the output of unit *j*, is computed as

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

Step 2: Propagating the inputs forward

- This function is also referred to as a ***squashing function,*** because it maps a large input domain onto the smaller range of 0 to 1.

- We **compute the output values, *Oj*** , for each hidden layer, up to and including the output layer, which gives the **network's prediction**.

Step 3: BackPropagating the Error

- The error is propagated backward by updating the weights and biases to reflect the error of the network's prediction.
- **For a unit *j* in the output layer,** the error E is computed by

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

- where *yi* is the obtained output of unit *i*, and *Ti* is the known target value of the given training tuple.

- Backpropagate the error using Gradient Decent technique.

Step 4: Terminating Condition

- **Terminating condition:**
- Training stops when
  - All *delta wij* in the previous epoch are so small as to be below some specified threshold, or
  - Error at output layer is below the specified threshold, or
  - A pre specified number of epochs has expired.
- In practice, several hundreds of thousands of epochs may be required before the weights will converge.

Topics

- **Motivation to ANN**
- **Perceptron**
- **Gradient Descent**
- **ANN with Backpropagation**

Training Rules

- understanding how to train the network (adjusting the weights) for a single perceptron/MLP

- Several algorithms are known to solve this learning problem. Here we consider two:
  - Perceptron rule
  - Delta rule

Gradient Descent and the Delta Rule

- Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable.

- **A second training rule, called the *delta rule, is designed*** to overcome this difficulty.

Gradient Descent

To understand, consider simpler *linear unit*, where

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

Let's learn w_i's that minimize the squared error

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

Where D is set of training examples

Gradient Descent

Gradient

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

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.,

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

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

Gradient Descent

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

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

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

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

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

Training rule:

$$\Delta\vec{w} = -\eta\nabla E[\vec{w}]$$

i.e.,

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

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

Process of Gradient Descent

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta (t - o) x_i$$

  - For each linear unit weight w_i, Do

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

Standard Vs Stochastic Gradient Descent

- One common variation on gradient descent is called *incremental gradient descent, or stochastic gradient descent.*

- The key differences between standard gradient descent and stochastic gradient descent are:
  - In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
  - In cases where there are multiple local minima, stochastic gradient descent can sometimes avoid falling into these local minima

Difficulties in Gradient Descent

- The key practical difficulties in applying gradient descent are:
  - converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps)
  - if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

Topics

- **Motivation to ANN**
- **Perceptron**
- **Gradient Descent**
- **ANN with Backpropagation**

Notations Used

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network

Backpropagation-SGD

- For each training example d every weight w_{ji} ***is updated by adding to it*** Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \qquad\qquad E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

- We now derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ ***in order to implement the stochastic*** gradient descent***.***
- ***To begin, notice that weight*** w_{ji} can influence the rest of the network only through net_j.
- ***Therefore, we can use the*** chain rule to write

$$\begin{aligned} \frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial net_j} x_{ji} \end{aligned}$$

Backpropagation-SGD

Case 1: Training Rule for Output Unit Weights. Just as w_{ji} can influence the rest of the network only through net_j, net_j can influence the network only through o_j. Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j} \tag{4.23}$$

To begin, consider just the first term in Equation (4.23)

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

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

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

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

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

Backpropagation-SGD

Partial Derivative for Sigmoid Activation Function:

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

Sigmoid function

$$\sigma'(x) = \frac{d}{dx}\sigma(x) = \frac{d}{dx}\frac{1}{1+e^{-x}}$$

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

$$= -(1+e^{-x})^{-2}.\frac{d}{dx}(1+e^{-x})$$

$$= -(1+e^{-x})^{-2}.(\frac{d}{dx}[1] + \frac{d}{dx}[e^{-x}])$$

$$= -(1+e^{-x})^{-2}.(0 + \frac{d}{dx}[e^{-x}])$$

$$= -(1+e^{-x})^{-2}.(e^{-x}.\frac{d}{dx}[-x])$$

Backpropagation-SGD

$$= -(1+e^{-x})^{-2}.(e^{-x}.\frac{d}{dx}[-x])$$

$$= -(1+e^{-x})^{-2}.(e^{-x}.-1)$$

$$= (1+e^{-x})^{-2}.e^{-x}$$

$$= \frac{e^{-x}}{(1+e^{-x})^2}$$

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

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

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

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

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

$$= \sigma(x).(1-\sigma(x))$$

$$= o_j(1-o_j)$$

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

$$\boxed{\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j)\end{aligned}}$$

$$\boxed{\begin{aligned}\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j)\end{aligned}}$$

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

$$\begin{aligned}\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j}\ \frac{\partial net_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial net_j}\ x_{ji}\end{aligned}$$

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

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

Backpropagation-SGD

Case 2: Training Rule for Hidden Unit Weights. In the case where j is an internal, or hidden unit in the network, the derivation of the training rule for w_{ji} must take into account the indirect ways in which w_{ji} can influence the network outputs and hence E_d. For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network (i.e., all units whose direct inputs include the output of unit j). We denote this set of units by $Downstream(j)$. Notice that net_j can influence the network outputs (and therefore E_d) only through the units in $Downstream(j)$. Therefore, we can write

$$
\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1 - o_j)
\end{aligned}
\tag{4.28}
$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

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

and

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

Backpropagation-SGD

$$\begin{bmatrix} w_1^+ \\ w_2^+ \\ \vdots \\ w_n^+ \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \vdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$

Classification by Backpropagation

Algorithm: Backpropagation. Neural network learning for classification or numeric prediction, using the backpropagation algorithm.

Input:

- D, a data set consisting of the training tuples and their associated target values;
- l, the learning rate;
- *network*, a multilayer feed-forward network.

Output: A trained neural network.

Method:

(1) Initialize all weights and biases in *network*;
(2) **while** terminating condition is not satisfied {
(3) **for each** training tuple X in D {
(4)   // Propagate the inputs forward:
(5)   **for each** input layer unit j {
(6) $O_j = I_j$; // output of an input unit is its actual input value
(7)   **for each** hidden or output layer unit j {
(8) $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit j with respect to the previous layer, i
(9) $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
(10)   // Backpropagate the errors:
(11)   **for each** unit j in the output layer
(12) $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
(13)   **for each** unit j in the hidden layers, from the last to the first hidden layer
(14) $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$; // compute the error with respect to the next higher layer, k
(15)   **for each** weight w_{ij} in *network* {
(16) $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
(17) $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18)   **for each** bias θ_j in *network* {
(19) $\Delta \theta_j = (l) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta \theta_j$; } // bias update
(21)   } }