MODULE -3

ARTIFICIAL NEURAL NETWORKS

CONTENT

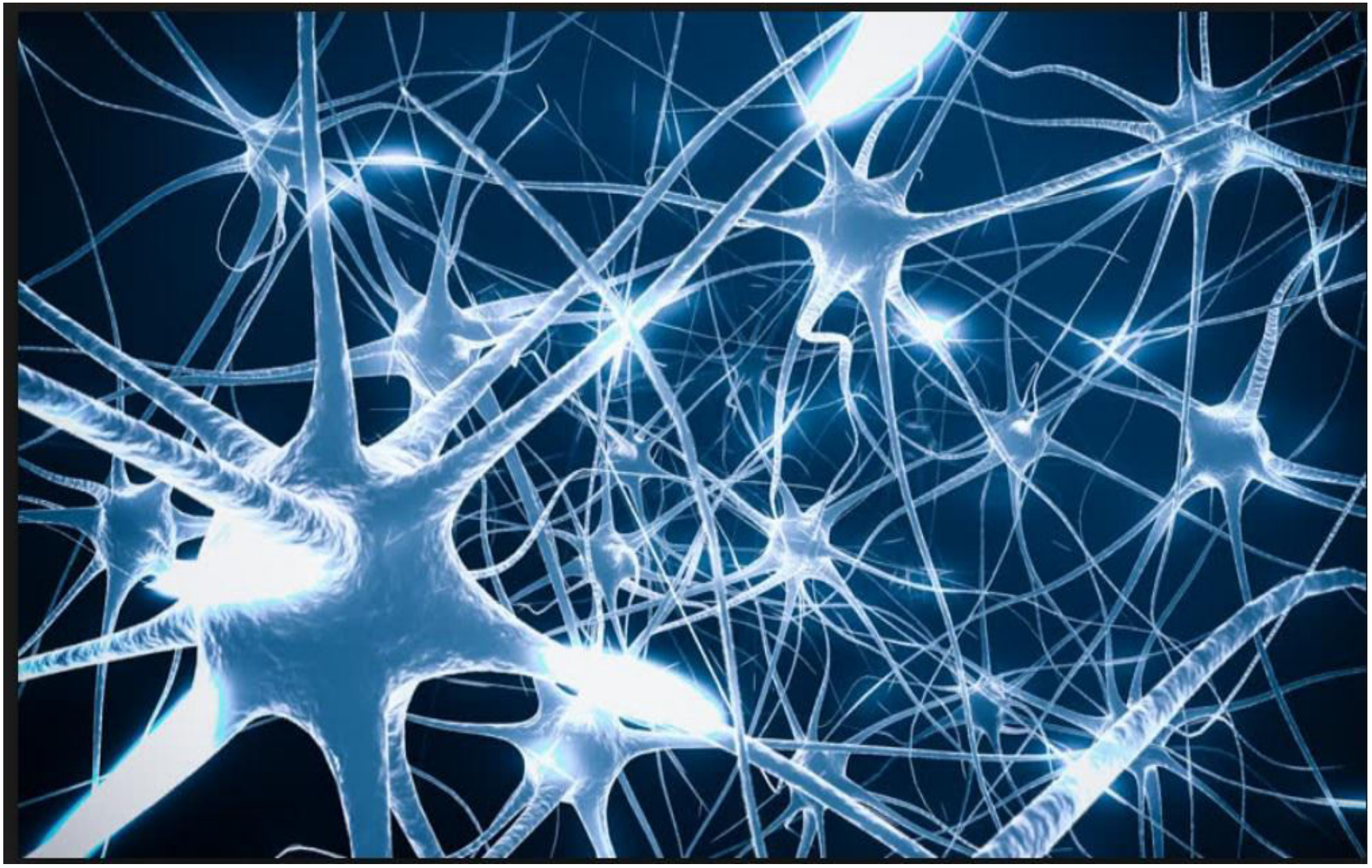
- Introduction
- Neural Network Representation
- Appropriate Problems for Neural Network Learning
- Perceptrons
- Multilayer Networks and BACKPROPAGATION Algorithms
- Remarks on the BACKPROPAGATION Algorithms

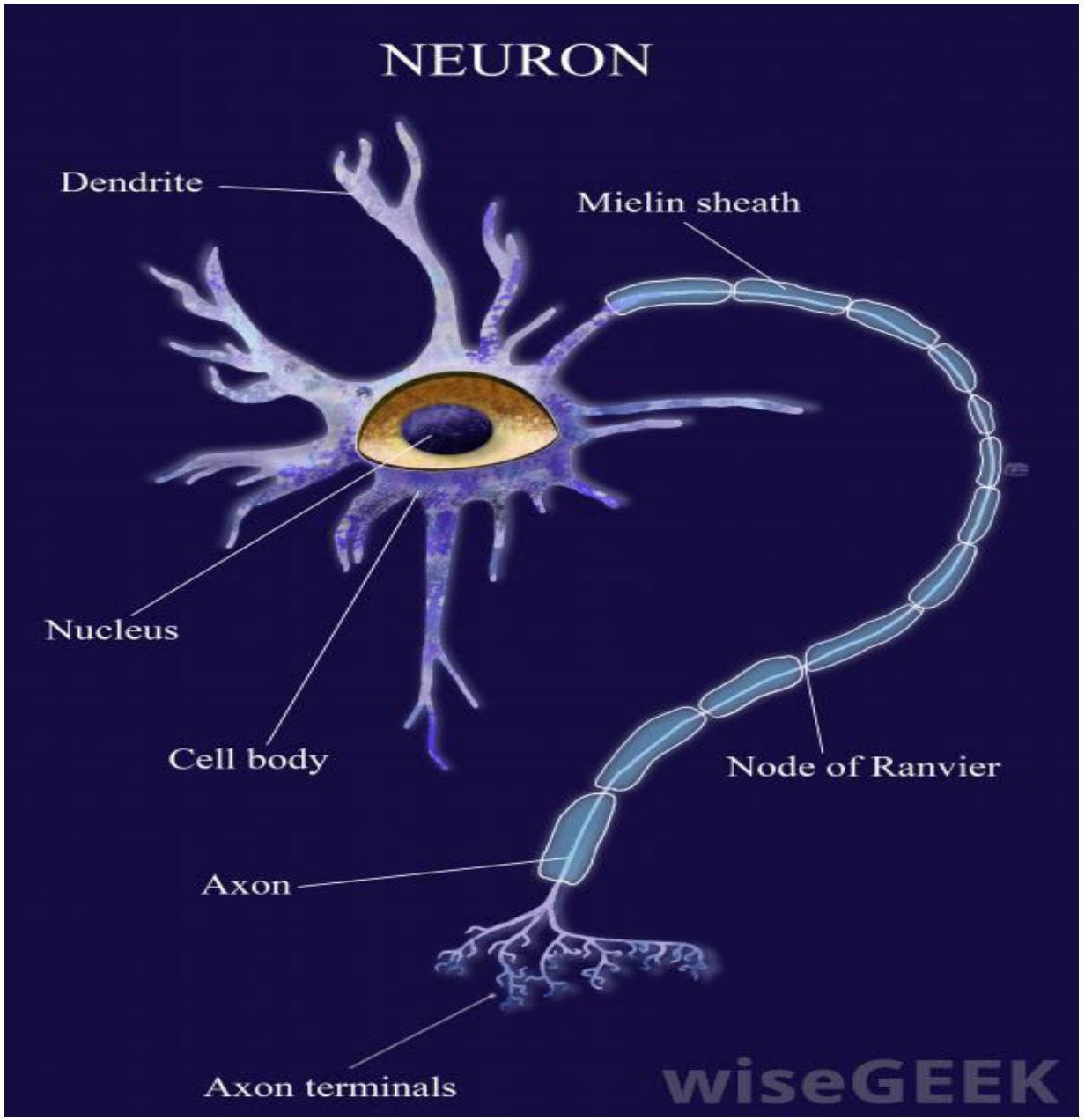
INTRODUCTION

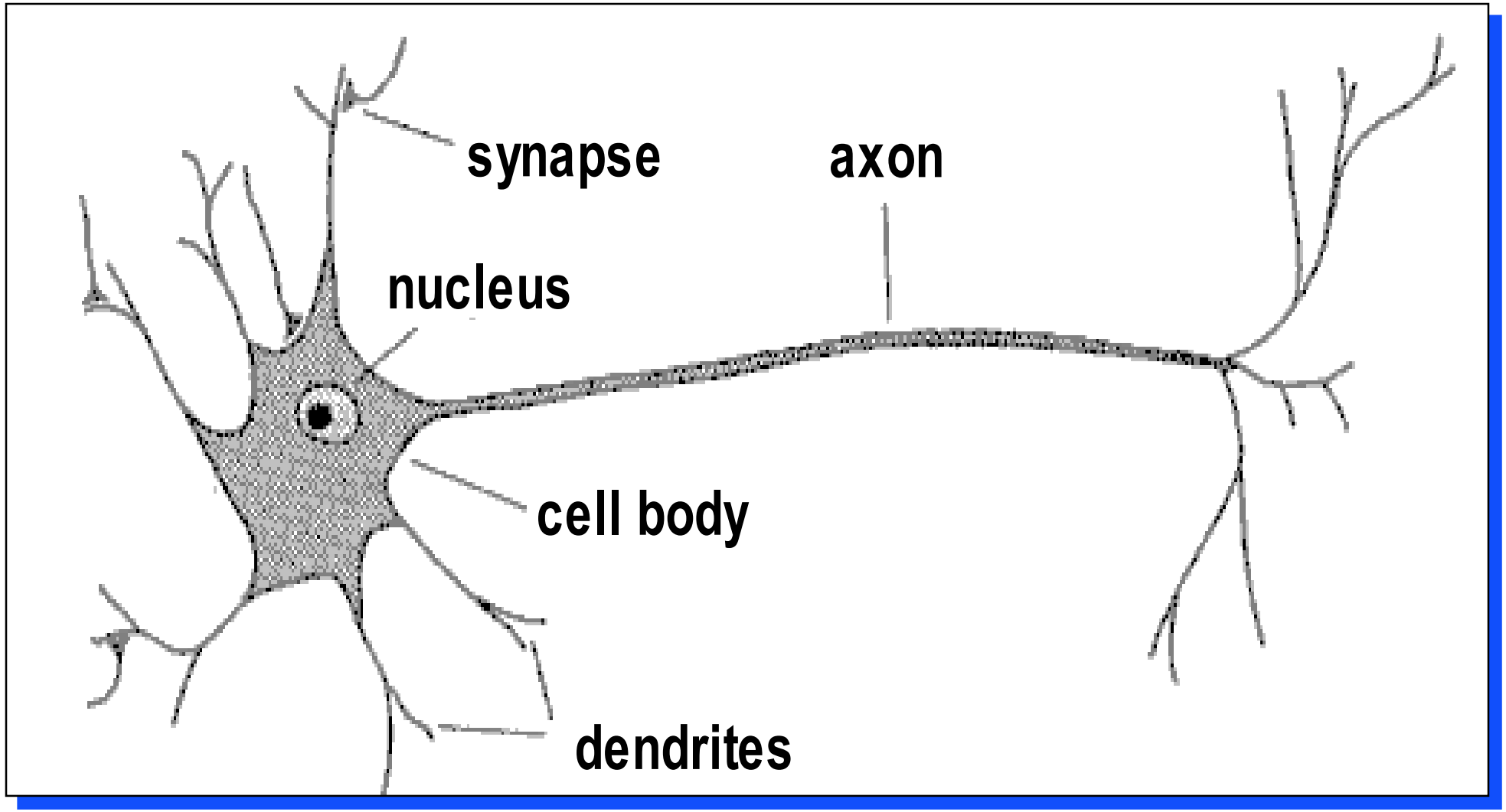
Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued target functions from examples.

Biological Motivation

- The study of artificial neural networks (ANNs) has been inspired by the observation that biological learning systems are built of very complex webs of interconnected ***Neurons***
- Human information processing system consists of brain neuron: basic building block cell that communicates information to and from various parts of body
- Simplest model of a neuron: considered as a threshold unit –a processing element (PE)
- Collects inputs & produces output if the sum of the input exceeds an internal threshold value




NEURON
Dendrite
Mielin sheath
Nucleus
Cell body
Node of Ranvier
Axon
Axon terminals
wiseGEEK


synapse
axon
nucleus
cell body
dendrites

Facts of Human Neurobiology

- Number of neurons ~ 10^{11}
- Connection per neuron ~ 10^{4-5}
- Neuron switching time ~ 0.001 second or 10^{-3}
- Scene recognition time ~ 0.1 second
- 100 inference steps doesn't seem like enough
- Highly parallel computation based on distributed representation

Properties of Neural Networks

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically
- Input is a high-dimensional discrete or real-valued (e.g, sensor input)

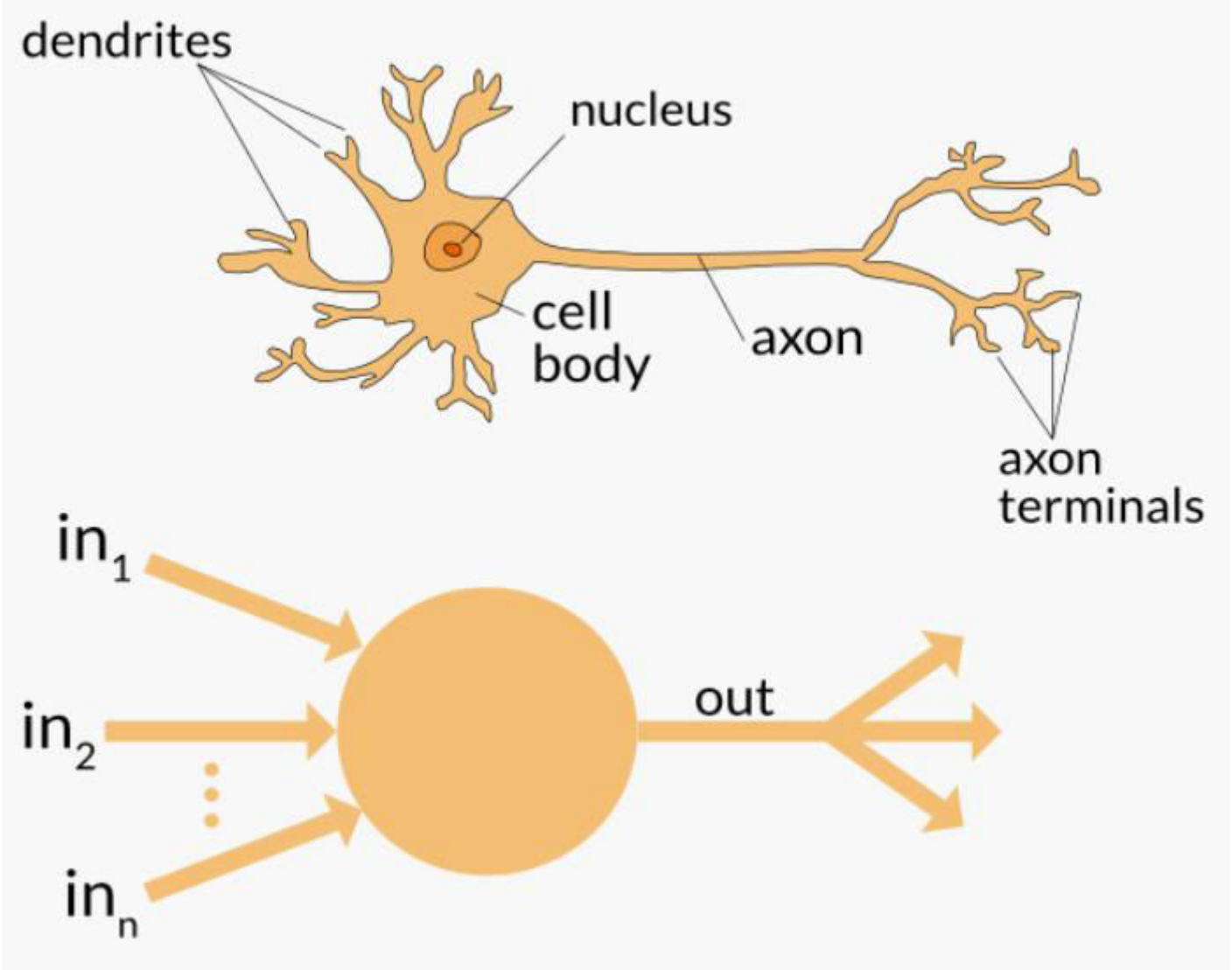
When to consider Neural Networks ?

- Input is a high-dimensional discrete or real-valued (e.g., sensor input)
- Output is discrete or real-valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

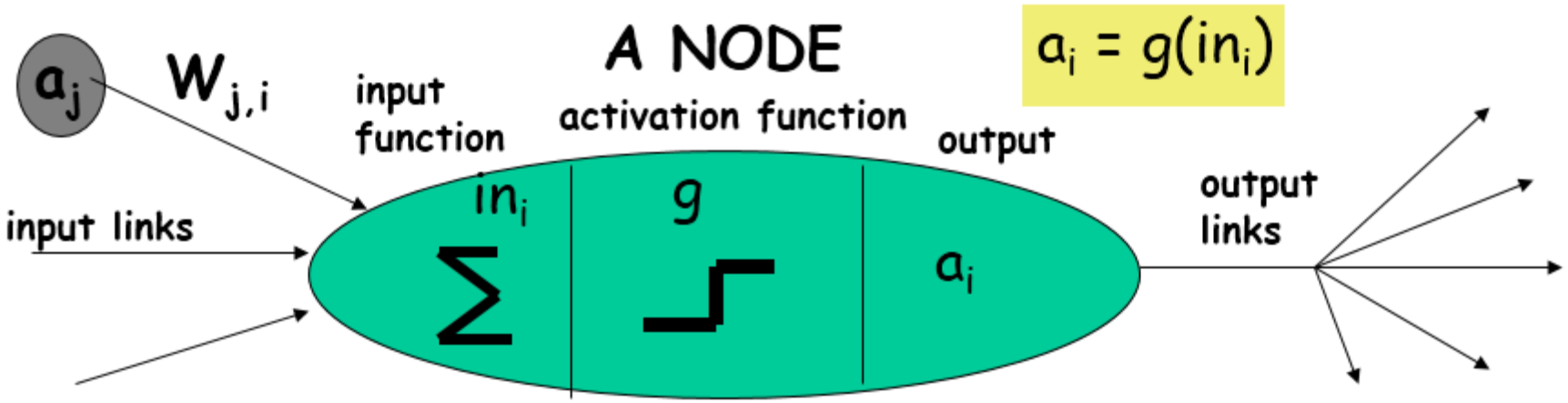
Examples:

1. Speech phoneme recognition
2. Image classification
3. Financial perdition

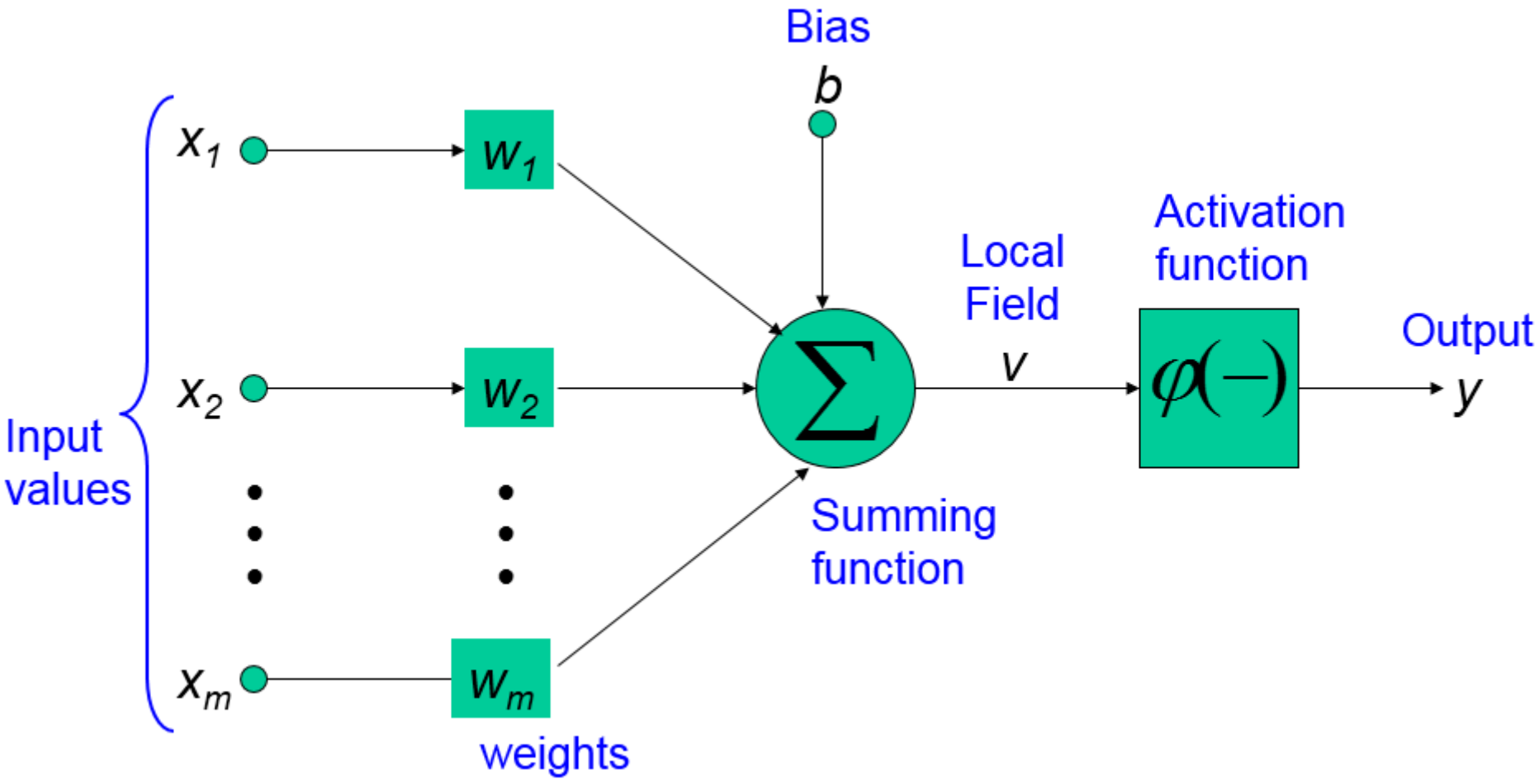
Neuron


dendrites
nucleus
cell body
axon
axon terminals
in1
in2
inn
out

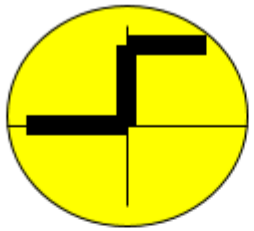
Neuron


A NODE
$a_i = g(in_i)$
a_j
$W_{j,i}$
input function
activation function
output
in_i
g
a_i
input links
output links

Neuron


Bias
b
x_1
x_2
x_m
w_1
w_2
w_m
Input values
weights
Summing function
Local Field
v
Activation function
$\varphi(-)$
Output
y

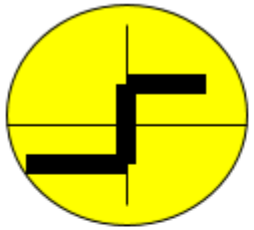
Neuron



Step function

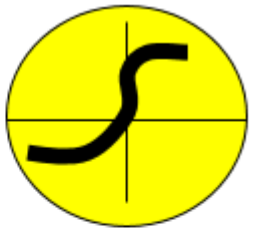
(Linear Threshold Unit)

step(x) = 1, if x >= threshold
0, if x < threshold



Sign function

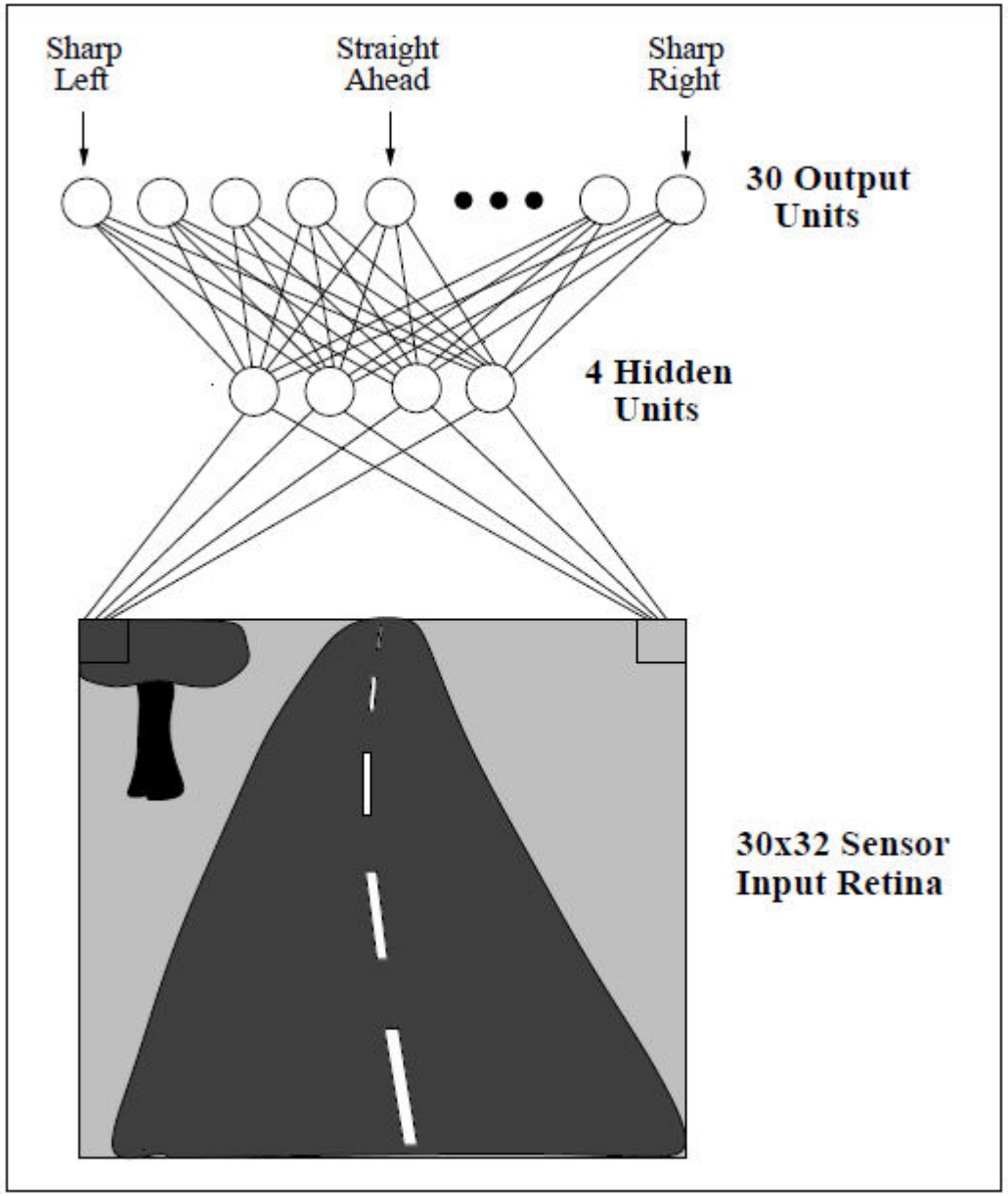
sign(x) = +1, if x >= 0
-1, if x < 0

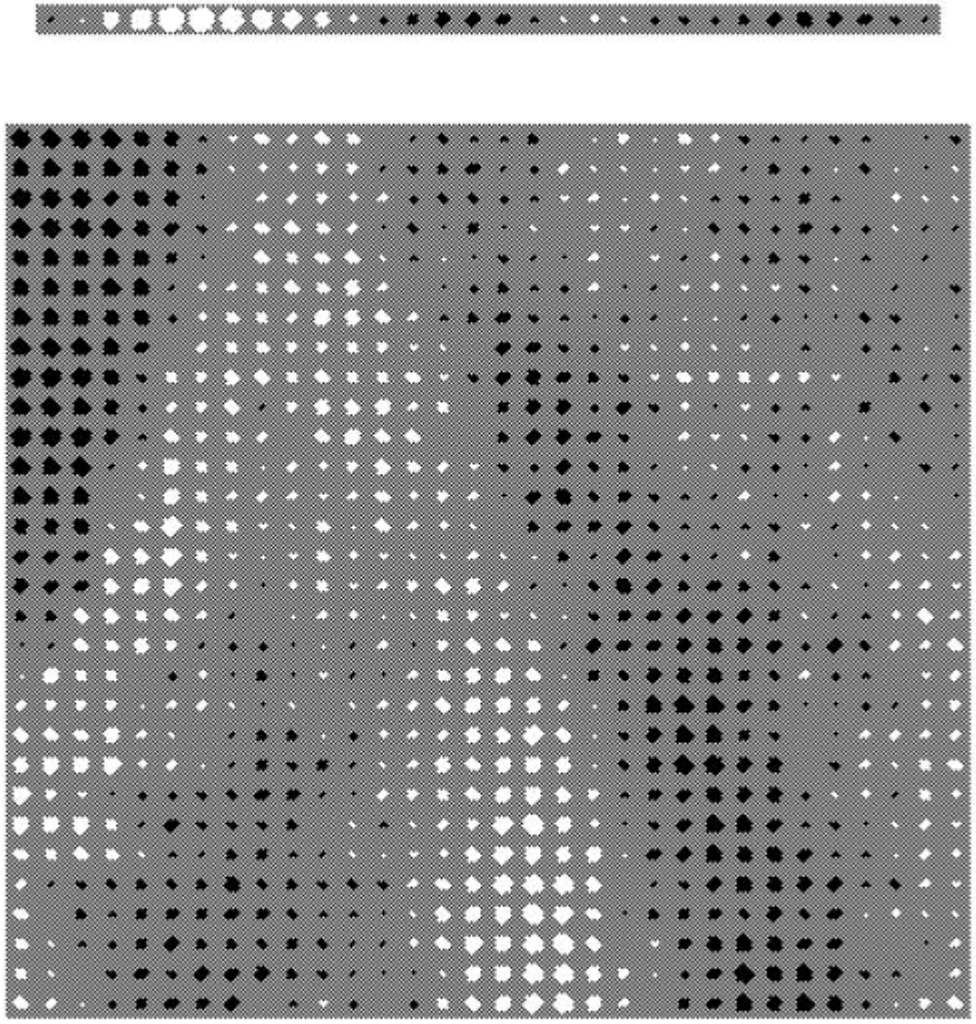


Sigmoid function

sigmoid(x) = $1/(1+e^{-x})$

NEURAL NETWORK REPRESENTATIONS


Sharp
Left
Straight
Ahead
Sharp
Right
30 Output
Units
4 Hidden
Units
30x32 Sensor
Input Retina



- A prototypical example of ANN learning is provided by Pomerleau's (1993) system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways.

- The ***input*** to the neural network is a 30x32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle.

- The network ***output*** is the direction in which the vehicle is steered.

- Figure illustrates the neural network representation.
- The network is shown on the left side of the figure, with the input camera image depicted below it.
- Each node (i.e., circle) in the network diagram corresponds to the output of a single network ***unit,*** and the lines entering the node from below are its *inputs.*
- There are four units that receive inputs directly from all of the 30 x 32 pixels in the image. These are called *"hidden" units* because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs
- These hidden unit outputs are then used as inputs to a second layer of 30 "output" units.
- Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.

- The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN.
- The large matrix of black and white boxes on the lower right depicts the weights from the 30 x 32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude.
- The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN is appropriate for problems with the following characteristics :

- 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

Architectures of Artificial Neural Networks

An artificial neural network can be divided into three parts (layers), which are known as:

- ***Input layer:*** This layer is responsible for receiving information (data), signals, features, or measurements from the external environment. These inputs are usually normalized within the limit values produced by activation functions
- ***Hidden, intermediate, or invisible layers:*** These layers are composed of neurons which are responsible for extracting patterns associated with the process or system being analysed. These layers perform most of the internal processing from a network.
- ***Output layer :*** This layer is also composed of neurons, and thus is responsible for producing and presenting the final network outputs, which result from the processing performed by the neurons in the previous layers.

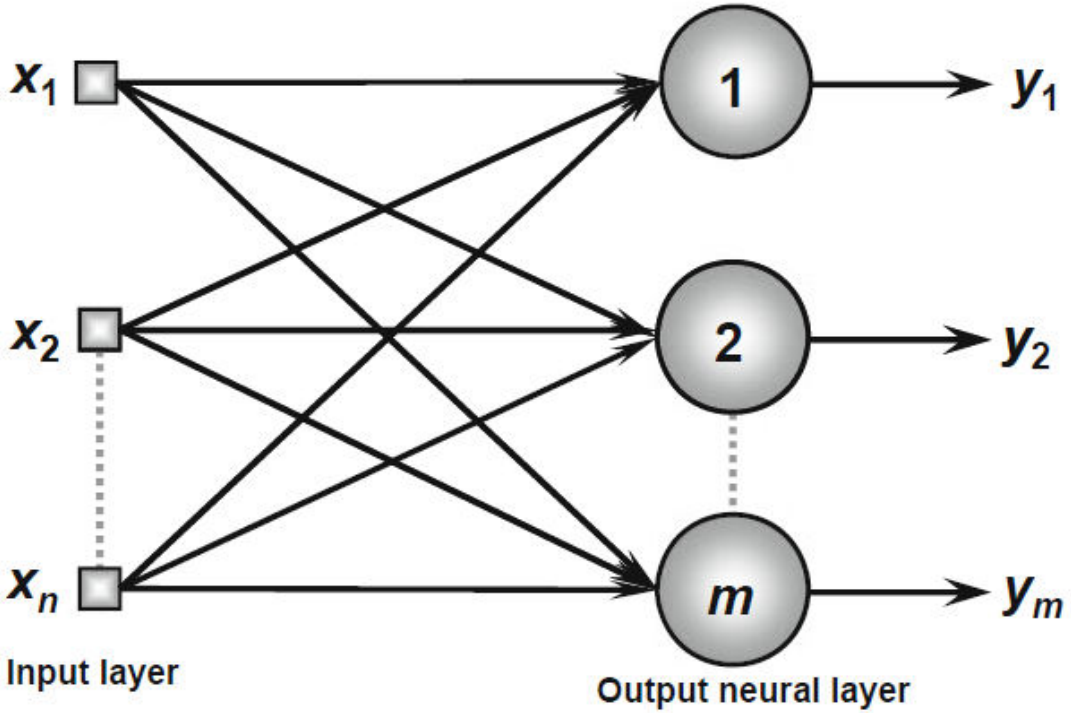
Architectures of Artificial Neural Networks

The main architectures of artificial neural networks, considering the neuron disposition, how they are interconnected and how its layers are composed, can be divided as follows:

1. Single-layer feedforward network
2. Multi-layer feedforward networks
3. Recurrent or Feedback networks
4. Mesh networks

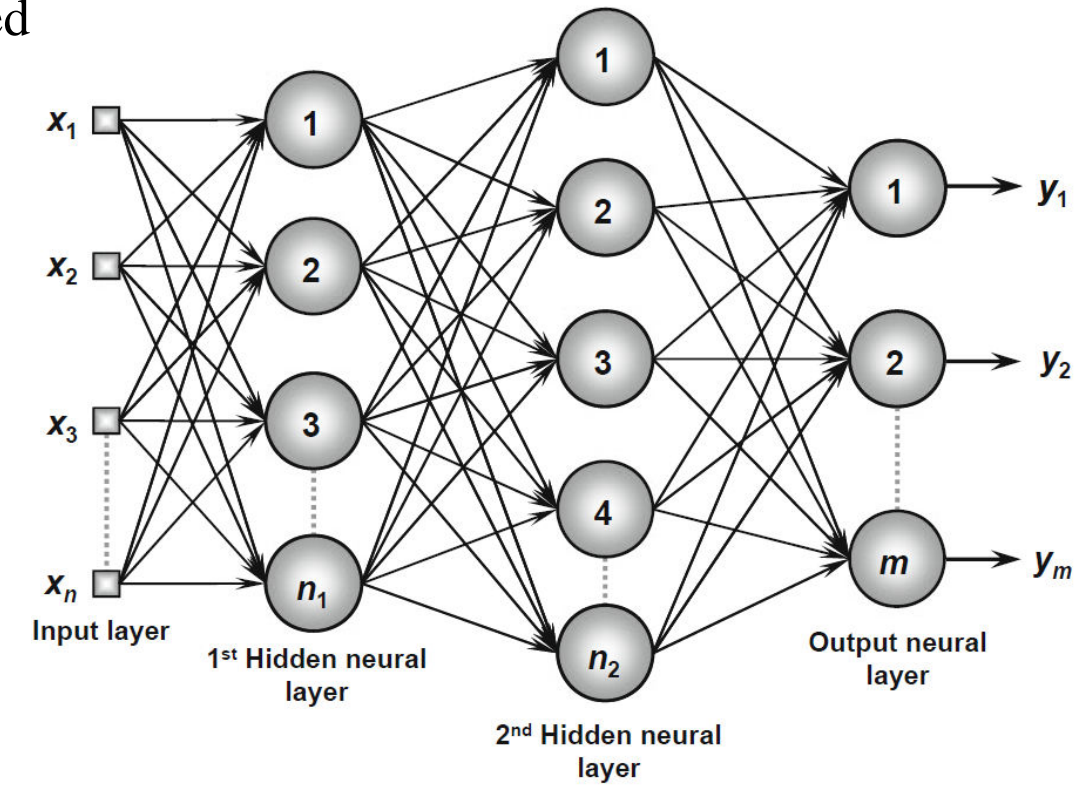
Single-Layer Feedforward Architecture

- This artificial neural network has just one input layer and a single neural layer, which is also the output layer.
- Figure illustrates a simple-layer feedforward network composed of n inputs and m outputs.
- The information always flows in a single direction (thus, unidirectional), which is from the input layer to the output layer



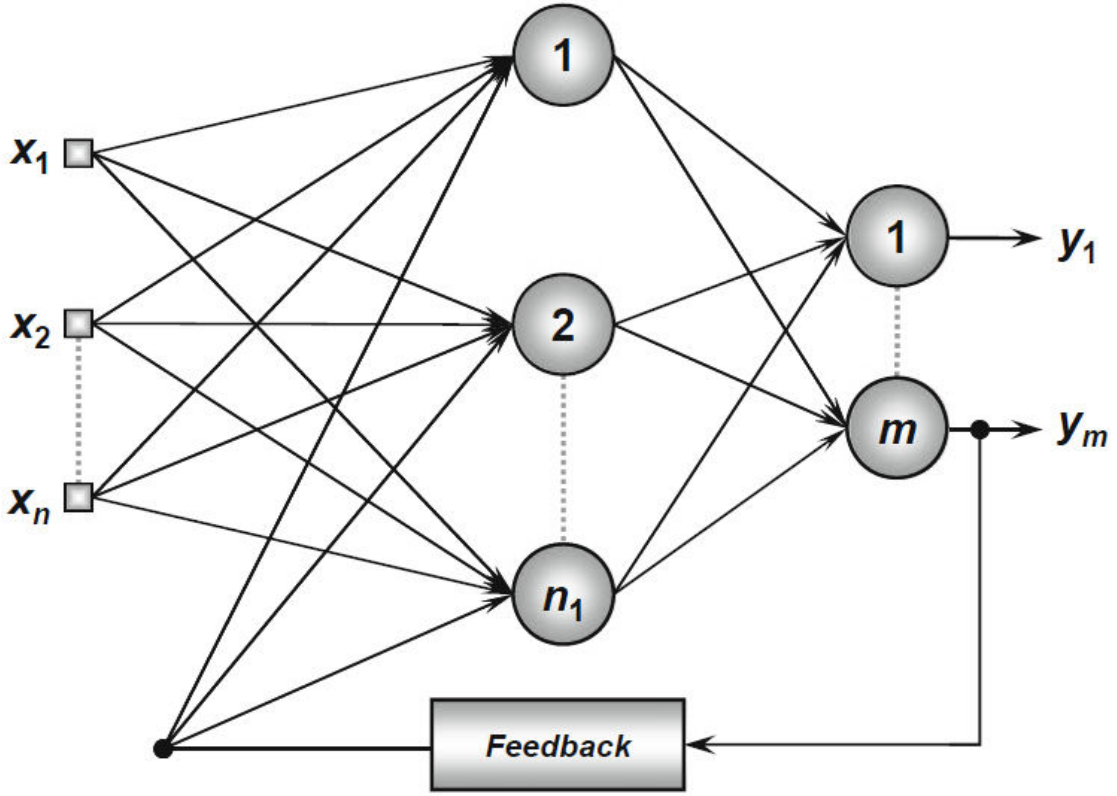
Multi-Layer Feedforward Architecture

- This artificial neural feedforward networks with multiple layers are composed of one or more hidden neural layers.
- Figure shows a feedforward network with multiple layers composed of one input layer with n sample signals, two hidden neural layers consisting of n_1 and n_2 neurons respectively, and, finally, one output neural layer composed of m neurons representing the respective output values of the problem being analyzed



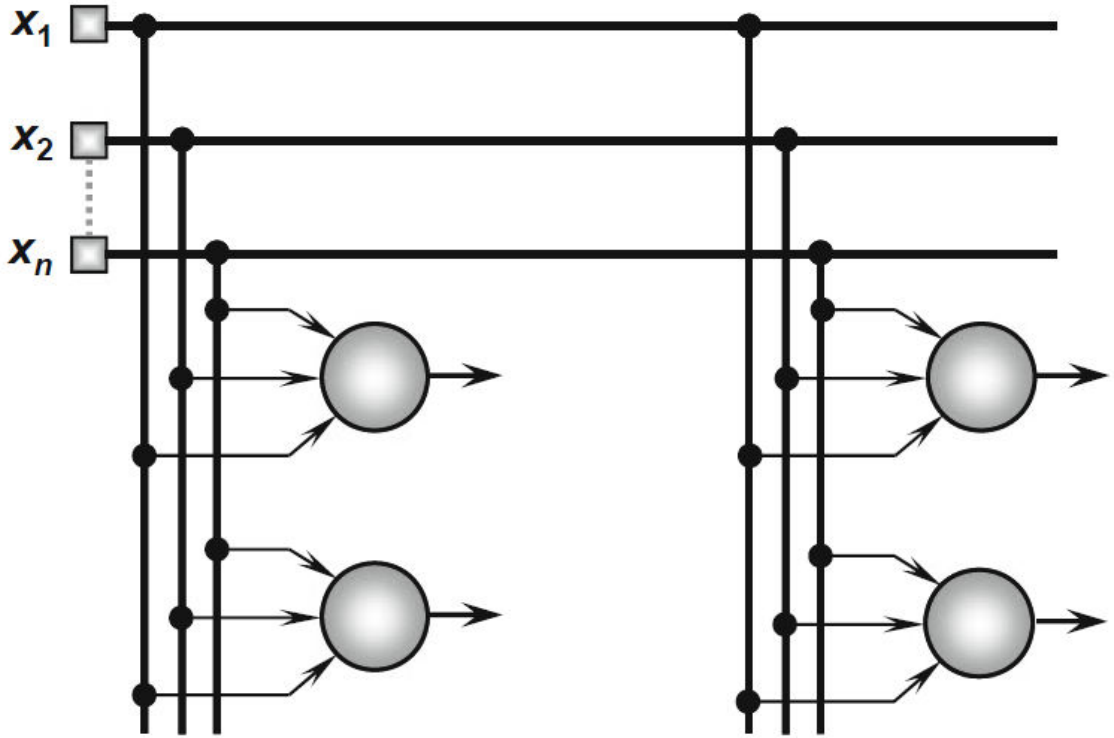
Recurrent or Feedback Architecture

- In these networks, the outputs of the neurons are used as feedback inputs for other neurons.
- Figure illustrates an example of a Perceptron network with feedback, where one of its output signals is fed back to the middle layer.



Mesh Architectures

- The main features of networks with mesh structures reside in considering the spatial arrangement of neurons for pattern extraction purposes, that is, the spatial localization of the neurons is directly related to the process of adjusting their synaptic weights and thresholds.
- Figure illustrates an example of the Kohonen network where its neurons are arranged within a two-dimensional space



PERCEPTRONS

- Perceptron is a single layer neural network.
- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise
- Given inputs x_1 through x_n, the output $O(x_1, \ldots, x_n)$ computed by the perceptron is

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

- where each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.
- $-w_0$ is a threshold that the weighted combination of inputs $w_1 x_1 + \ldots + w_n x_n$ must surpass in order for the perceptron to output a 1.

Sometimes, the perceptron function is written as,

$$O(\vec{x}) = \text{sgn}\ (\vec{w} \cdot \vec{x})$$

Where,

$$\text{sgn}(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$. Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors

$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

Why do we need Weights and Bias?

Weights shows the strength of the particular node.

A bias value allows you to shift the activation function curve up or down

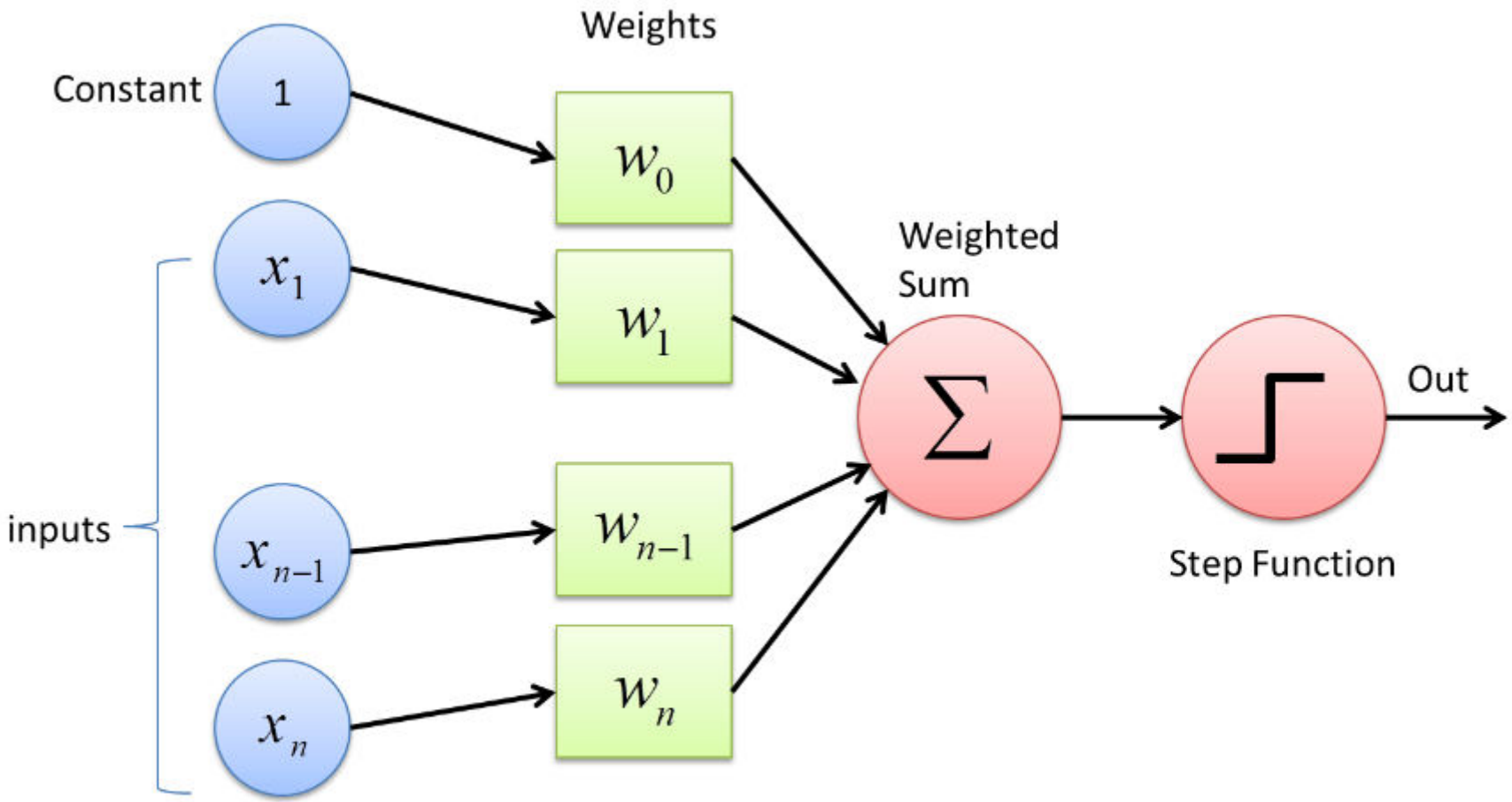
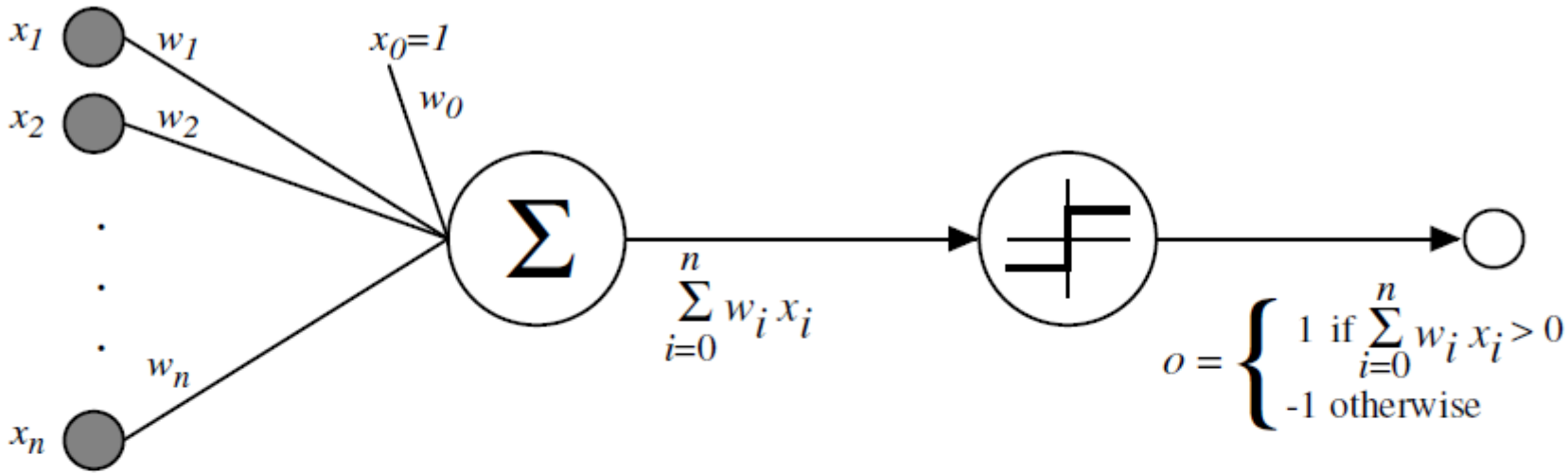


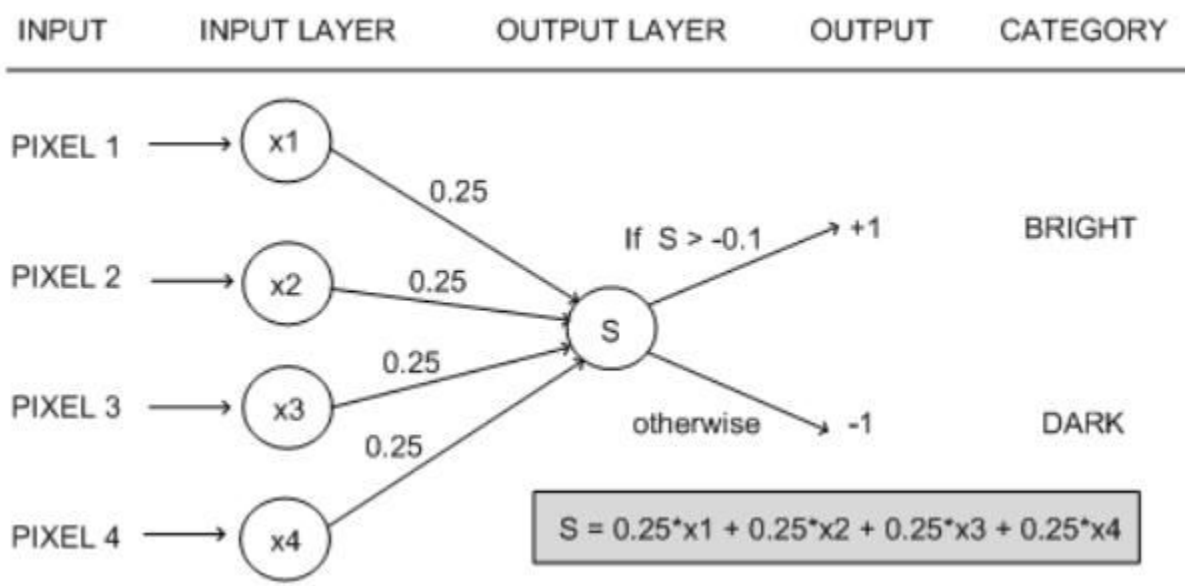
Fig : Perceptron



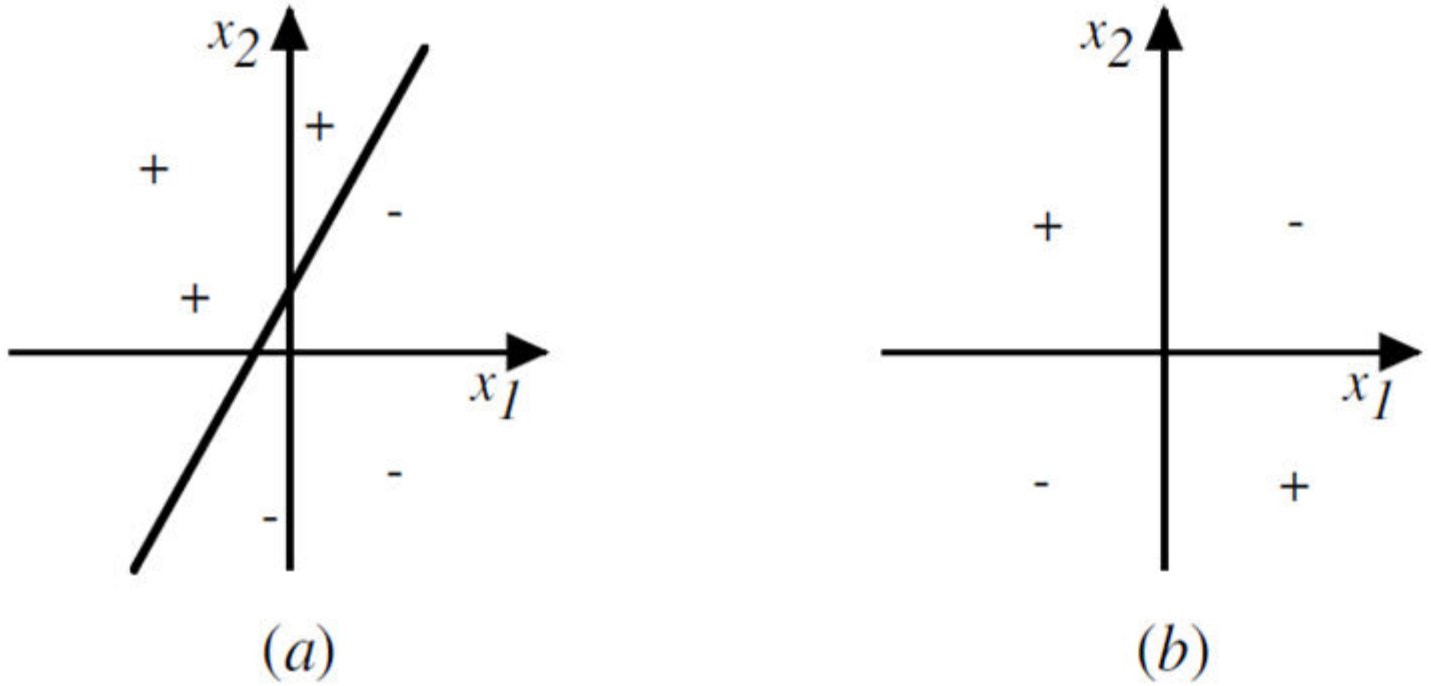
$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

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


INPUT
INPUT LAYER
OUTPUT LAYER
OUTPUT
CATEGORY
PIXEL 1
PIXEL 2
PIXEL 3
PIXEL 4
x1
x2
x3
x4
0.25
0.25
0.25
0.25
S
If S > -0.1
+1
BRIGHT
otherwise
-1
DARK
S = 0.25*x1 + 0.25*x2 + 0.25*x3 + 0.25*x4

Representational Power of Perceptrons



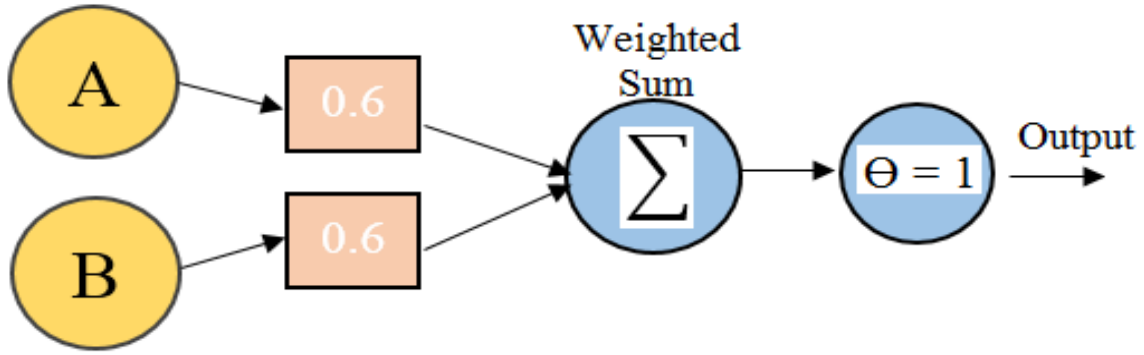
* The perceptron can be viewed as representing a hyperplane decision surface in the n-dimensional space of instances.
* The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side

Figure : The decision surface represented by a two-input perceptron.
(a) A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable.
x_1 and x_2 are the Perceptron inputs. Positive examples are indicated by "+", negative by "-".

A single perceptron can be used to represent many Boolean functions

AND function

A	B	A ^ B
0	0	0
0	1	0
1	0	0
1	1	1



- If A=0 & B=0 → 0*0.6 + 0*0.6 = 0.
  This is not greater than the threshold of 1, so the output = 0.
- If A=0 & B=1 → 0*0.6 + 1*0.6 = 0.6.
  This is not greater than the threshold, so the output = 0.
- If A=1 & B=0 → 1*0.6 + 0*0.6 = 0.6.
  This is not greater than the threshold, so the output = 0.
- If A=1 & B=1 → 1*0.6 + 1*0.6 = 1.2.
  This exceeds the threshold, so the output = 1.

The Perceptron Training Rule

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

To learn an acceptable weight vector

- Begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example.
- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.
- Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input x_i according to the rule.

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

Where,

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

Here,
t is the target output for the current training example
o is the output generated by the perceptron
η is a positive constant called the ***learning rate***

- The role of the ***learning rate*** is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases

Drawback: 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.

Gradient Descent and the Delta Rule

- If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.
- The key idea behind the ***delta rule*** is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

To understand the delta training rule, consider the task of training an unthresholded perceptron. That is, a linear unit for which the output O is given by

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

$$O(\vec{x}) = (\vec{w} \cdot \vec{x}) \qquad \text{equ. (1)}$$

To derive a weight learning rule for linear units, specify a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples.

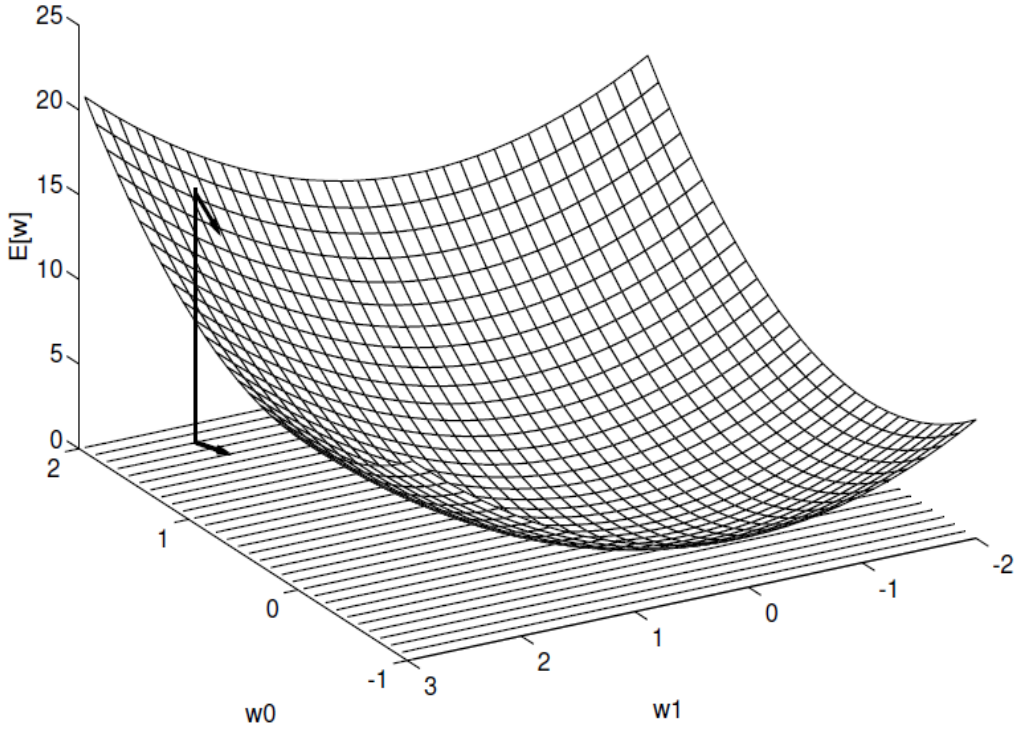
$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \qquad \text{equ. (2)}$$

Where,

- D is the set of training examples,
- t_d is the target output for training example d,
- o_d is the output of the linear unit for training example d
- E [$\overrightarrow{w}$] is simply half the squared difference between the target output t_d and the linear unit output o_d, summed over all training examples.

Visualizing the Hypothesis Space

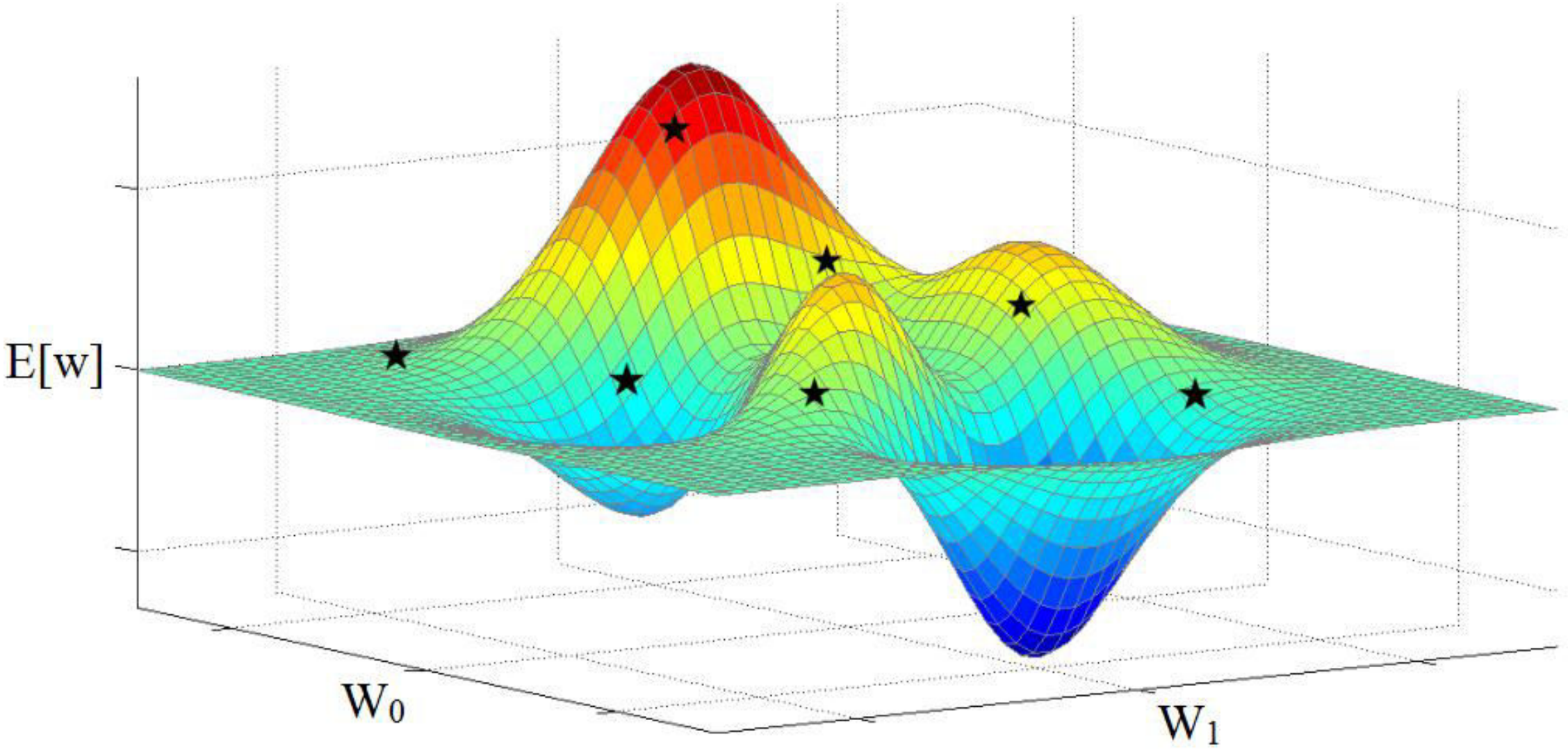
- To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values as shown in below figure.
- Here the axes w_0 and w_1 represent possible values for the two weights of a simple linear unit. The w_0, w_1 plane therefore represents the entire hypothesis space.
- The vertical axis indicates the error E relative to some fixed set of training examples.
- The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.
- The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space

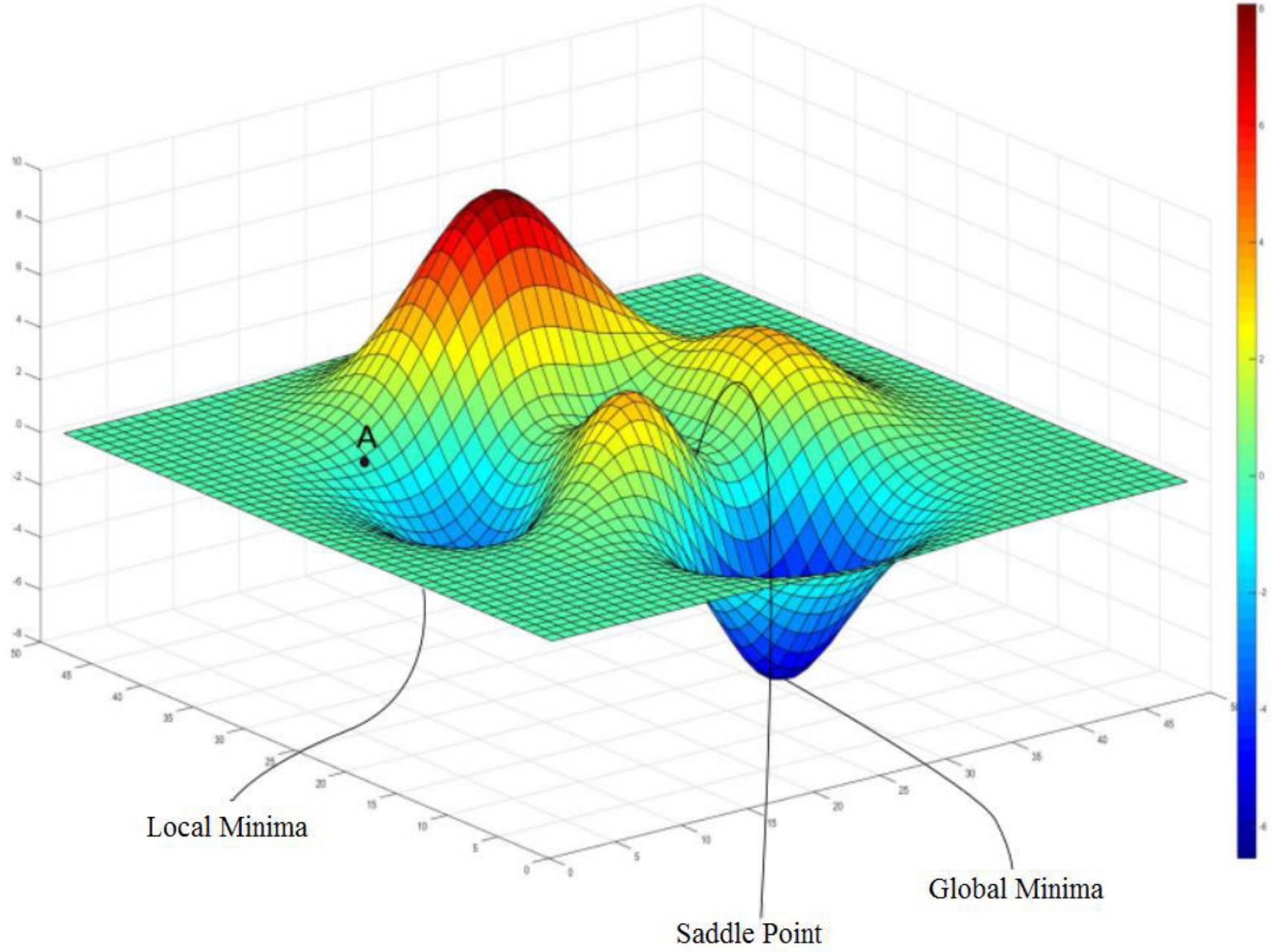


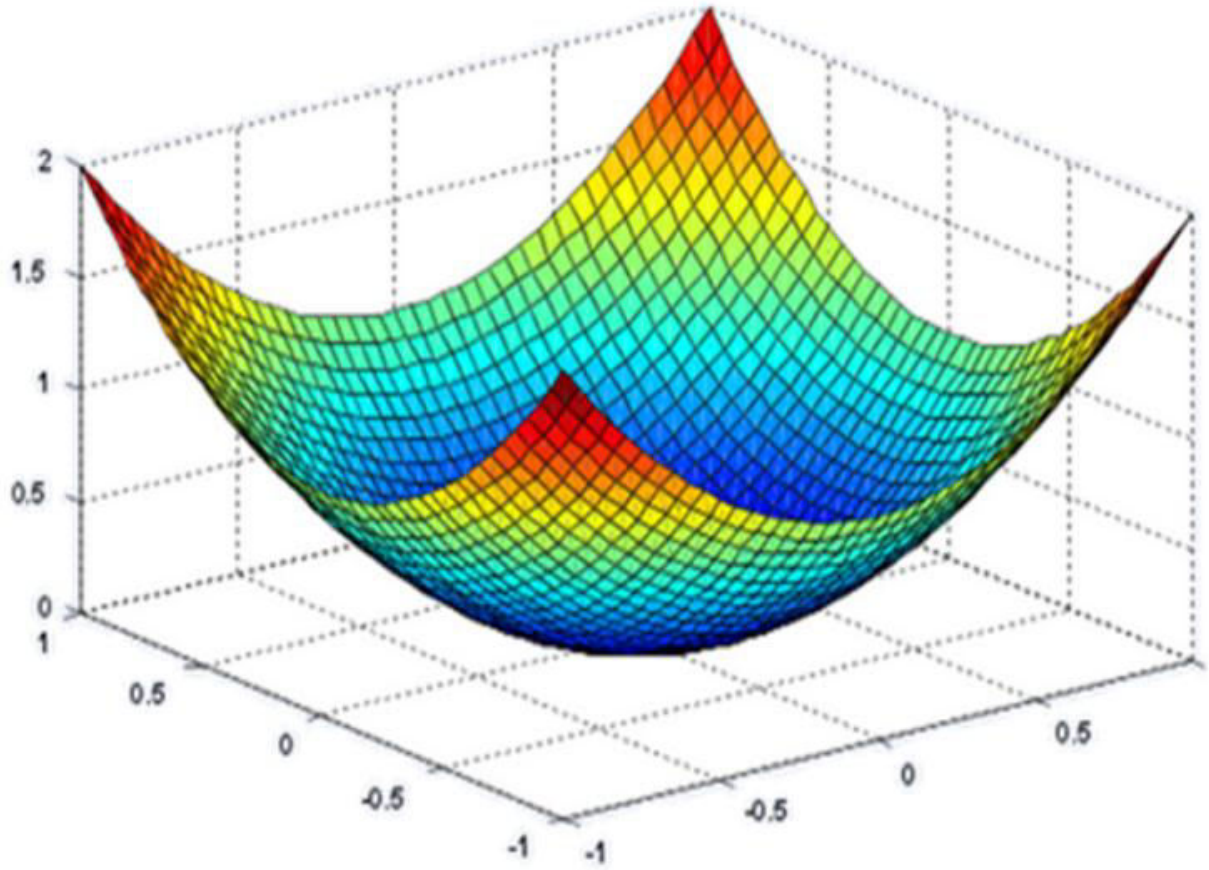
- Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum.

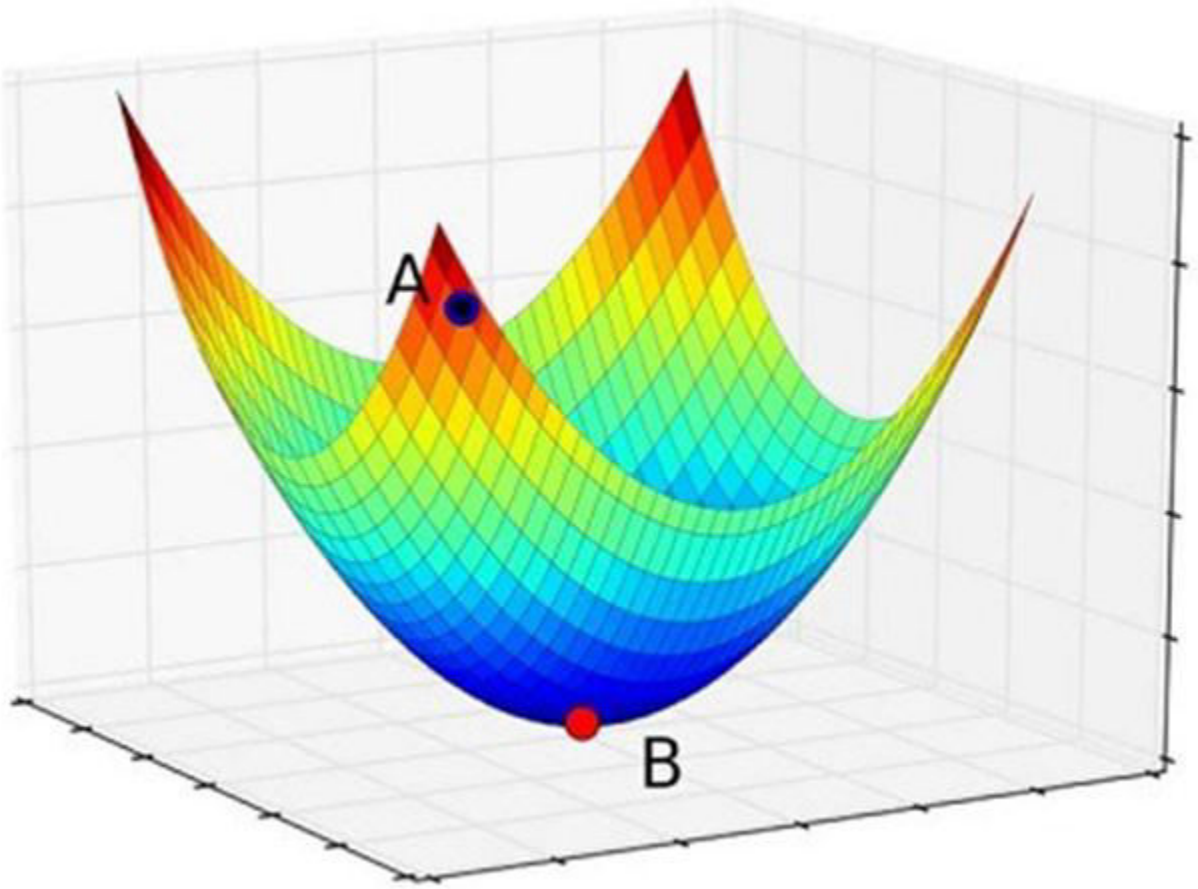
Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.

At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in above figure. This process continues until the global minimum error is reached.


E[w]
W0
W1


A
Local Minima
Saddle Point
Global Minima


2
1.5
1
0.5
0
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5


A
B

Derivation of the Gradient Descent Rule

How to calculate the direction of steepest descent along the error surface?

The direction of steepest can be found by computing the derivative of E with respect to each component of the vector $\overrightarrow{w}$. This vector derivative is called the gradient of E with respect to $\overrightarrow{w}$, written as

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right] \qquad \text{equ. (3)}$$

Notice $\nabla E[\vec{w}]$ is itself a vector, whose components are the partial derivatives of ***E*** with respect to each of the w_i

When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E.

The negative of this vector therefore gives the direction of steepest decrease.

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

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

Where,

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \qquad \text{equ. (4)}$$

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

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

Where,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \qquad \text{equ. (5)}$$

Calculate the gradient at each step. The vector of $\frac{\partial E}{\partial wi}$ derivatives that form the gradient can be obtained by differentiating E from Equation (2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d}) \qquad \text{equ. (6)}
\end{aligned}$$

Substituting Equation (6) into Equation (5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\ x_{id} \qquad \text{equ. (7)}$$

GRADIENT DESCENT algorithm for training a linear unit

GRADIENT-DESCENT($training\_examples, \eta$)

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

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

To summarize, the gradient descent algorithm for training linear units is as follows:

- Pick an initial random weight vector.
- Apply the linear unit to all training examples, then compute Δw_i for each weight according to Equation (7).
- Update each weight w_i by adding Δw_i, then repeat this process

Features of Gradient Descent Algorithm

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever

1. The hypothesis space contains continuously parameterized hypotheses
2. The error can be differentiated with respect to these hypothesis parameters

The key practical difficulties in applying gradient descent are

1. Converging to a local minimum can sometimes be quite slow
2. If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum

Stochastic Approximation to Gradient Descent

- The gradient descent training rule presented in Equation *(7)* computes weight updates after summing over ***all*** the training examples in D
- The idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for ***each*** individual example

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

where t, o, and x_i are the target value, unit output, and i^{th} input for the training example in question

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

$$w_i \leftarrow w_i + \eta(t - o)\, x_i \tag{1}$$

stochastic approximation to gradient descent

One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ defined for each individual training example d as follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ for each individual training example d **as** follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

Where, t_d and o_d are the target value and the unit output value for training example d.

- Stochastic gradient descent iterates over the training examples ***d*** in D, at each iteration altering the weights according to the gradient with respect to $E_d(\vec{w})$
- The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation to descending the gradient with respect to our original error function $E_d(\vec{w})$
- By making the value of **η** sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely

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.
- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.
- In cases where there are multiple local minima with respect to stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode Gradient Descent:
Do until satisfied

- For each training example d in D
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

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

$$E_d[\vec{w}] \equiv \frac{1}{2}(t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

Multilayer networks learned by the **BACKPROPACATION** algorithm are capable of expressing a rich variety of nonlinear decision surfaces

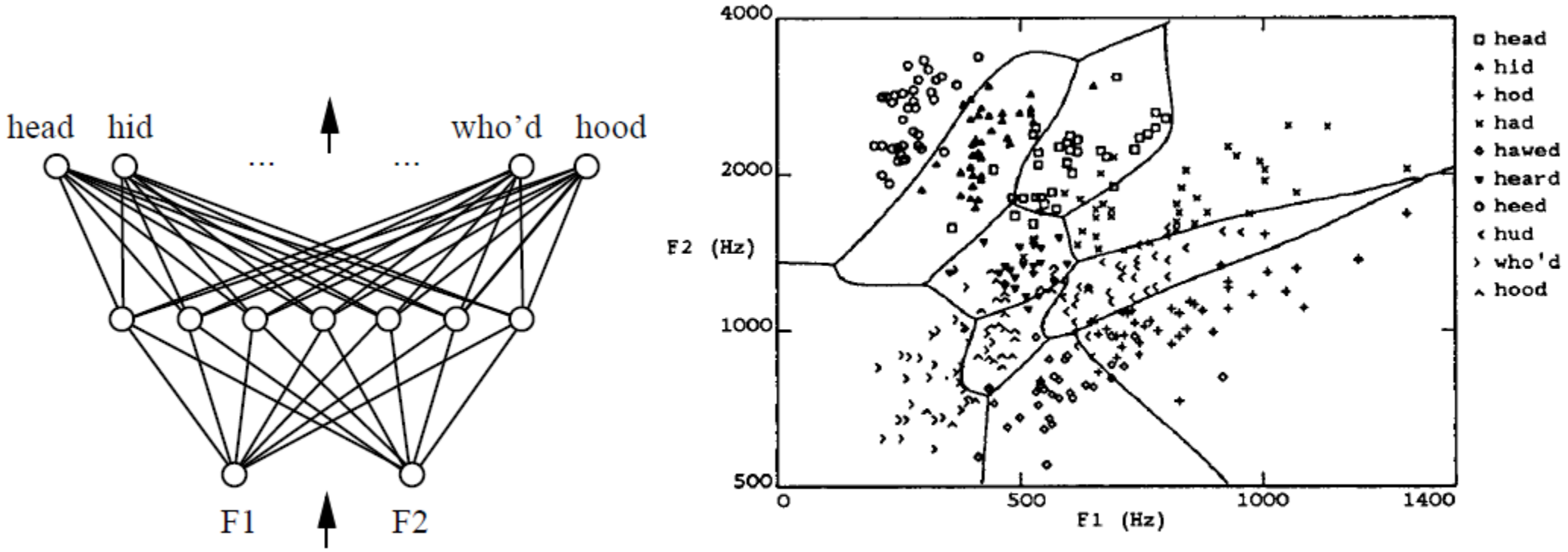


Figure: Decision regions of a multilayer feedforward network.

- Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of ***10*** vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest.
- The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network.

A Differentiable Threshold Unit

- Sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.
- The sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a continuous function of its input.
- More precisely, the sigmoid unit computes its output ***O*** as

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

Where,

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

σ is the sigmoid function

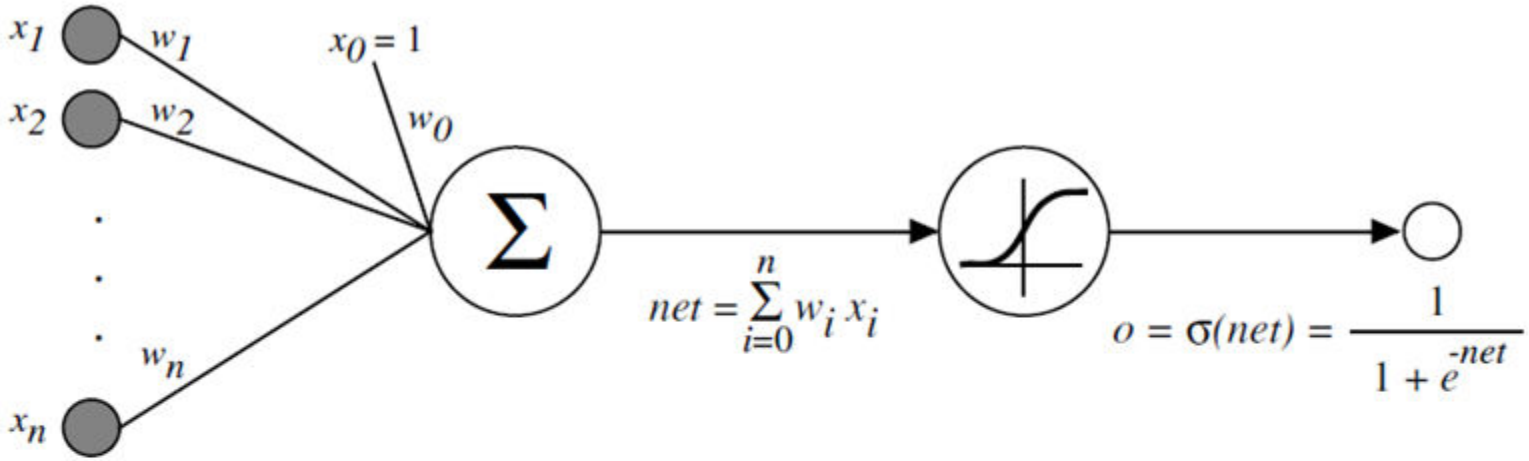


Figure: A Sigmoid Threshold Unit

$\sigma(y)$ is the sigmoid function

$$\frac{1}{1 + e^{-y}}$$

Nice property: $\frac{d\sigma(y)}{dy} = \sigma(y)(1 - \sigma(y))$

The BACKPROPAGATION Algorithm

- The BACKPROPAGATION Algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.
- In BACKPROPAGATION algorithm, we consider networks with multiple output units rather than single units as before, so we redefine E to sum the errors over all of the network output units.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \quad \text{.....equ. (1)}$$

where,

- ***outputs*** **-** is the set of output units in the network
- t_{kd} and O_{kd} - the target and output values associated with the k^{th} output unit
- d - training example

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

Each training example is a pair of the form $(\vec{x}, \vec{t})$, *where* $(\vec{x})$ *is the vector of network input values,* $(\vec{t})$ *and is the vector of target network output values.*
η *is the learning rate (e.g., .05).* n_i *is the number of network inputs,* n_{hidden} *the number of units in the hidden layer, and* n_{out} *the number of output units.*
The input from unit i into unit j is denoted x_{ji}, *and the weight from unit i to unit j is denoted* w_{ji}

- Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers
- Until the termination condition is met, Do

  - For each $(\vec{x}, \vec{t})$, in training examples, Do

    Propagate the input forward through the network:
    1. Input the instance $\vec{x}$, to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

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

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

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

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

4. Update each network weight w_{ji}

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

Where

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

Derivation of the BACKPROPAGATION Rule

- Deriving the stochastic gradient descent rule: Stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error E_d with respect to this single example
- For each training example d every weight w_{ji} is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \qquad \text{.....equ. (1)}$$

where, E_d is the error on training example d, summed over all output units in the network

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

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

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables

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

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

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

o_j = the output computed by unit j

t_j = the target output for unit j

σ = the sigmoid function

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

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

derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule

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

notice that weight w_{ji} can influence the rest of the network only through net_j.

Use chain rule to write

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

$$= \frac{\partial E_d}{\partial net_j} x_{ji} \qquad \text{.....equ(2)}$$

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

Consider two cases in turn: the case where unit j is an ***output unit*** for the network, and the case where j is an ***internal unit (hidden unit)***.

Case 1: Training Rule for Output Unit Weights.

- 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

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \qquad \text{.....equ(3)}$$

To begin, consider just the first term in Equation (3)

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

The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when $k = j$. We therefore drop the summation over output units and simply set $k = j$.

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

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

$$= -(t_j - o_j) \qquad \text{.....equ(4)}$$

Next consider the second term in Equation (3). Since $o_j = \sigma(net_j)$, the derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function, which we have already noted is equal to $\sigma(net_j)(1 - \sigma(net_j))$. Therefore,

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

$$= o_j(1 - o_j) \qquad \text{.....equ(5)}$$

Substituting expressions (4) and (5) into (3), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\ o_j(1 - o_j) \qquad \text{.....equ(6)}$$

and combining this with Equations (1) and (2), we have the stochastic gradient descent rule for output units

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j)x_{ji} \qquad \text{.....equ(7)}$$

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 and denoted this set of units by ***Downstream(*** j).
- net_j can influence the network outputs only through the units in ***Downstream(j).*** Therefore, we can write

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

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

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

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

$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j) \qquad \text{.....equ (8)}$$

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

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

REMARKS ON THE BACKPROPAGATION ALGORITHM

1. Convergence and Local Minima

- The BACKPROPAGATION multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.
- Despite the lack of assured convergence to the global minimum error, BACKPROPAGATION is a highly effective function approximation method in practice.
- Local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases.

Common heuristics to attempt to alleviate the problem of local minima include:

1. Add a momentum term to the weight-update rule. Momentum can sometimes carry the gradient descent procedure through narrow local minima
2. Use stochastic gradient descent rather than true gradient descent
3. Train multiple networks using the same data, but initializing each network with different random weights

2. Representational Power of Feedforward Networks

What set of functions can be represented by feed-forward networks?

The answer depends on the width and depth of the networks. There are three quite general results are known about which function classes can be described by which types of Networks

1. Boolean functions – Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs
2. Continuous functions – Every bounded continuous function can be approximated with arbitrarily small error by a network with two layers of units
3. Arbitrary functions – Any function can be approximated to arbitrary accuracy by a network with three layers of units.

3. Hypothesis Space Search and Inductive Bias

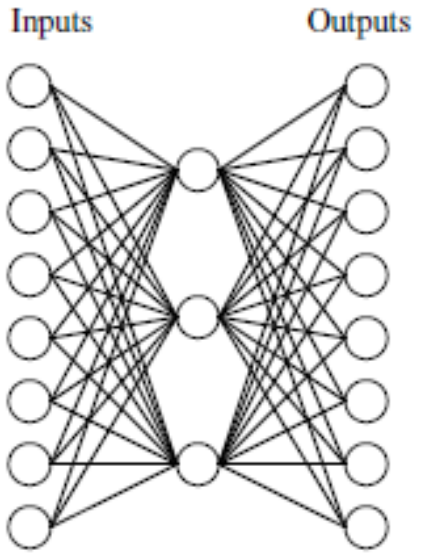
- Hypothesis space is the n-dimensional Euclidean space of the ***n*** network weights and hypothesis space is continuous.
- As it is continuous, E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis.
- It is difficult to characterize precisely the inductive bias of BACKPROPAGATION algorithm, because it depends on the interplay between the gradient descent search and the way in which the weight space spans the space of representable functions. However, one can roughly characterize it as smooth interpolation between data points.

4. Hidden Layer Representations

BACKPROPAGATION can define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.

Consider example, the network shown in below Figure

A network:



Learned hidden layer representation:

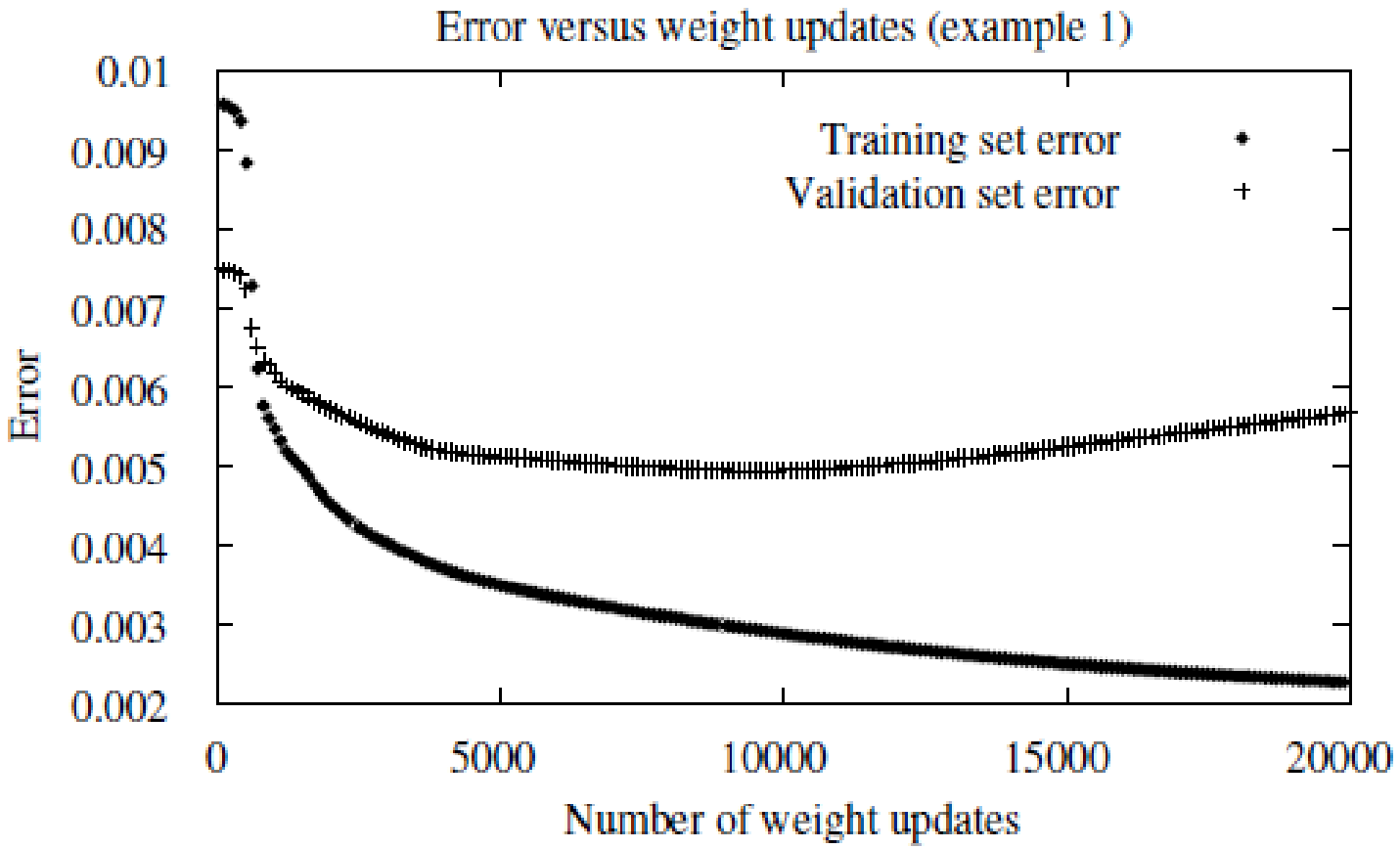
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

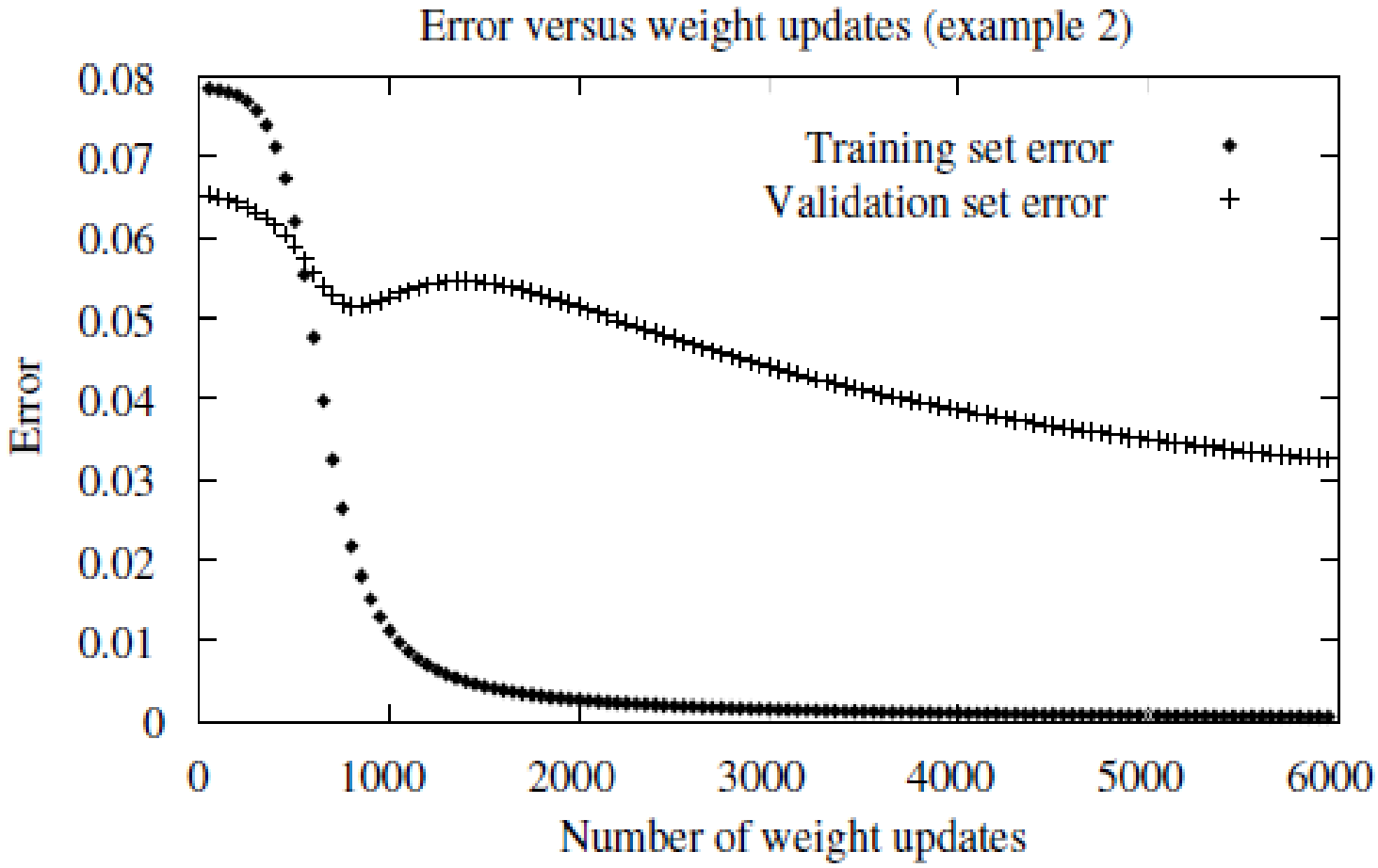
- Consider training the network shown in Figure to learn the simple target function f (x) = x, where x is a vector containing seven 0's and a single 1.
- The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this case is constrained to use only three hidden units. Therefore, the essential information from all eight input units must be captured by the three learned hidden units.
- When BACKPROPAGATION applied to this task, using each of the eight possible vectors as training examples, it successfully learns the target function. By examining the hidden unit values generated by the learned network for each of the eight possible input vectors, it is easy to see that the learned encoding is similar to the familiar standard binary encoding of eight values using three bits (e.g., 000,001,010,. . . , 111). The exact values of the hidden units for one typical run of shown in Figure.
- This ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning

5. Generalization, Overfitting, and Stopping Criterion

What is an appropriate condition for terminating the weight update loop?

- One choice is to continue training until the error E on the training examples falls below some predetermined threshold.
- To see the dangers of minimizing the error over the training data, consider how the error E varies with the number of weight iterations


Error versus weight updates (example 1)
Training set error
Validation set error
Error
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0
5000
10000
15000
20000
Number of weight updates


Error versus weight updates (example 2)
Training set error
Validation set error
Error
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
1000
2000
3000
4000
5000
6000
Number of weight updates

- Consider first the top plot in this figure. The lower of the two lines shows the monotonically decreasing error E over the training set, as the number of gradient descent iterations grows. The upper line shows the error E measured over a different validation set of examples, distinct from the training examples. This line measures the generalization accuracy of the network-the accuracy with which it fits examples beyond the training data.
- The generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease. How can this occur? This occurs because the weights are being tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples. The large number of weight parameters in ANNs provides many degrees of freedom for fitting such idiosyncrasies

Why does overfitting tend to occur during later iterations, but not during earlier iterations?

- By giving enough weight-tuning iterations, BACKPROPAGATION will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training sample.

THANK YOU

UNIT-5: ARTIFICIAL NEURAL NETWORKS

- Introduction
- Neural network representation
- Appropriate problems for Neural Network Learning
- Perceptron
- MultiLayer Networks and Back propagation Algorithm
- Remarks of Back propagation algorithm

Introduction: How can we answer this problems?

- Pattern recognition: Does that image contain a face?
- Classification problems: Is this product defective?
- Prediction: Given these symptoms, the patient has disease X
- Forecasting: predicting behavior of stock market
- Handwriting: is character recognized?
- Optimization: Find the shortest path for the .

- Neural Networks can be used to solve these problems

What led to the development of ANNs:

- **Neurobiological studies:**
  - How do nerves behave when stimulated by different magnitudes of electric current?
  - Is there a minimal threshold needed for nerves to be activated?
  - How do different nerve cells communicate among each other?

- **Psychological studies:**
  - How do animals learn, forget, recognize and perform various types of tasks?

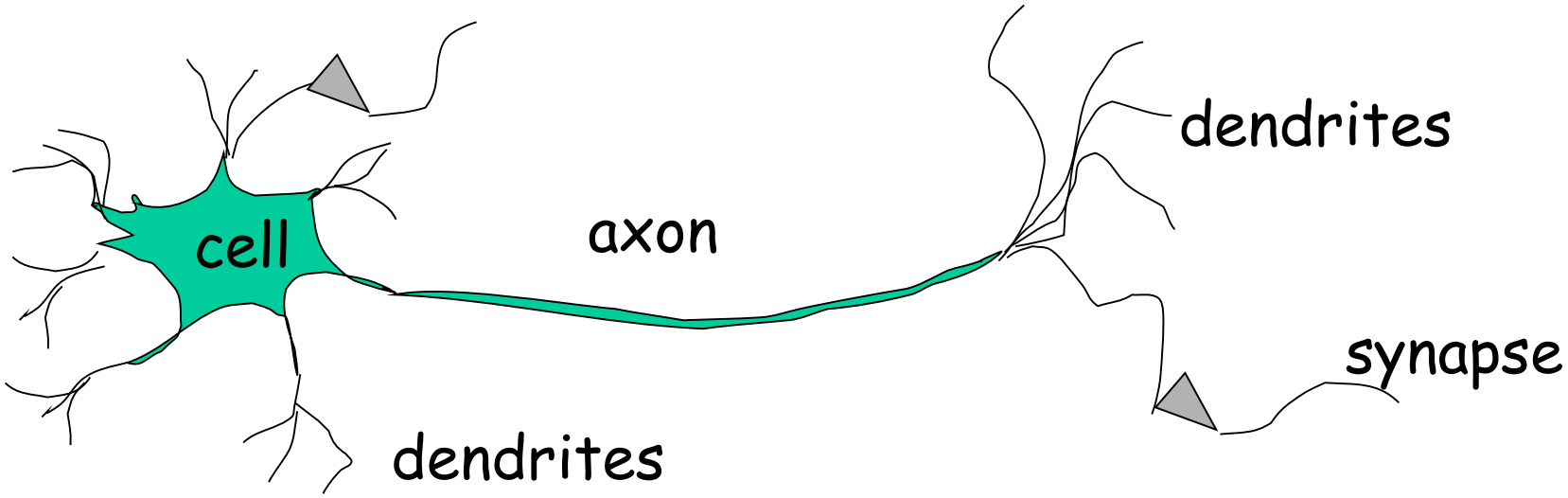
- **Psycho-physical:** experiments help to understand how individual neurons and groups of neurons work.

- **McCulloch and Pitts** : can we have an artificial network which does similar to human brain nerve cells?
  - introduced the first mathematical model of single neuron, widely applied in subsequent work.

Human Brain Biological Neurons:

- human information processing system consists of brain neuron: basic building block
  - cell that communicates information to and from various parts of body
- Simplest model of a neuron: considered as a threshold unit –a processing element (PE)
- Collects inputs & produces output if the sum of the input exceeds an internal threshold value

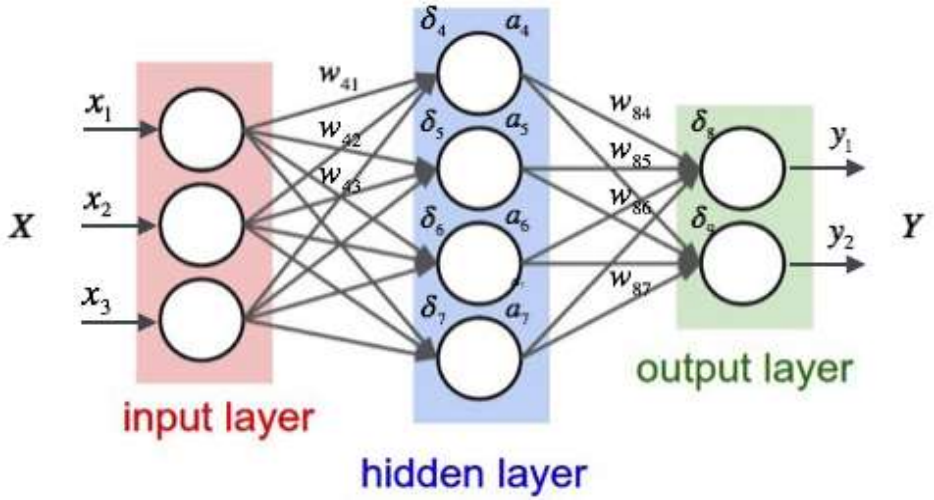
Human Brain Biological neurons:



What Is an Artificial Neural Network (ANN)?

- An artificial neural network (ANN) is the piece of a computing system designed to simulate the way the human brain analyzes and processes information.
- ANNs have self-learning capabilities that enable them to produce better results as more data becomes available.

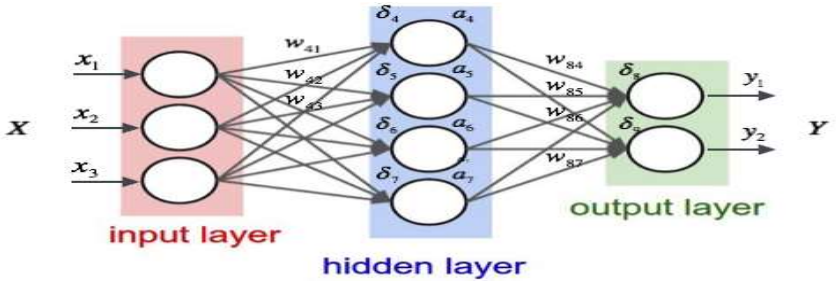
Structure of ANN:



Topological units of ANN:

- Neural Network.
  - Is a Mathematical model.
  - Inspired by Human biological neurons and now leading in solving many real world problems
- ANN topology Units include:
  - Neurons.
  - Layers.
  - Connecting Links.
  - Initial weights(on links)

    and bias and activation function
  - Inputs/outputs

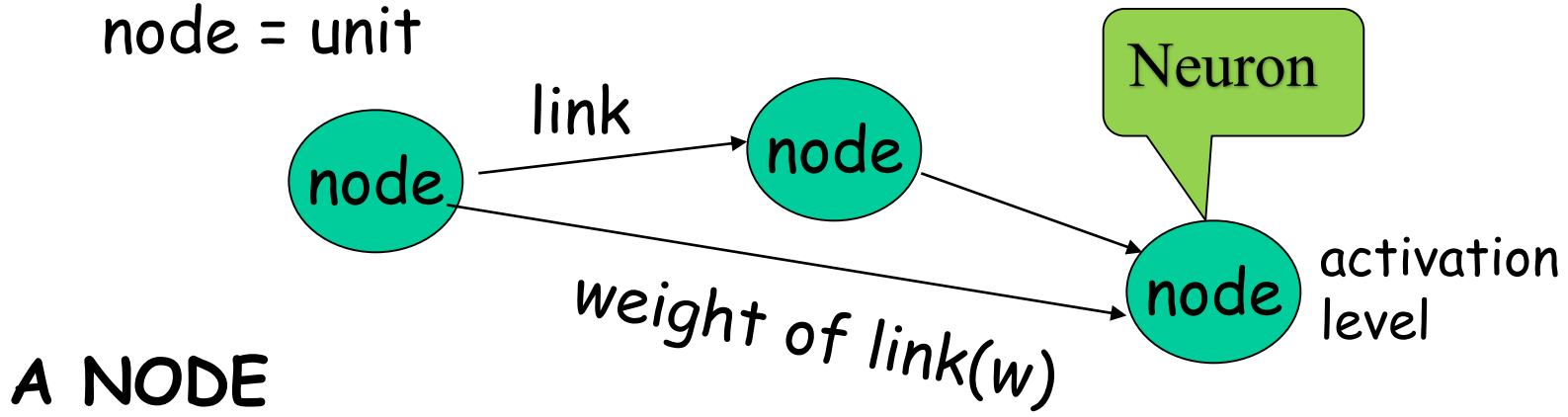


Properties of ANNs

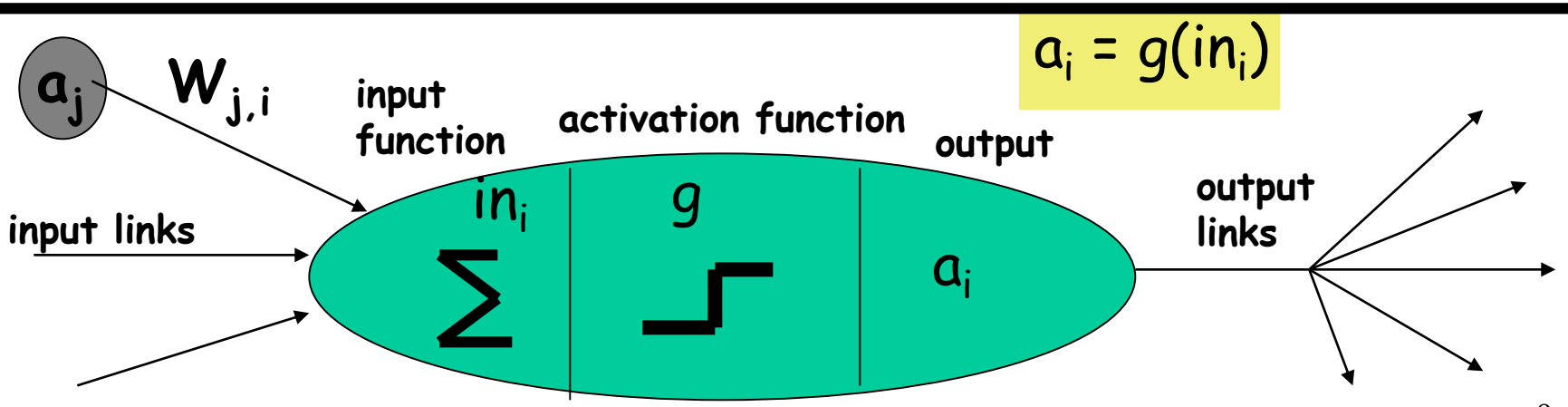
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically
- Input is a high-dimensional discrete or real-valued (e.g, sensor input)

Neurons:

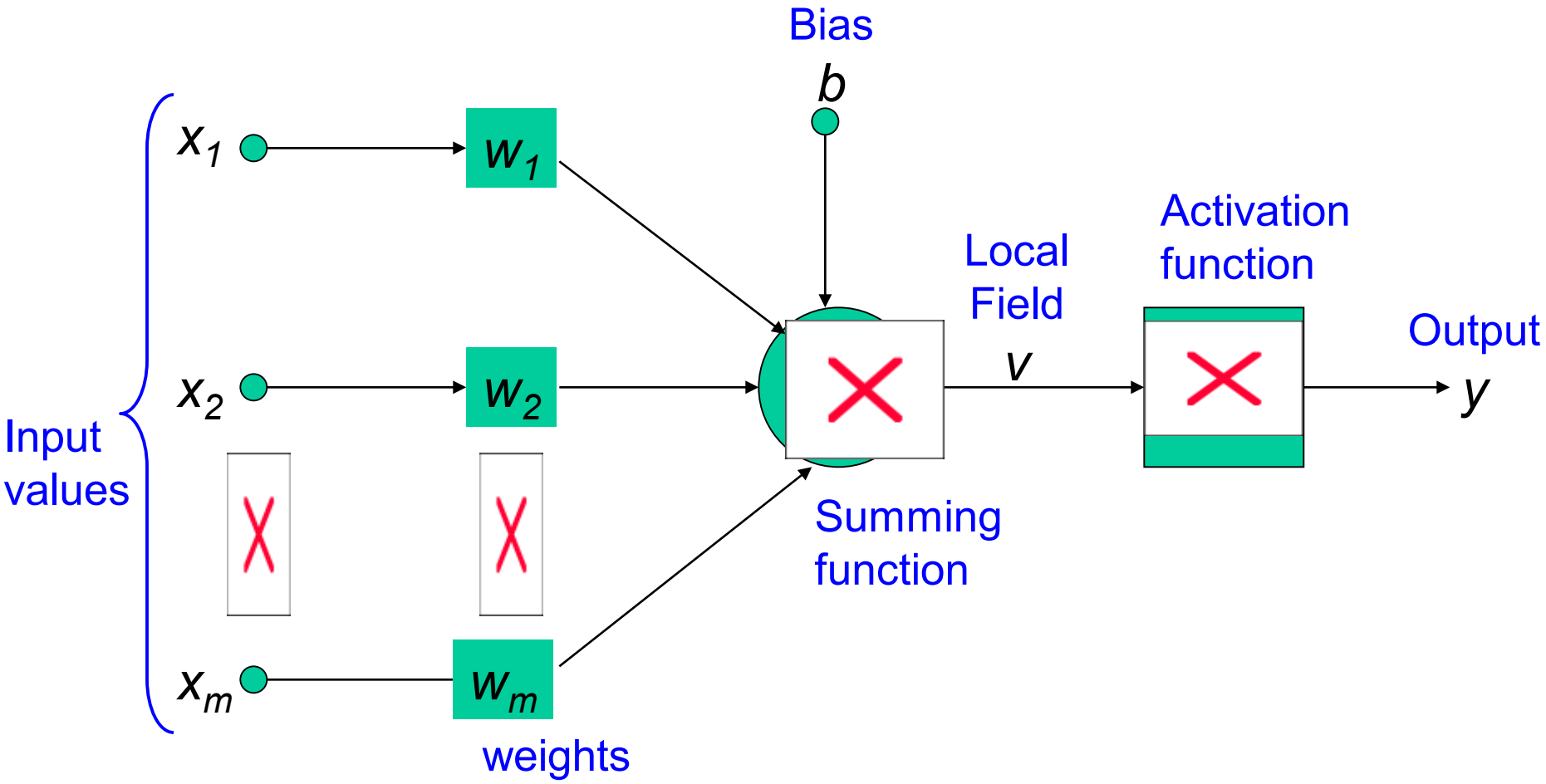
An ANN has hundreds or thousands of artificial processing units called neurons, which are interconnected by nodes.



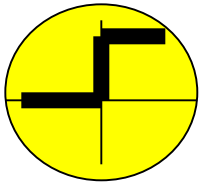
A NODE



ANN Architecture:



g = Activation functions for units

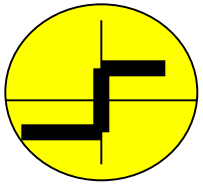


Step function

(Linear Threshold Unit)

step(x) = 1, if x >= threshold

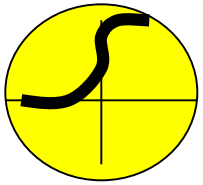
0, if x < threshold



Sign function

sign(x) = +1, if x >= 0

-1, if x < 0



Sigmoid function

sigmoid(x) = $1/(1+e^{-x})$

Purpose of activation function:

- Activation function is added for Hidden and O/P layers
- The **purpose** of an **activation function** is to add some kind of non-linear property to the input.
- Without the **activation functions**, the only mathematical operation during the forward propagation would be dot-products between an input vector and a weight matrix: which is linear

Why non linear Activation function: (ex: sigmoid)

- Linear functions always do exhibit a constant slope.
- Slope as we know pictures the rate of change of the predictor variable (*y*) with respect to the independent variable (*x*).
- Real-world problems emphasize on finding the points of minimal or maximal change: may be minimization of loss or maximization of gain.
- If we take linear activation functions, these changes are constant and cannot be distinguishable between points of minimal and maximal changes.
- Nonlinear activation functions have gradients that vary between various points.
- Based on these gradients descent or accent, we can address all the minimization and maximization problems.

.

Why non linear Activation function: (ex: sigmoid)

- The nonlinear exponential function e^X has a range between [0, ∞] and has an infinite range.
- The gradients for the function e^X exist between [0, ∞].
- To find the optimal gradients, it is quite difficult within this infinite range.
- Hence, such functions though nonlinear cannot be taken as activation functions.
- Hence, the needs for activation functions which show active gradients within shorter intervals are needed.
- One such is the sigmoid activation function which will analyze the gradients between a small [0, 1] interval

Role of Bias:

Bias is one more important parameter in an ANN. The output from a neuron is of
the form:
O/P = Σ (weights*Inputs) + Bias .

- This is more similar to a linear function: $Y = mX + C$; here a positive or a negative constant C shifts Y right or left; even the bias does the same.
- Weights of the NN are used to steer the steepness of activation function; bias will shift this steepness to the right or left of the curve.
- These shifts are indications of the delayed triggering of the activation function.
- Thus, bias is used to preserve the trigging active nature of the activation cell even for zero inputs.
- Bias is critical for successful learning of the networks; with them the networks can learn different outputs for inputs.

APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN is appropriate for problems with the following characteristics :

- 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

Architectures of Artificial Neural Networks

An artificial neural network can be divided into three parts (layers), which are known as:

- ***Input layer:*** This layer is responsible for receiving information (data), signals, features, or measurements from the external environment. These inputs are usually normalized within the limit values produced by activation functions
- ***Hidden, intermediate, or invisible layers:*** These layers are composed of neurons which are responsible for extracting patterns associated with the process or system being analysed. These layers perform most of the internal processing from a network.
- ***Output layer :*** This layer is also composed of neurons, and thus is responsible for producing and presenting the final network outputs, which result from the processing performed by the neurons in the previous layers.

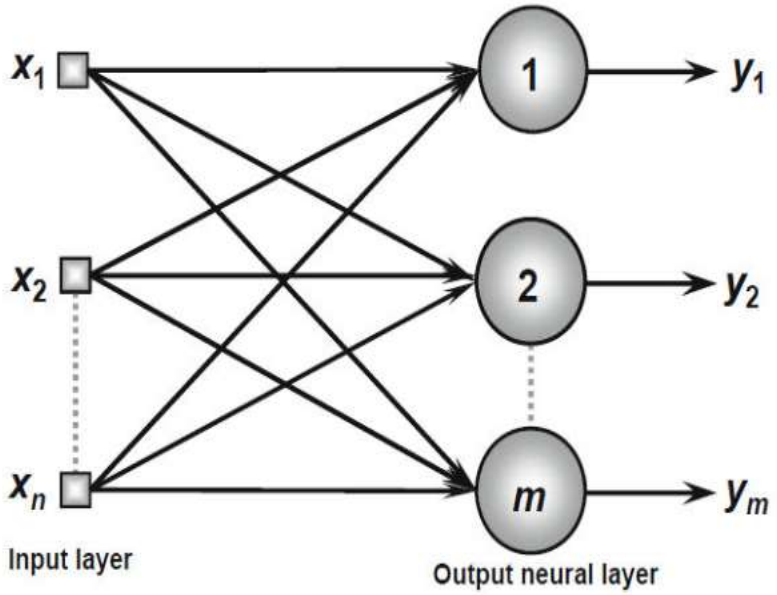
Architectures of Artificial Neural Networks

The main architectures of artificial neural networks, considering the neuron disposition, how they are interconnected and how its layers are composed, can be divided as follows:

1. Single-layer feedforward network
2. Multi-layer feedforward networks
3. Recurrent or Feedback networks
4. Mesh networks

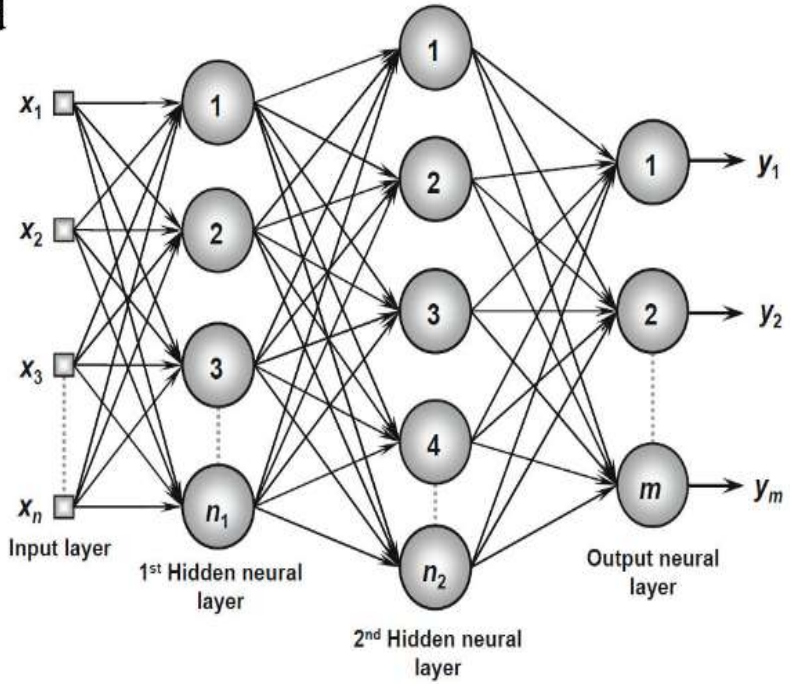
Single-Layer Feedforward Architecture

- This artificial neural network has just one input layer and a single neural layer, which is also the output layer.
- Figure illustrates a simple-layer feedforward network composed of n inputs and m outputs.
- The information always flows in a single direction (thus, unidirectional), which is from the input layer to the output layer



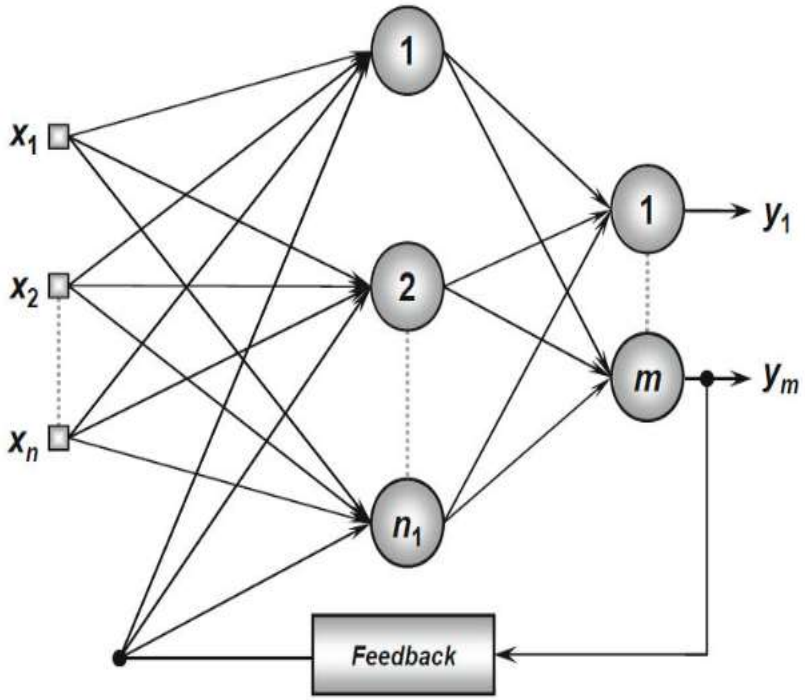
Multi-Layer Feedforward Architecture

- This artificial neural feedforward networks with multiple layers are composed of one or more hidden neural layers.
- Figure shows a feedforward network with multiple layers composed of one input layer with n sample signals, two hidden neural layers consisting of n_1 and n_2 neurons respectively, and, finally, one output neural layer composed of m neurons representing the respective output values of the problem being analyzed



Recurrent or Feedback Architecture

- In these networks, the outputs of the neurons are used as feedback inputs for other neurons.
- Figure illustrates an example of a Perceptron network with feedback, where one of its output signals is fed back to the middle layer.



PERCEPTRONS

- Perceptron is a single layer neural network.
- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise
- Given inputs x_1 through x_n, the output $O(x_1, \ldots, x_n)$ computed by the perceptron is

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

- where each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.
- $-w_0$ is a threshold that the weighted combination of inputs $w_1 x_1 + \ldots + w_n x_n$ must surpass in order for the perceptron to output a 1.

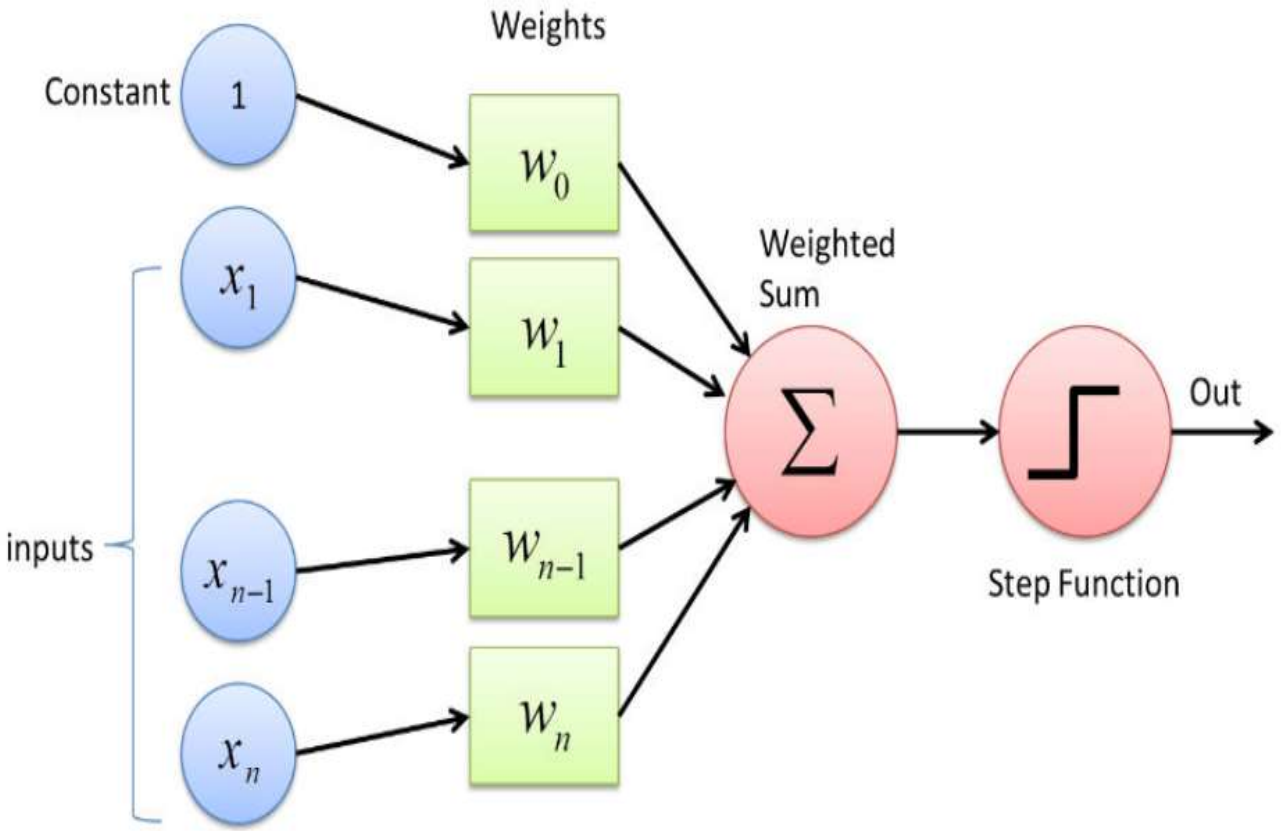
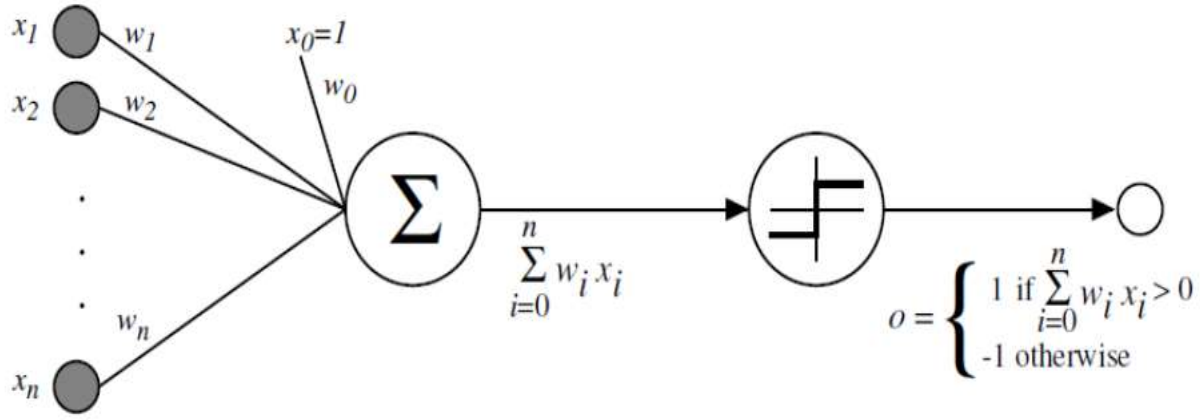


Fig : Perceptron

Perceptron:

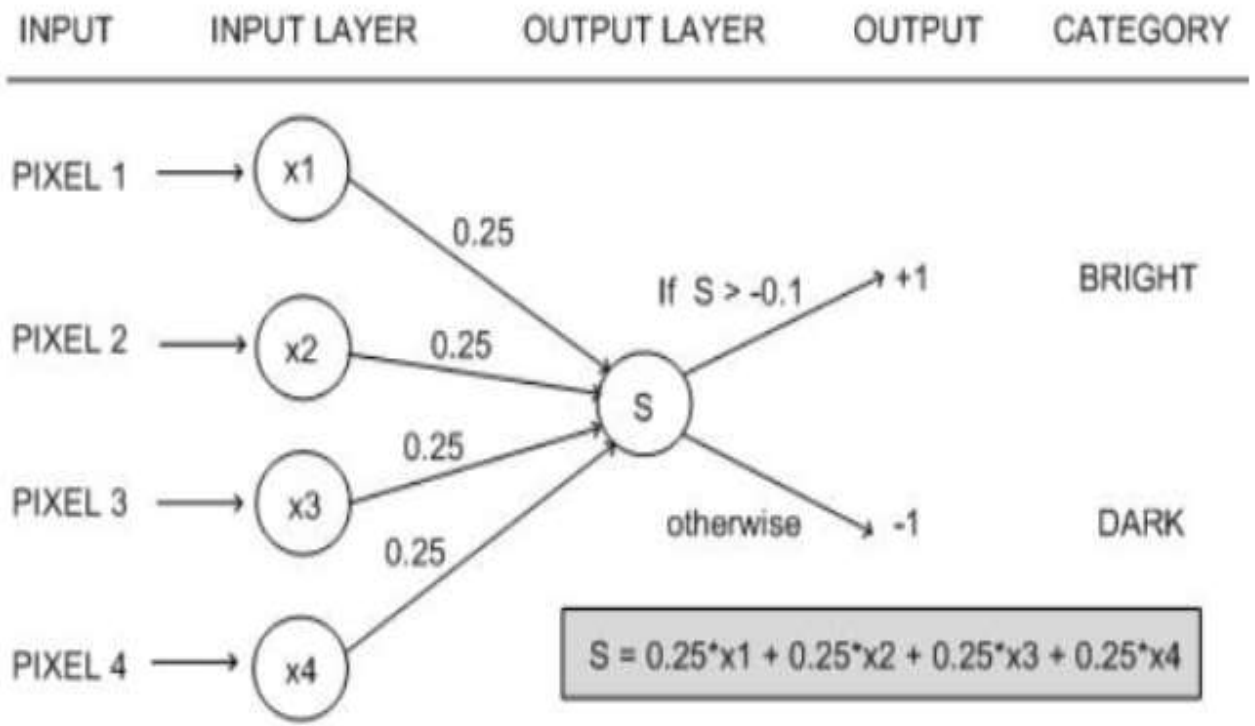


$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

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

Example network:



The Perceptron Training Rule

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

To learn an acceptable weight vector

- Begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example.
- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.
- Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input x_i according to the rule.

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

Where,

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

Here,
t is the target output for the current training example
o is the output generated by the perceptron
η is a positive constant called the ***learning rate***

- The role of the ***learning rate*** is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases

Drawback: 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.

Gradient Descent and the Delta Rule

- If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.
- The key idea behind the ***delta rule*** is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

To understand the delta training rule, consider the task of training an unthresholded perceptron. That is, a linear unit for which the output O is given by

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

$$O(\vec{x}) = (\vec{w} \cdot \vec{x}) \qquad \text{equ. (1)}$$

To derive a weight learning rule for linear units, specify a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples.

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

Where,

- D is the set of training examples,
- t_d is the target output for training example d,
- o_d is the output of the linear unit for training example d
- E [$\vec{w}$] is simply half the squared difference between the target output t_d and the linear unit output o_d, summed over all training examples.

GRADIENT DESCENT algorithm for training a linear unit

GRADIENT-DESCENT($training\_examples, \eta$)

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

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

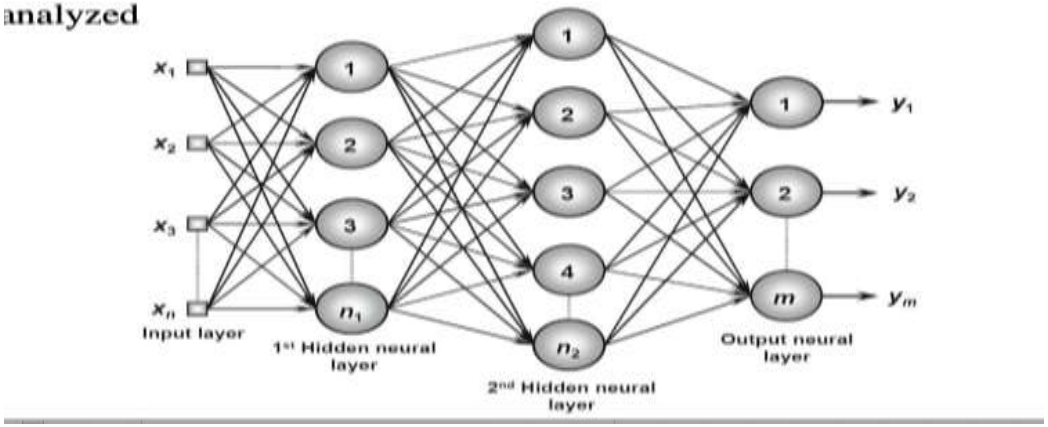
To summarize, the gradient descent algorithm for training linear units is as follows:

- Pick an initial random weight vector.
- Apply the linear unit to all training examples, then compute Δw_i for each weight according to Equation (7).
- Update each weight w_i by adding Δw_i, then repeat this process

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

Multilayer networks learned by the **BACKPROPACATION** algorithm are capable of expressing a rich variety of nonlinear decision surfaces

analyzed



A Differentiable Threshold Unit

- Sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.
- The sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a continuous function of its input.
- More precisely, the sigmoid unit computes its output ***O*** as

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

Where,

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

σ is the sigmoid function

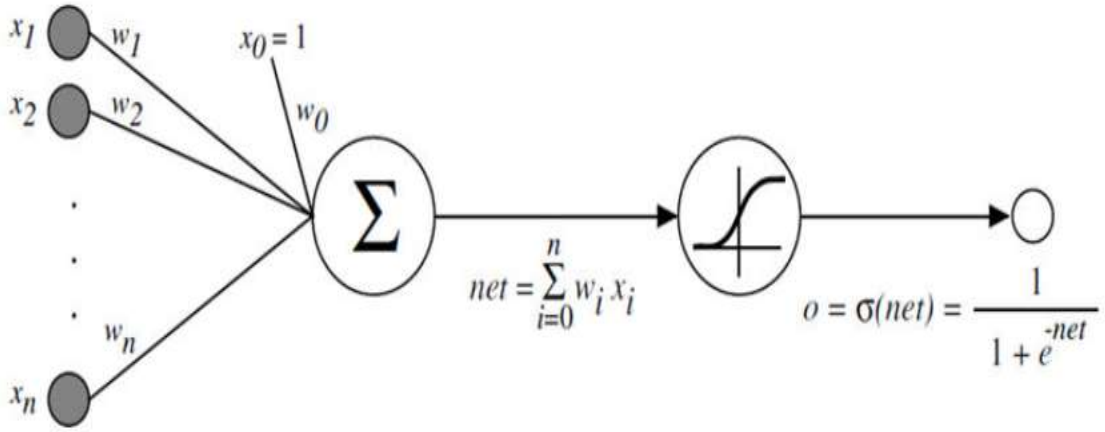


Figure: A Sigmoid Threshold Unit

$\sigma(y)$ is the sigmoid function

$$\frac{1}{1+e^{-y}}$$

Nice property: $\frac{d\sigma(y)}{dy} = \sigma(y)(1-\sigma(y))$

The BACKPROPAGATION Algorithm

- The BACKPROPAGATION Algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.
- In BACKPROPAGATION algorithm, we consider networks with multiple output units rather than single units as before, so we redefine E to sum the errors over all of the network output units.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \quad \text{.....equ. (1)}$$

where,

- ***outputs*** **-** is the set of output units in the network
- t_{kd} and O_{kd} - the target and output values associated with the k^{th} output unit
- d - training example

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

Each training example is a pair of the form $(\vec{x}, \vec{t})$, *where* $(\vec{x})$ *is the vector of network input values,* $(\vec{t})$ *and is the vector of target network output values.*
η *is the learning rate (e.g., .05).* n_i *is the number of network inputs,* n_{hidden} *the number of units in the hidden layer, and* n_{out} *the number of output units.*
The input from unit i into unit j is denoted x_{ji}, *and the weight from unit i to unit j is denoted* w_{ji}

- Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers
- Until the termination condition is met, Do

  - For each $(\vec{x}, \vec{t})$, in training examples, Do

    Propagate the input forward through the network:
    1. Input the instance $\vec{x}$, to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

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

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

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

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

4. Update each network weight w_{ji}

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

Where

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

Number of training pairs needed?

Difficult question. Depends on the problem, the training examples, and network architecture. However, a good rule of thumb is:

Where W = No. of weights; P = No. of training pairs, e = error rate

For example, for e = 0.1, a net with 80 weights will require 800 training patterns to be assured of getting 90% of the test patterns correct (assuming it got 95% of the training examples correct).

How long should a net be trained?

- The objective is to establish a balance between correct responses for the training patterns and correct responses for new patterns. (a balance between memorization and generalization).
- If you train the net for too long, then you run the risk of overfitting.
- In general, the network is trained until it reaches an acceptable error rate (e.g., 95%)

Implementing Backprop – Design Decisions

1. Choice of *learning rate*
2. Network architecture
   a) How many Hidden layers? how many hidden units per a layer?
   b) How should the units be connected? (e.g., Fully, Partial, using domain knowledge
3. Stopping criterion – when should training stop?

Backpropagation

- Performs gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Minimizes error over training examples
  - Will it generalize well to subsequent examples
    - Guarding against overfitting needed
- Training can take thousands of iterations (epocs) → Slow!
- Using network after training is very fast

Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum…
- Add momentum
- Stochastic gradient descent
- Train Multiple Nets with different initial weights

Back-propagation Using Gradient Descent

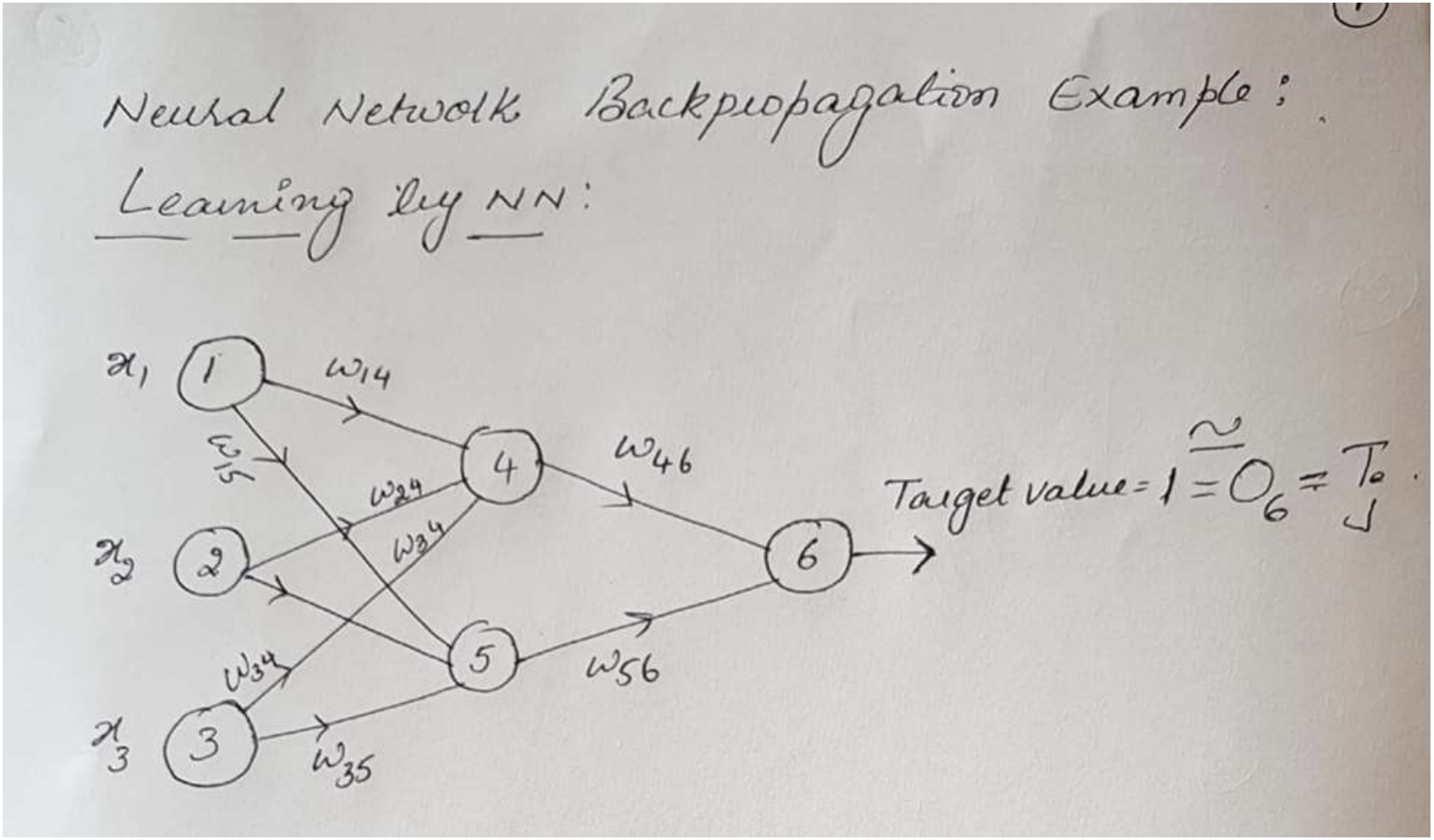
- Advantages
  - Relatively simple implementation
  - Standard method and generally works well
- Disadvantages
  - Slow and inefficient
  - Can get stuck in local minima resulting in sub-optimal solutions

Learning rate

- Ideally, each **weight** should have its own learning rate (*extra notes on tricks for BP*)

- As a substitute, each neuron or each layer could have its own rate

Back propagation Example:



First training tuple $X = (x_1, x_2, x_3) = (1, 0, 1)$

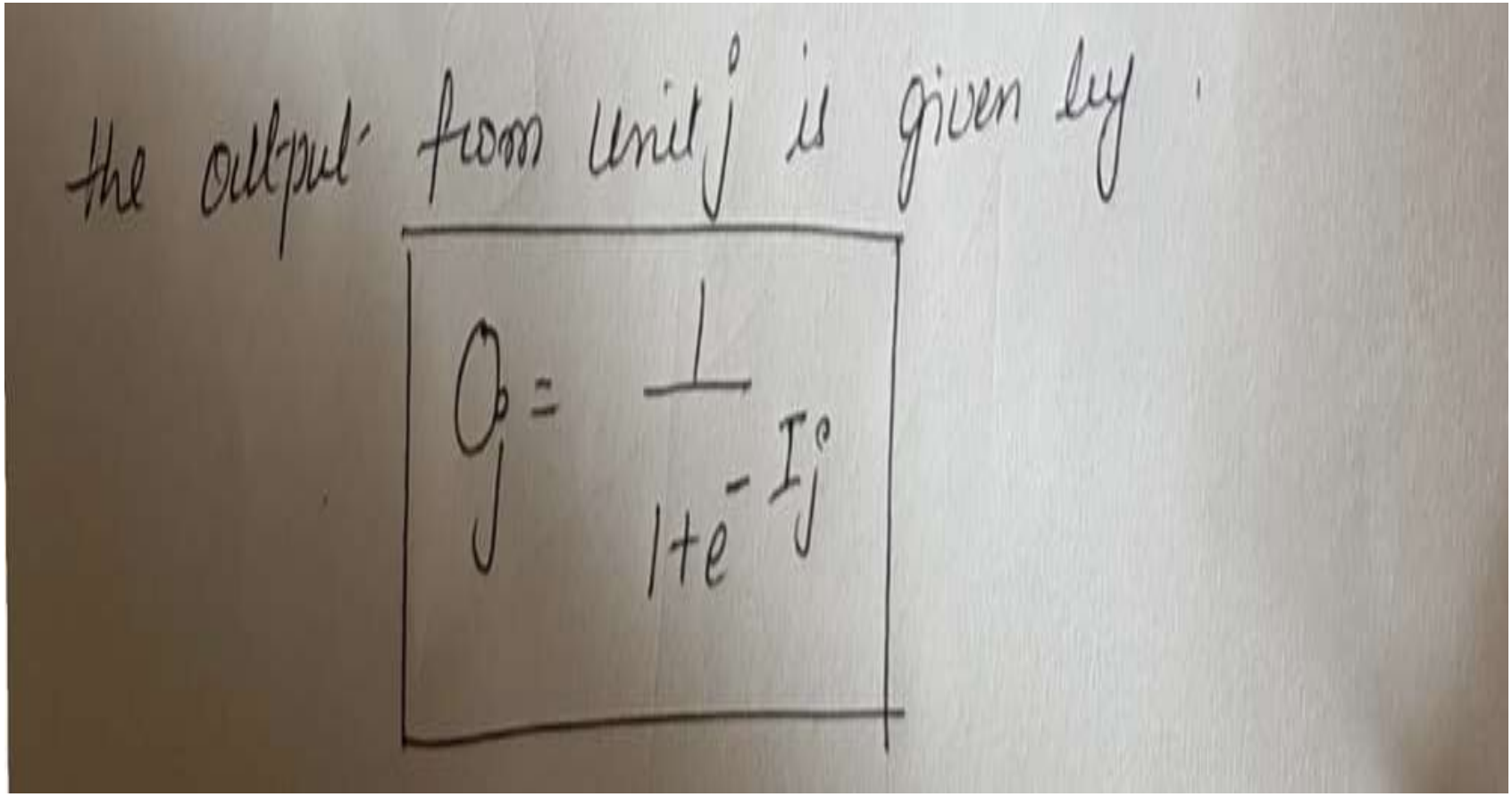
Initial I/ps, weights and Bias.

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	Bias θ_4	Bias θ_5	Bias θ_6
1	0	1	0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1

For unit j in hidden/output layer, I/p is given by

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

O_i – output of unit i.


the output from unit j is given by:
$O_j = \frac{1}{1+e^{-I_j}}$

Net I/O calculations: (2)

Unit j	I/P (I_j)	O/P (O_j)
4	$0.2+0-0.5^{-0.4} = -0.7$	$1/(1+e^{0.7}) = 0.332$
5	$-0.3+0+0.2+0.2 = 0.1$	$1/(1+e^{-0.1}) = 0.525$
6	$(-0.3)(0.332)-(0.2)(0.525)+0.1 = -0.105$	$1/(1+e^{0.105}) = 0.474$

$O_6 = 0.474$.

$O_6 = 0.474$.

Target $O_6 = 1$

Error $= 1 - 0.474 = 0.526$

This error is resultant from all the Internal nodes.

In Backpropagation algorithm this error is Backpropagated through all the layer and weights / bias are updated So as to reduce the error of the next iteration.

Error calculation:

for a unit j in the o/p layer**, the error Err_j is given by:

$$Err_j = O_j(1-O_j)(T_j - O_j)$$

for a unit j in the hidden layer, the error Err_j is given by

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

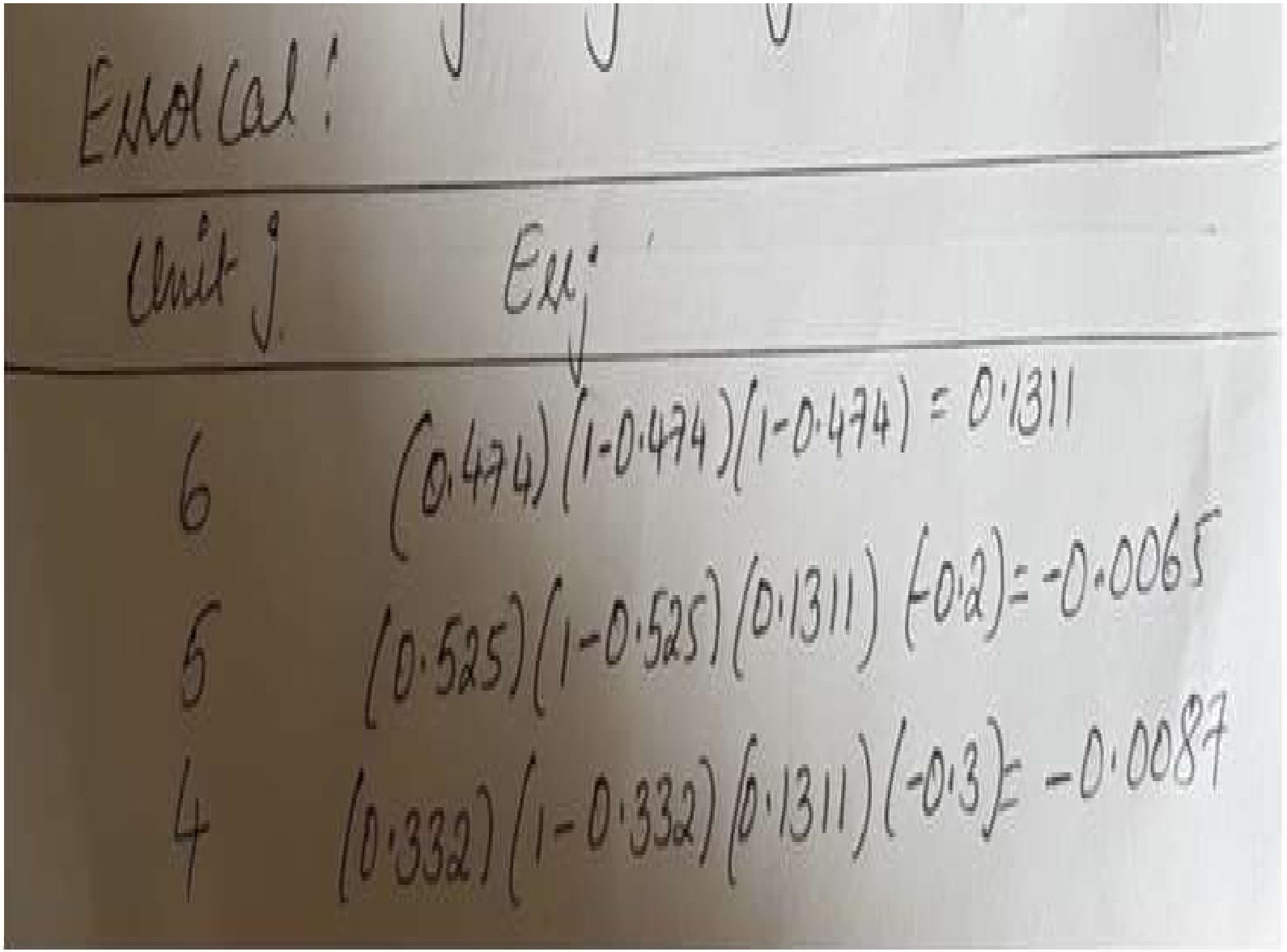
the weights and biases are updated using

$$\Delta w_{ij} = (l)\, Err_j O_i$$

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

$$\Delta \theta_j = (l)\, Err_j$$

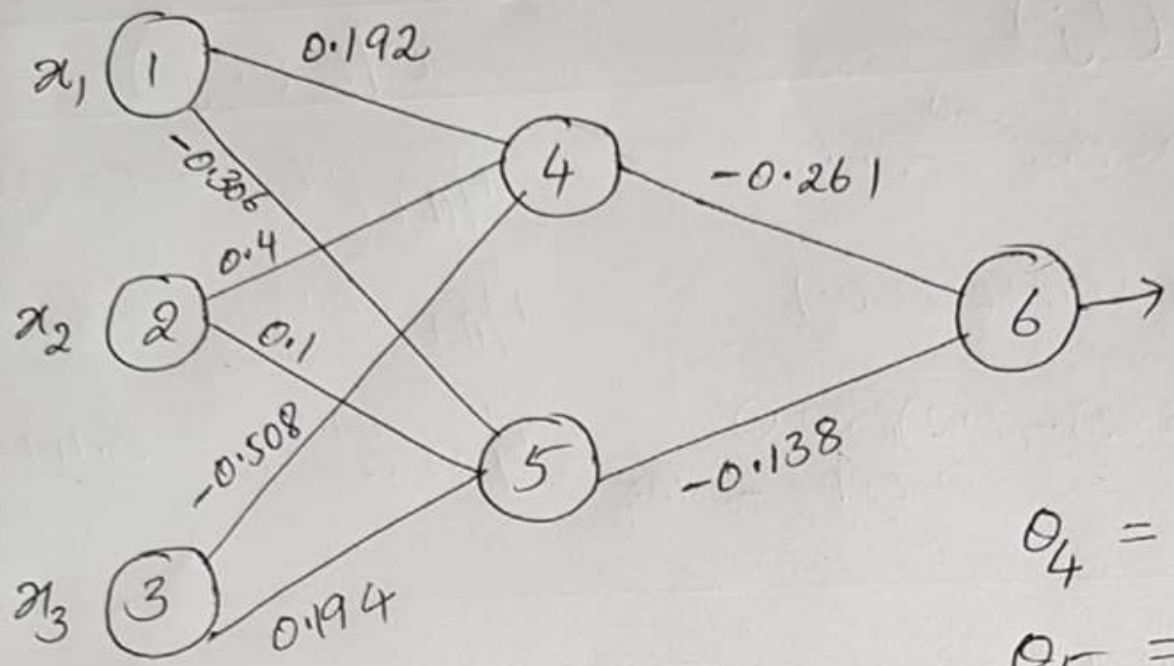
$$\theta_j = \theta_j + \Delta \theta_j$$


Error cal:
Unit j
Err j
6
(0.474)(1-0.474)(1-0.474) = 0.1311
6
(0.525)(1-0.525)(0.1311)(-0.2) = -0.0065
4
(0.332)(1-0.332)(0.1311)(-0.3) = -0.0087

updating weights and Bias: (4)

weight/Bias	updated value
w_{46}	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
w_{56}	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
w_{14}	$0.2 + (0.9)(-0.0087)(1) = 0.192$
w_{15}	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
w_{24}	$0.4 + (0.9)(-0.0087)(0) = 0.4$
w_{25}	$0.1 + (0.9)(-0.0065)(0) = 0.1$
w_{34}	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
w_{35}	$0.2 + (0.9)(-0.0065)(1) = 0.194$
θ_6	$0.1 + (0.9)(0.1311) = 0.218$
θ_5	$0.2 + (0.9)(-0.0065) = 0.194$
θ_4	$-0.4 + (0.9)(-0.0087) = -0.408$

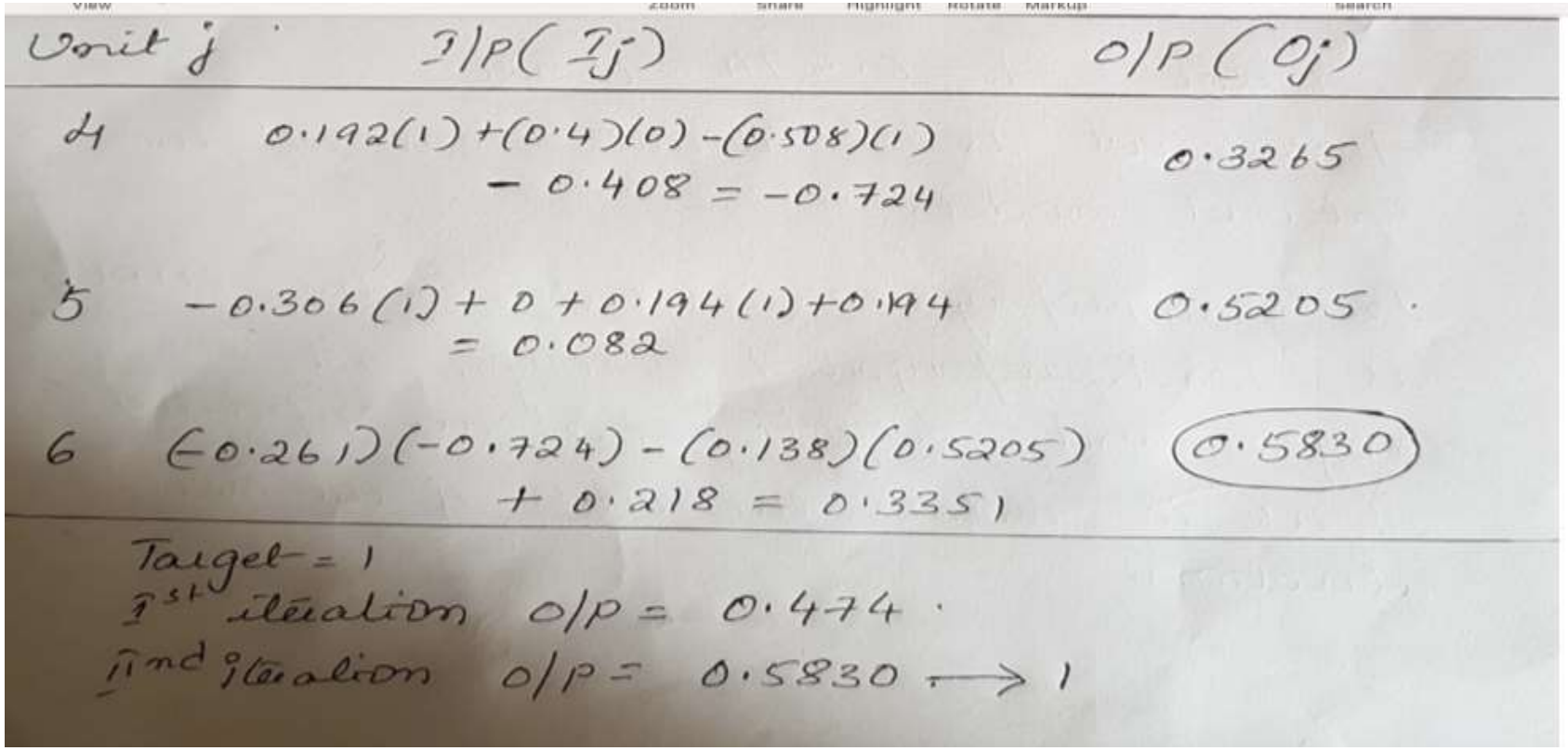
Updated Network | Iteration -2 (5)



$\theta_4 = -0.408$

$\theta_5 = 0.194$

$\theta_6 = 0.218$



The o/p from 2nd iteration is 0.5830

There is (1- 0.5830) error .
Further iterations will converge the o/p to target=1

Setting the parameter values

- How are the weights initialized?
- Do weights change after the presentation of each pattern or only after all patterns of the training set have been presented?
- How is the value of the learning rate chosen?
- When should training stop?
- How many hidden layers and how many nodes in each hidden layer should be chosen to build a feedforward network for a given problem?
- How many patterns should there be in a training set?
- How does one know that the network has learnt something useful?

REMARKS ON THE BACKPROPAGATION ALGORITHM

1. Convergence and Local Minima

- The BACKPROPAGATION multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.
- Despite the lack of assured convergence to the global minimum error, BACKPROPAGATION is a highly effective function approximation method in practice.
- Local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases.

Neural Networks: Advantages

- **Distributed representations**
- **Simple computations**
- **Robust with respect to noisy data**
- **Robust with respect to node failure**
- **Empirically shown to work well for many problem domains**
- **Parallel processing**

Neural Networks: Disadvantages

- **Training is slow**
- **Interpretability is hard**
- **Network topology layouts ad hoc**
- **Can be hard to debug**
- **May converge to a local, not global, minimum of error**
- **Not known how to model higher-level cognitive mechanisms**
- **May be hard to describe a problem in terms of features with numerical values**

Applications

- Classification:
  - Image recognition
  - Speech recognition
  - Diagnostic
  - Fraud detection
  - Face recognition ..
- Regression:
  - Forecasting (prediction on base of past history)
  - Forecasting e.g., predicting behavior of stock market
- Pattern association:
  - Retrieve an image from corrupted one
  - …
- Clustering:
  - clients profiles
  - disease subtypes
  - …

Applications

- Pronunciation: NETtalk program (Sejnowski & Rosenberg 1987) is a neural network that learns to pronounce written text: maps characters strings into phonemes (basic sound elements) for learning speech from text

- Handwritten character recognition:a network designed to read zip codes on hand-addressed envelops

- ALVINN (Pomerleau) is a neural network used to control vehicles steering direction so as to follow road by staying in the middle of its lane