Artificial Neural Networks

Prepared By

Dr.Seema Patil
Assistant Professor,
Department of CSE,
BMSCE,Bangalore

Introduction & Definition:

- Artificial neural network is a computing system that is designed to simulate/ work in a way how human brain works, analyzes & processes the information.
- ANN performs the task similar to how brain works.
- 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) is inspired by the observation that biological learning systems are built of very complex webs of interconnected neurons.
- ANNs are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs and produces a single real-valued output.
- Human brain consists of billions of cells called neurons: basic building blocks that communicate information to and from various parts of body.
- Human brain consists of 10^{11} neurons, each connected to 10^4 neurons.
- Similar to human body, artificial neural networks are built with a set of input, output units.
- ANN has thousands of artificial neurons called processing units. These units are input & output units.

Facts of Human Neurobiology

- Number of neurons ~ 10^{11}
- Connection per neuron ~ 10^{4-5}
- Neuron switching time ~ 0.001 second or 10^{-3} (Computer: 10^{-10}sec)
- 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

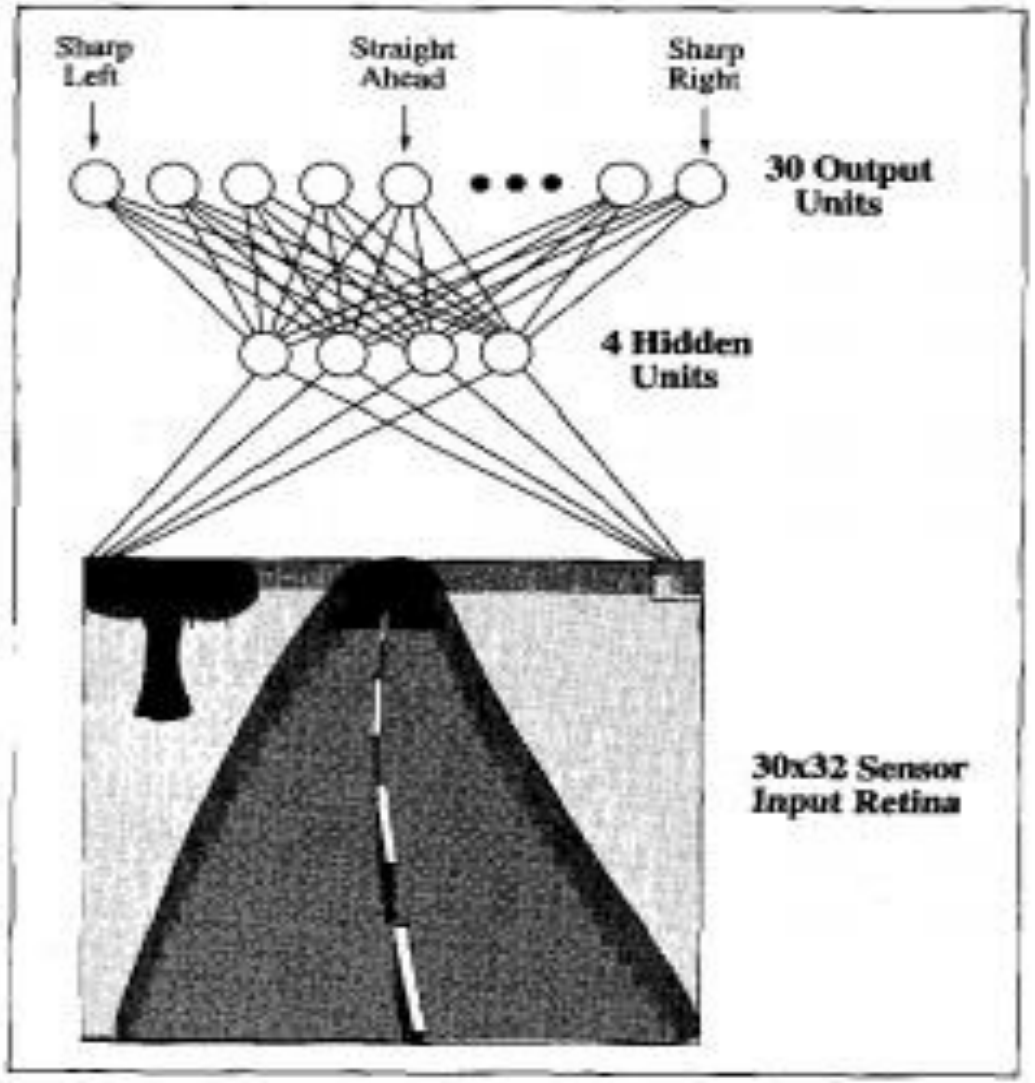
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically
- Input is a high-dimensional discrete or real-valued (e.g, sensor input)

Neural Network Representations (ALVINN)

Neural Network Representations (ALVINN)

- A prototypical example of ANN learning is provided by system 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 system is trained to mimic steering commands of a human driver.
- Also, the system is trained using road images.
- The network ***output*** is the direction in which the vehicle should move.
- Each output unit corresponds to a particular steering direction.

Neural Network Representations (ALVINN)



Appropriate problems for neural network learning

- **Instances are represented by many attribute-value pairs**: The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example.

- **The target function output may be discrete-valued, real-valued, or a vector valued attributes**: In the ALVINN system the output is a vector of 30 attributes, each corresponding to a steering direction. The value of each output is some real number between 0 and 1.

- **The training examples may contain errors**: ANN learning methods are quite robust to noise in the training data.

Appropriate problems for neural network learning

- **Long training times are acceptable**: Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.

- **Fast evaluation of learned target function may be required:** ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.

- **The ability of humans to understand the learned target function is not important:** The weights learned by neural networks are often difficult for humans to interpret.

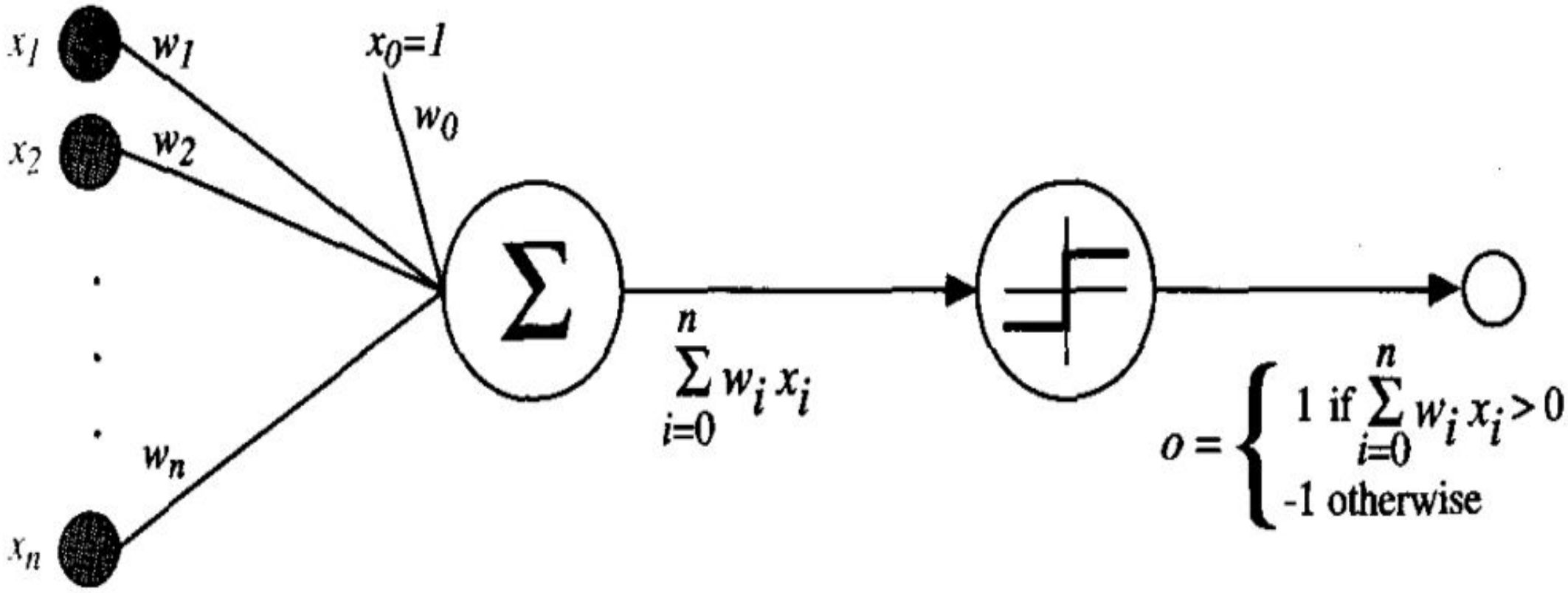
Perceptrons

A perceptron unit is used to build the ANN system.
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.
More precisely, given inputs x1 through xn, the output o(x1, . . . , xn) computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

where each wi is a real-valued constant, or weight, that determines the contribution of input xi to the perceptron output.

Perceptron Training Rule



Perceptron Training Rule

One way to learn an acceptable weight vector is to 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 wi associated with input xi according to the rule.

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

where

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

Perceptron Training Rule Algorithm

```
Perceptron_training_rule (X, η)
  initialize w (wi <- an initial (small) random value)
  repeat
      for each training instance (x, tx) ∈ X
            compute the real output ox = Activation(Summation(w.x))
            if (tx ≠ ox)
                  for each wi
                        wi <- wi + Δwi
                        Δwi <- η (tx - ox)xi
                  end for
            end if
      end for
  until all the training instances in X are correctly classified
  return w
```

Logical AND Gate-Perceptron Training Rule

Weights w1 = 1.2, w2 = 0.6, Threshold = 1 and Learning Rate n = 0.5 are given

For Training Instance 1:

A=0, B=0 and Target = 0

wi.xi = 0*1.2 + 0*0.6 = 0

This is not greater than the threshold of 1, so the output = 0.

Here the target is same as calculated output.

For Training Instance 2:

A=0, B=1 and Target = 0

wi.xi = 0*1.2 + 1*0.6 = 0.6

This is not greater than the threshold of 1, so the output = 0.

Here the target is same as calculated output.

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

Logical AND Gate-Perceptron Training Rule (Contd..)

Weights w1 = 1.2, w2 = 0.6, Threshold = 1 and Learning Rate n = 0.5 are given

For Training Instance 3:

A=1, B=0 and Target = 0

wi.xi = 1*1.2 + 0*0.6 = 1.2

This is greater than the threshold of 1, so the output = 1.

Here the target does not match with the calculated output.

Hence we need to update the weights.

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

Now, After updat

$$wi = wi + n(t - o)xi$$

$$w1 = 1.2 + 0.5(0 - 1)1 = 0.7$$

$$w2 = 0.6 + 0.5(0 - 1)0 = 0.6$$

1 and Learning Rate n = 0.5

Logical AND Gate-Perceptron Training Rule (Contd..)

w1 = 0.7, w2 = 0.6 Threshold = 1 and Learning Rate n = 0.5

For Training Instance 1:

A=0, B=0 and Target = 0

wi.xi = 0*0.7 + 0*0.6 = 0

This is not greater than the threshold of 1, so the output = 0

Here the target is same as calculated output.

For Training Instance 2:

A=0, B=1 and Target = 0

wi.xi = 0*0.7 + 1*0.6 = 0.6

This is not greater than the threshold of 1, so the output = 0.

Here the target is same as calculated output.

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

Logical AND Gate-Perceptron Training Rule (Contd..)

w1 = 0.7, w2 = 0.6 Threshold = 1 and Learning Rate n = 0.5

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

For Training Instance 3:

A=1, B=0 and Target = 0

wi.xi = 1*0.7 + 0*0.6 = 0.7

This is not greater than the threshold of 1, so the output = 0.

Here the target is same as calculated output.

For Training Instance 4:

A=1, B=1 and Target = 1

wi.xi = 1*0.7 + 1*0.6 = 1.3

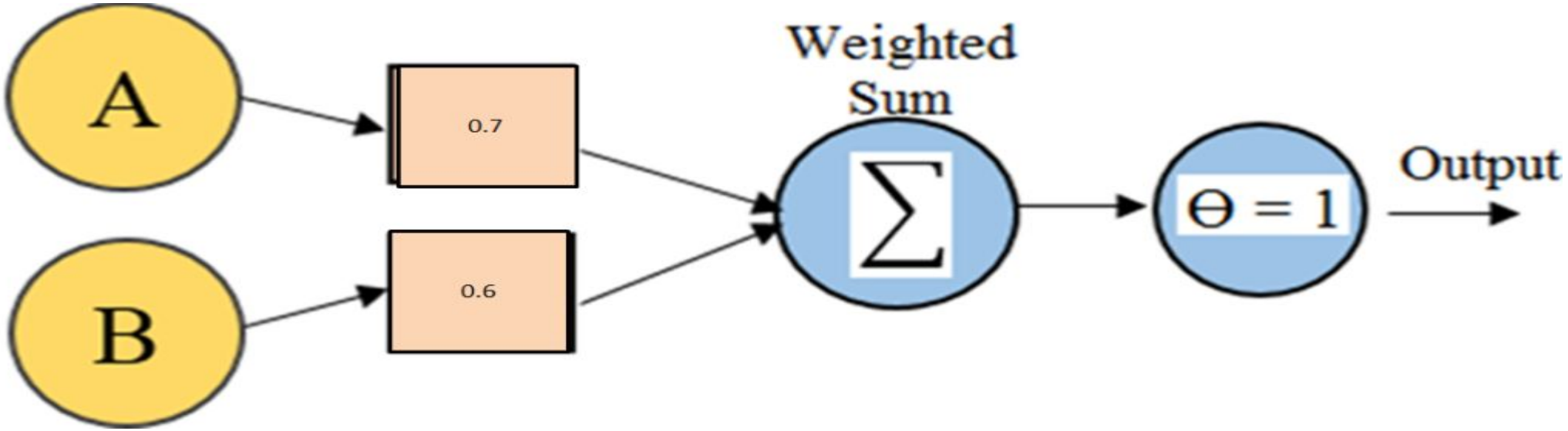
This is greater than the threshold of 1, so the output = 1. Here the target is same as calculated output.

Since, we have classified all the training examples correctly with the updated weights.

Hence the final weights are w1=0.7 and w2=0.6, Threshold=1 and Learning Rate n=0.5.

Logical AND Gate-Perceptron Training Rule (Contd..)

Final Perceptron Network for AND Gate, where A & B are inputs, 0.7 and 0.6 are the learned weights wrt. to A & B resp. with threshold as 1.



Logical OR Gate-Perceptron Training Rule

Weights w1 = 0.6, w2 = 0.6, Threshold = 1 and Learning Rate n = 0.5 are given

For Training Instance 1:

A=0, B=0 and Target = 0

wi.xi = 0*0.6 + 0*0.6 = 0

This is not greater than the threshold of 1, so the output = 0

Here the target is same as calculated output.

For Training Instance 2:

A=0, B=1 and Target = 1

wi.xi = 0*0.6 + 1*0.6 = 0.6

This is not greater than the threshold of 1, so the output = 0.

Here the target does not match with calculated output.

A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

Logical OR Gate-Perceptron Training Rule (Contd..)

Need to update the weights

$$wi = wi + n(t - o)xi$$

$$w1 = 0.6 + 0.5(1 - 0)0 = 0.6$$

$$w2 = 0.6 + 0.5(1 - 0)1 = 1.1$$

Now d=1 and Learning

A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

For Training Instance 1:

A=0, B=0 and Target = 0

wi.xi = 0*0.6 + 0*1.1 = 0

This is not greater than the threshold of 1, so the output = 0. Here the target is same as calculated output.

Logical OR Gate-Perceptron Training Rule (Contd..)

Weights w1 = 0.6, w2 = 1.1, Threshold=1 and Learning Rate n=0.5 are given

A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

For Training Instance 2:

A=0, B=1 and Target = 1

wi.xi = 0*0.6 + 1*1.1 = 1.1

This is not greater than the threshold of 1, so the output = 0.

Here the target is same as calculated output.

For Training Instance 3:

A=1, B=0 and Target = 1

wi.xi = 1*0.6 + 0*1.1 = 0.6

This is not greater than the threshold of 1, so the output = 0.

Here the target does not match with calculated output.

Logical OR Gate-Perceptron Training Rule (Contd..)

Need to update the weights

$$wi = wi + n(t - o)xi$$

$$w1 = 0.6 + 0.5(1 - 0)1 = 1.1$$

$$w2 = 1.1 + 0.5(1 - 0)0 = 1.1$$

A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

Now, Weights w1=1.1, w2 = 1.1, Threshold=1 and Learning [illegible]

For Training Instance 1:

A=0, B=0 and Target = 0

wi.xi = 0*2.2 + 0*1.1 = 0

This is not greater than the threshold of 1, so the output = 0. Here the target is same as calculated output.

For Training Instance 2:

A=0, B=1 and Target = 1

wi.xi = 0*1.1 + 1*1.1 = 1.1

This is not greater than the threshold of 1, so the output = 0. Here the target is same as calculated output.

Logical OR Gate-Perceptron Training Rule (Contd..)

Weights w1=1.1, w2 = 1.1, Threshold=1 and Learning Rate n=0.5 are given

For Training Instance 3:

A=1, B=0 and Target = 1

wi.xi = 1*1.1 + 0*1.1 = 1.1

This is not greater than the threshold of 1, so the output

Here the target is same as calculated output.

For Training Instance 4:

A=1, B=1 and Target = 1

wi.xi = 1*1.1 + 1*1.1 = 2.2

This is greater than the threshold of 1, so the output = 1. Here the target is same as calculated output.

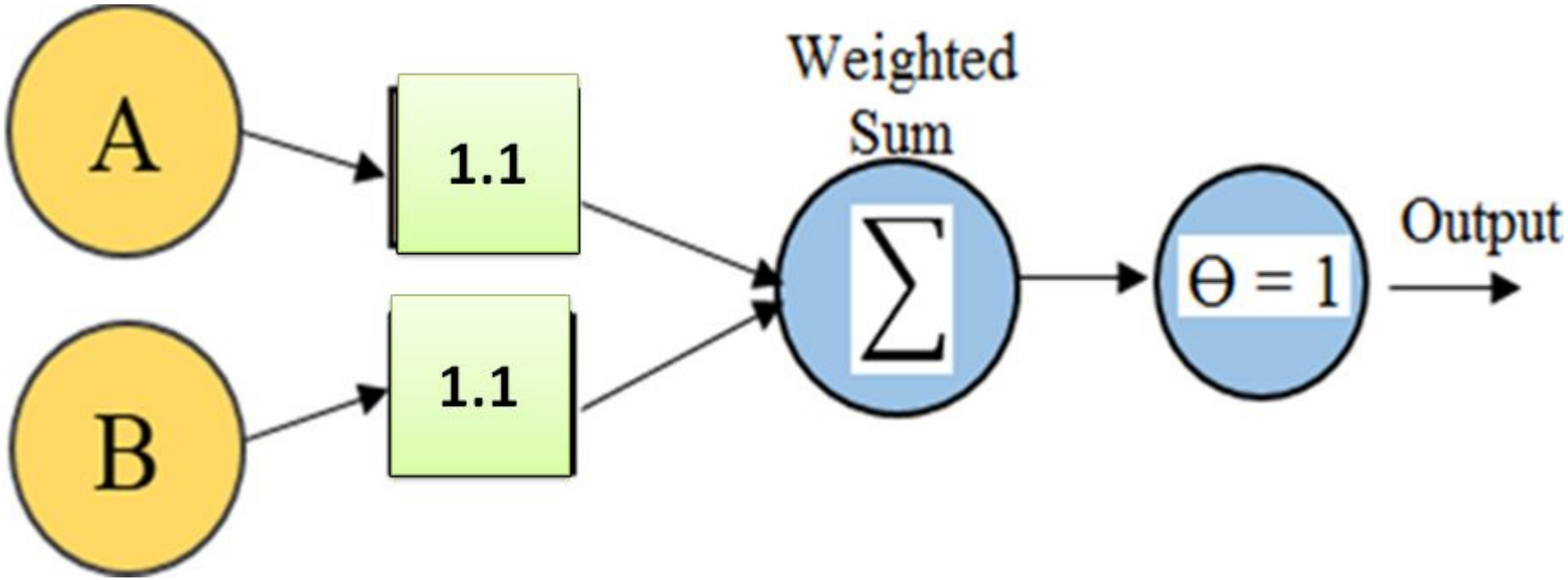
Since, we have classified all the training examples correctly with the updated weights.

Hence, final weights are w1=0.7 and w2=0.6, Threshold=1& Learning Rate n=0.5.

A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

Logical OR Gate-Perceptron Training Rule (Contd..)

Final Perceptron Network for OR Gate, where A & B are inputs, 1.1 and 1.1 are the learned weights wrt. to A & B resp. with threshold as 1.



Perceptron Training Rule (Contd..)

Drawback of Perceptron Training Rule Procedure:

Perceptron rule procedure will converge and finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable.

PERCEPTRONS

- 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.

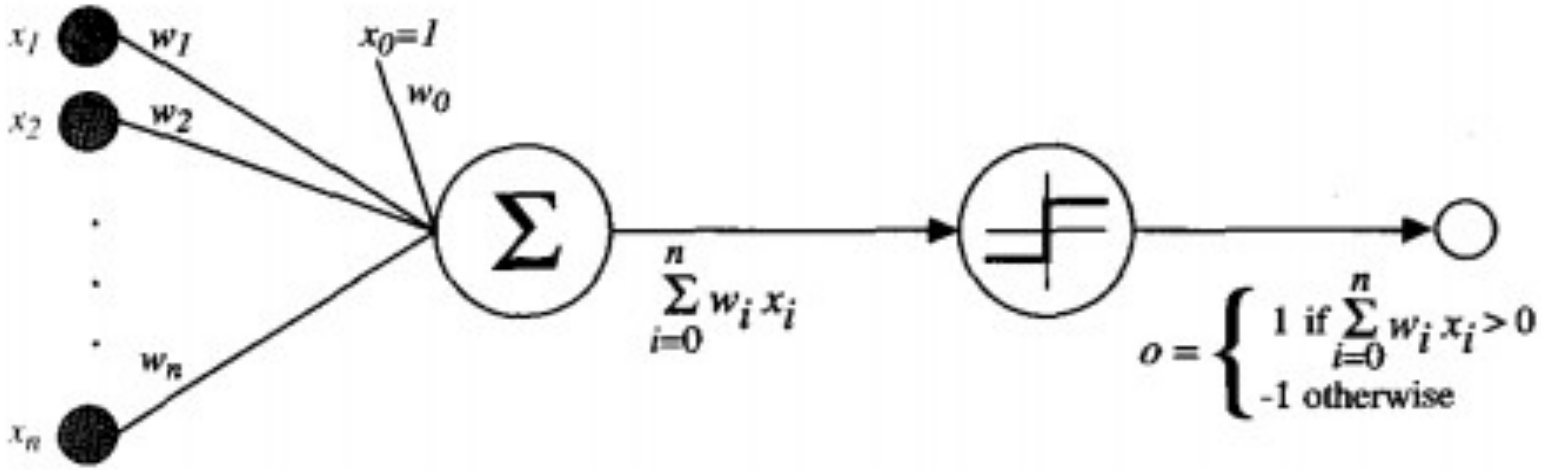


FIGURE 4.2
A perceptron.

Representational Power of Perceptrons

- We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points)

- The perceptron outputs 1 for instances lying on one side of the hyperplane and outputs -1 for instances lying on the other side, as illustrated in fig below.

- The equation for this decision hyperplane is w.x= 0.

- Of course, some sets of positive and negative examples cannot be separated by any hyperplane.

- Those that can be separated are called linearly separable sets of examples.

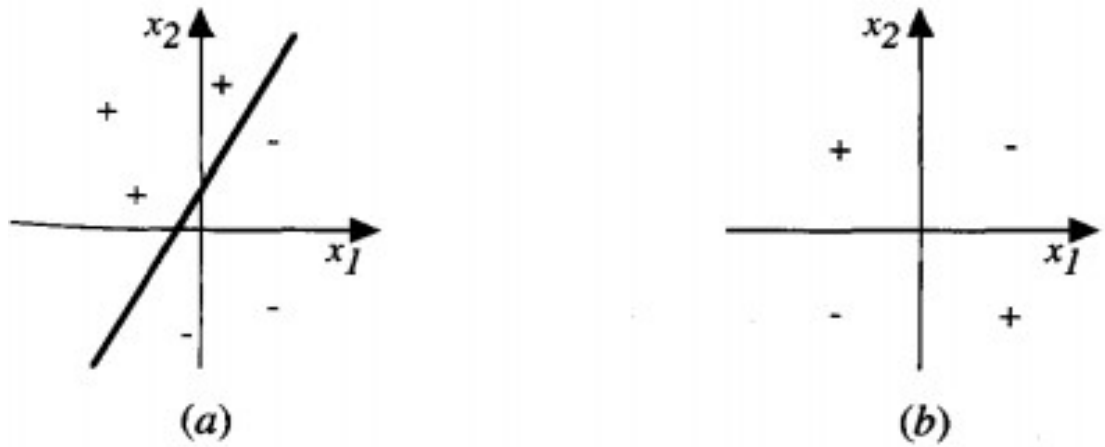


FIGURE 4.3
The decision surface represented by a two-input perceptron. (*a*) A set of training examples and the decision surface of a perceptron that classifies them correctly. (*b*) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). 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.
- For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights wo = -0.8 and wl = w2 = 0.5.
- This perceptron can be made to represent the OR function instead by altering the threshold to wo = -.3.
- In fact, AND and OR can be viewed as special cases of m-of-n functions: that is, functions where at least m of the n inputs to the perceptron must be true.
- The OR function corresponds to m = 1 and the AND function to m = n.
- Perceptrons can represent all of the primitive boolean functions AND, OR, NAND and NOR .
- Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if xl != x2.

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***

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 exampled,
- 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.

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 $\boldsymbol{E}$ with respect to each of the $\boldsymbol{w_i}$

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

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

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 thisprocess

Multi-Layer Perceptron (MLP)

An MLP is composed of

- One input layer
- One or more layers of LTUs(linear threshold units), called hidden layers
- And one final layer of LTUs called the output layer
- The layers close to the input layer are usually called the *lower layers.*
- The Layers close to the outputs are usually called the *upper layers.*
- The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a *feedforward neural network* (FNN).

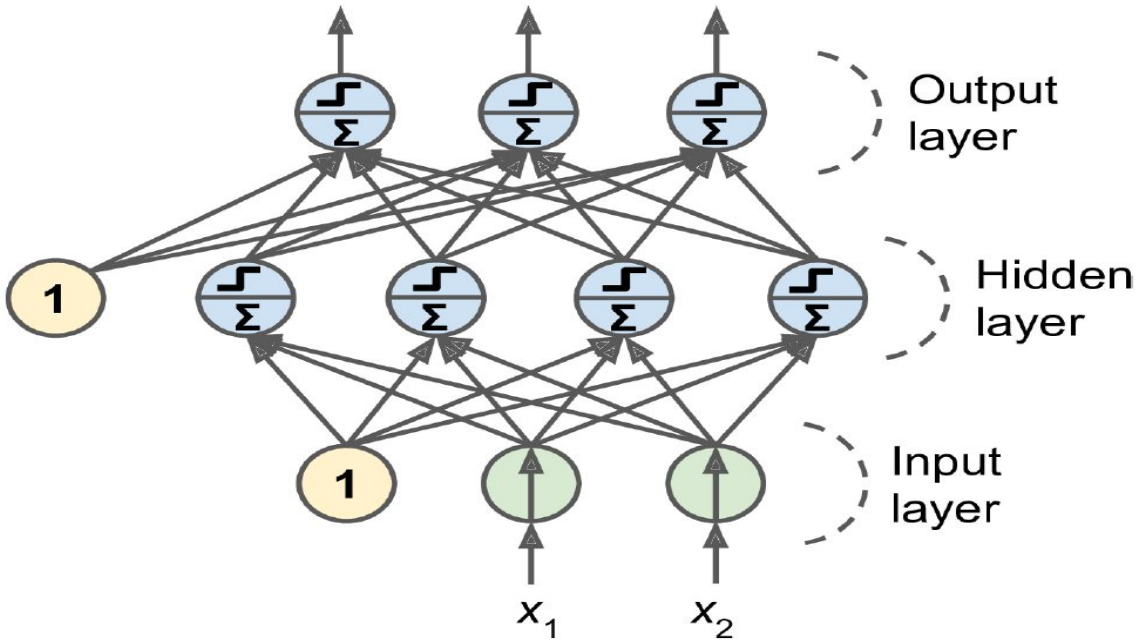


Figure 10-7. Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)

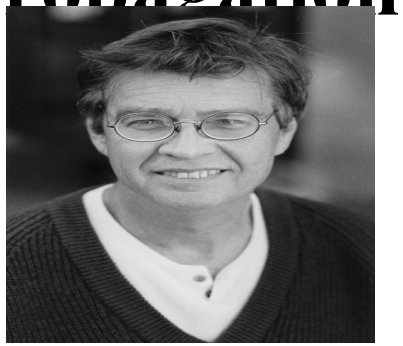
Multi-Layer Perceptron (MLP)

- Every layer except the output layer includes a bias neuron and is fully connected to the next layer.

Deep Neural Network

- When an ANN has two or more hidden layers, it is called a **deep neural network** (**DNN**)

Multi-Layer Perceptron and Backpropagation

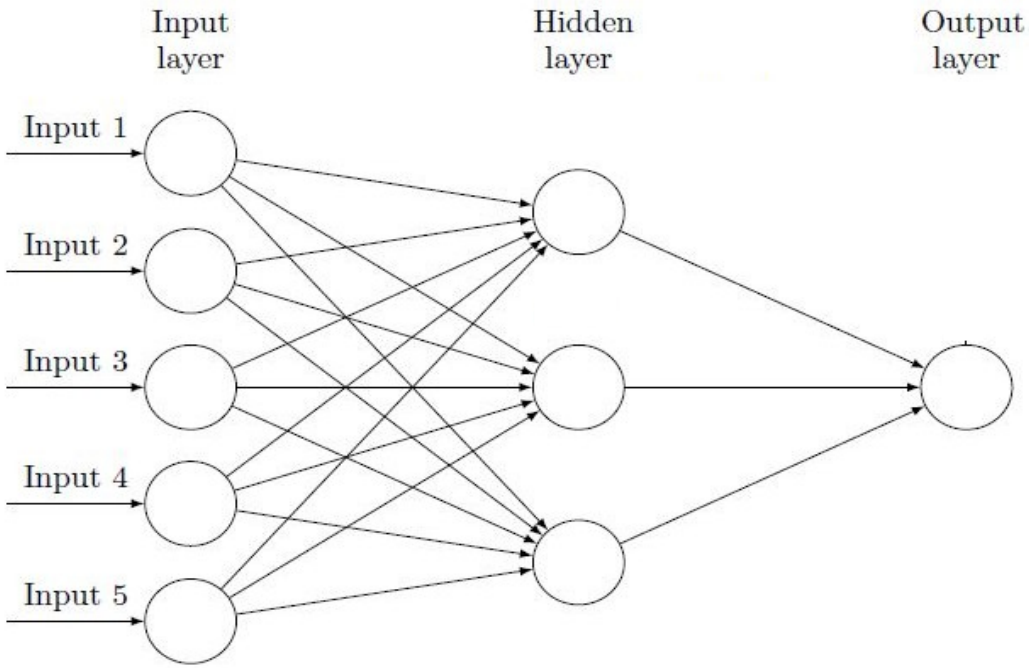


- For many years researchers struggled to find a way to train MLPs, without success
- In **1986**, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a **groundbreaking** paper introducing the **backpropagation** training algorithm.
- In short, it is Gradient Descent for computing the gradients automatically.
- In just two passes through the network (one forward, one backward), the backpropagation algorithm is able to compute the gradient of the network's error with regard to every single model parameter.
- In other words, it can find out how each connection weight and each bias term should be tweaked in order to reduce the error.
- Once it has these gradients, it just performs a regular Gradient Descent step, and the whole process is repeated until the network converges to the solution.
- Automatically computing gradients is called *automatic differentiation*, or *autodiff*.
- Backpropagation uses *reverse-mode autodiff.*

Multi-Layer Perceptron and Backpropagation

What is Backpropagation?
For each training instance

- The algorithm feeds it to the network
- And computes the output of every neuron in each consecutive layer
- This is the forward pass, just like when making predictions

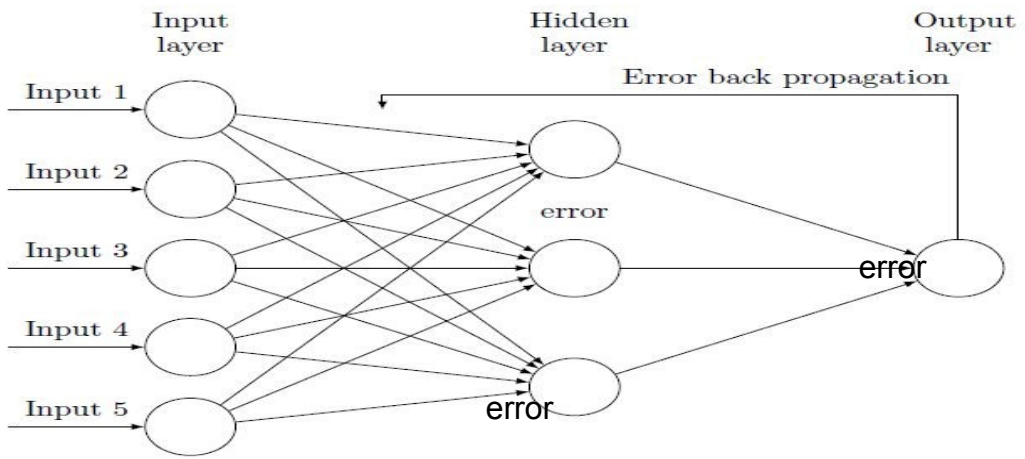


Output of every neuron in each consecutive layer is found

Multi-Layer Perceptron and Backpropagation

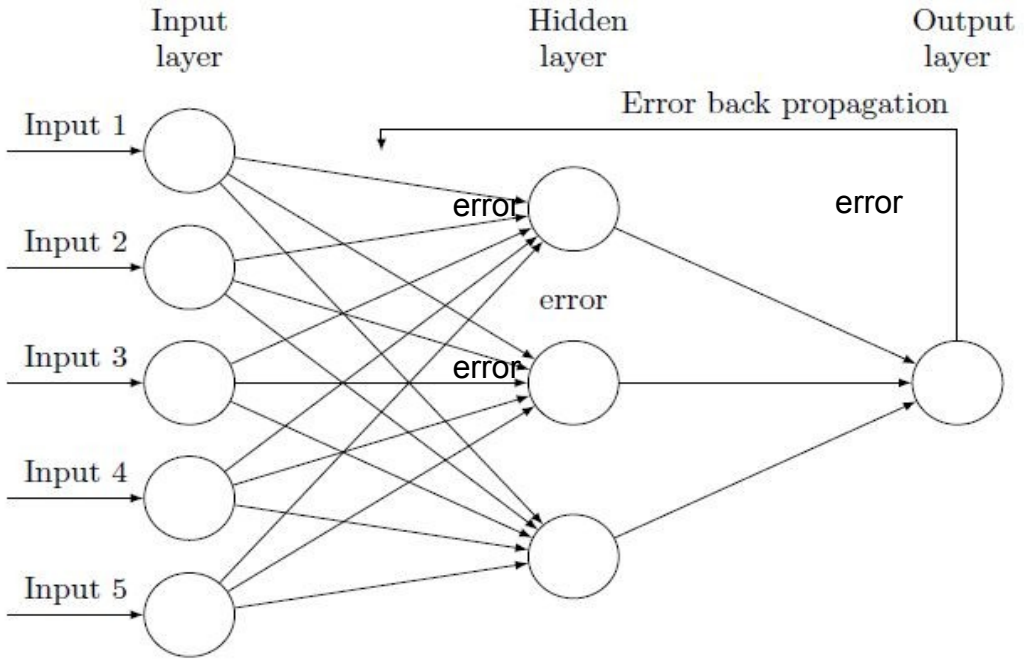
- Then it measures the network's output error i.e., the difference between the desired output and the actual output of the network
- It then computes how much each neuron in the last hidden layer contributed to each output neuron's error.

- It then proceeds to measure how much of these error contributions came from each neuron in the previous hidden layer and so on until the algorithm reaches the input layer.
- The last step of the backpropagation algorithm is a Gradient Descent step on all the connection weights in the network, using the error gradients measured earlier.



Multi-Layer Perceptron and Backpropagation

- This reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward in the network. Hence the name of the algorithm is Backpropagation



MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

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

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

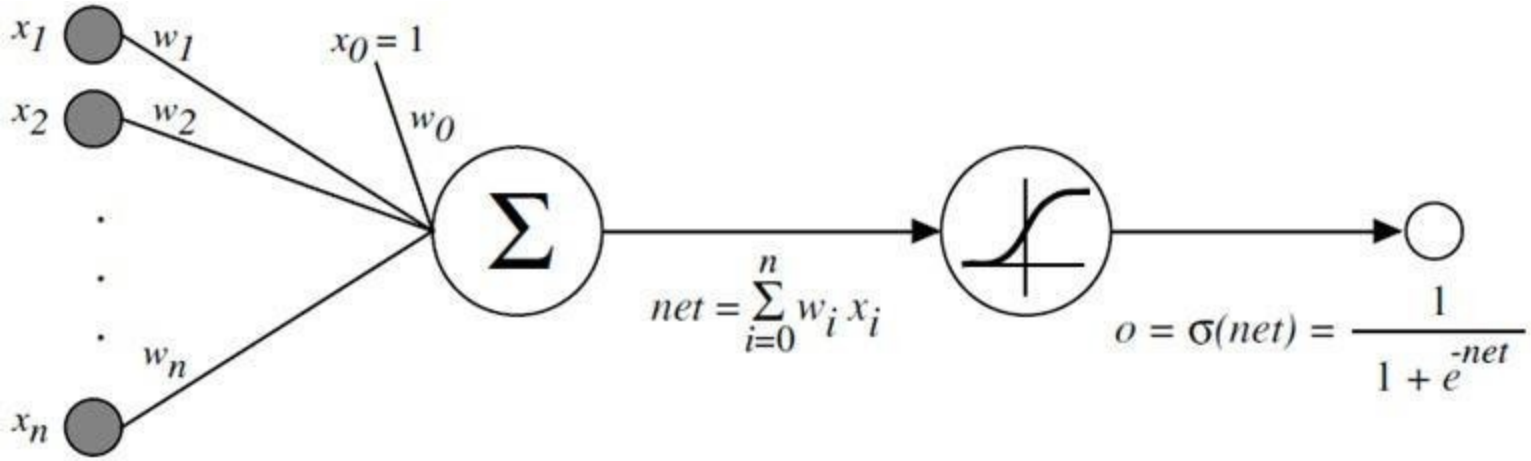


Figure: A Sigmoid Threshold Unit

$\sigma(y)$ 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

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

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 $k = j$.

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j) \qquad \text{.....equ(4)}\end{aligned}$$

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

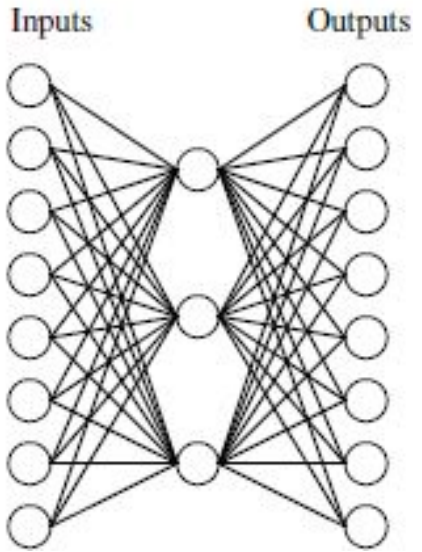
- Hypothesis space is the n-dimensional Euclidean space of the ***n*** network weights and hypothesis space is continuous.
- As it is continuous, E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure 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

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.