Module – 3

Artificial Neural Networks

Prof. Mahesh G Huddar

Dept. of Computer Science and Engineering

Artificial Neural Networks

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

Artificial Neural Networks

- ANN learning well-suited to problems which the training data corresponds to noisy, complex data (inputs from cameras or microphones)
- Can also be used for problems with symbolic representations
- Most appropriate for problems where
  - Instances have many attribute-value pairs
  - Target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes
  - Training examples may contain errors
  - Long training times are acceptable
  - Fast evaluation of the learned target function may be required
  - The ability for humans to understand the learned target function is not important

Appropriate Problems – for ANN

- ***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. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- ***The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.*** For example, in the ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction. We can also train a single network to output both the steering command and suggested acceleration, simply by concatenating the vectors that encode these two output predictions.

Appropriate Problems – for ANN

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

Long training times are acceptable. Network training algorithms typically require longer training times than, say, decision tree learning algorithms. 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.

Appropriate Problems – for ANN

Fast evaluation of the learned target function may be required. Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For example, 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. Learned neural networks are less easily communicated to humans than learned rules

Neural Network History

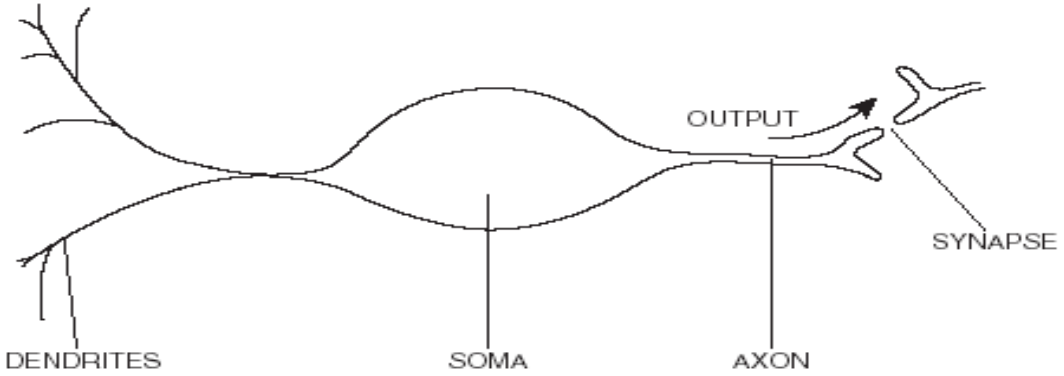
- History traces back to the 50's but became popular in the 80's with work by Rumelhart, Hinton, and Mclelland
  - A General Framework for Parallel Distributed Processing in Parallel Distributed Processing: Explorations in the Microstructure of Cognition
- Peaked in the 90's.:
  - Hundreds of variants
  - Less a model of the actual brain than a useful tool, but still some debate
- Numerous applications
  - Handwriting, face, speech recognition
  - Vehicles that drive themselves
  - Models of reading, sentence production, dreaming
- Debate for philosophers and cognitive scientists
  - Can human consciousness or cognitive abilities be explained by a connectionist model or does it require the manipulation of symbols?

Biological Motivation

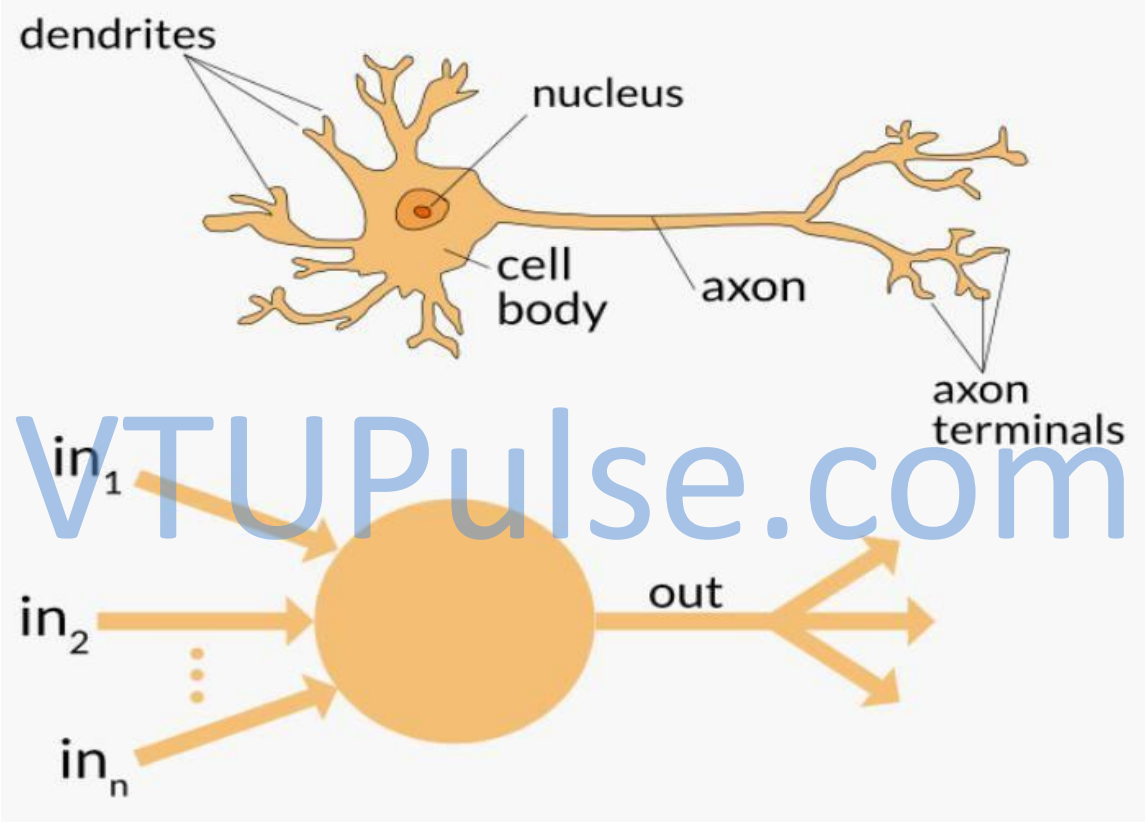
- The study of artificial neural networks (ANNs) has been inspired by the observation that biological learning systems are built of very complex webs of interconnected ***Neurons***
- Human information processing system consists of brain neuron: basic building block cell that communicates information 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

Biological Motivation

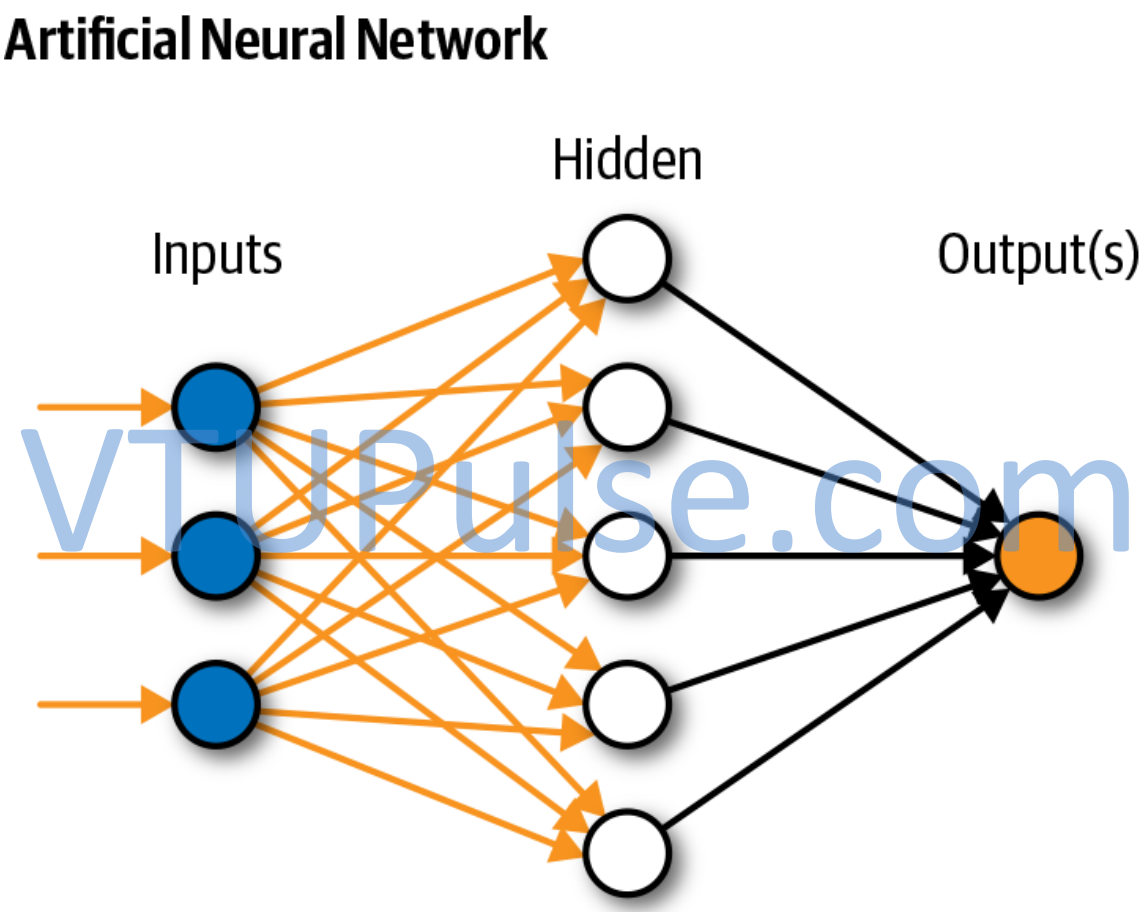
- The human brain is made up of billions of simple processing units – neurons.
- Inputs are received on dendrites, and if the input levels are over a threshold, the neuron fires, passing a signal through the axon to the synapse which then connects to another neuron.



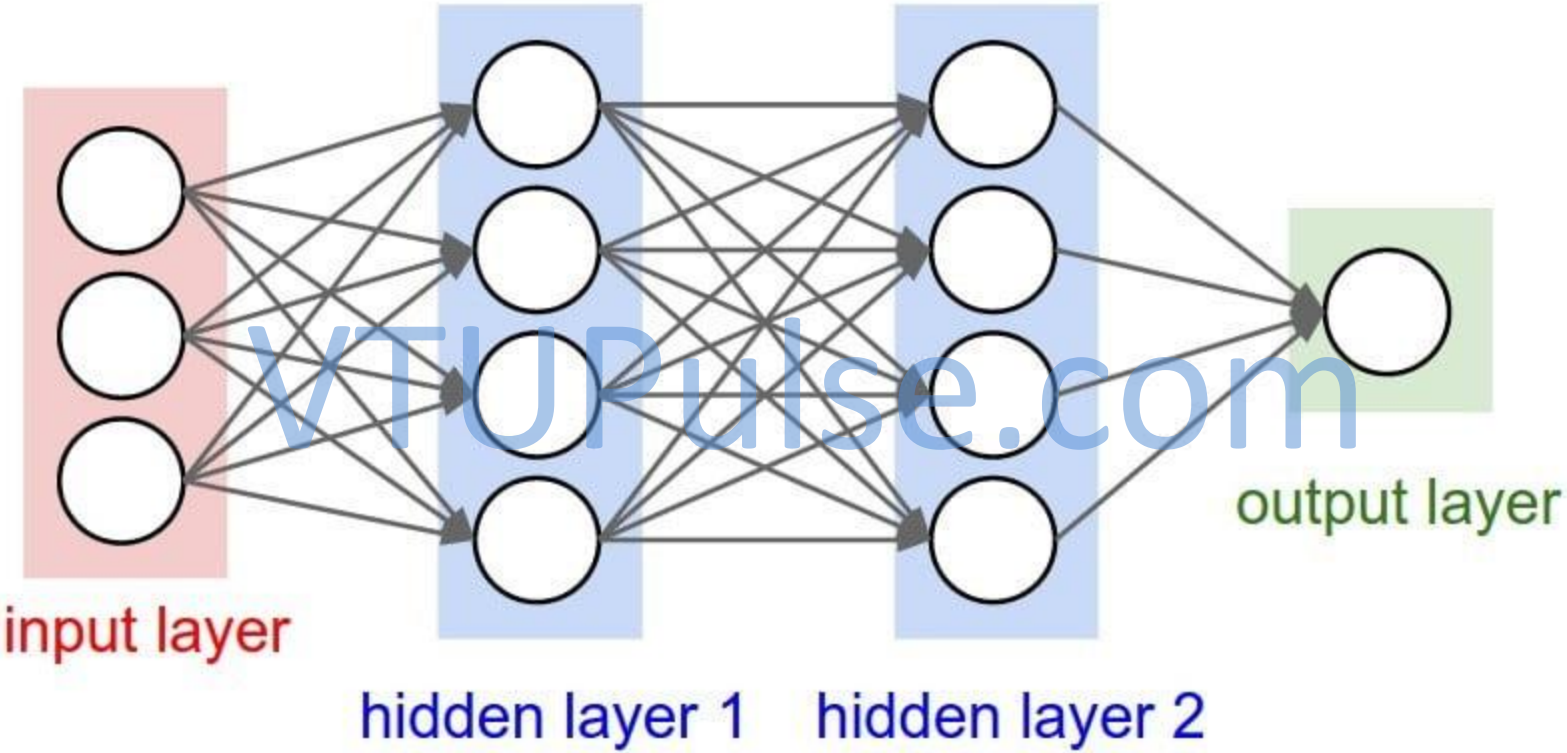
Biological Motivation



Simplest Neural Network

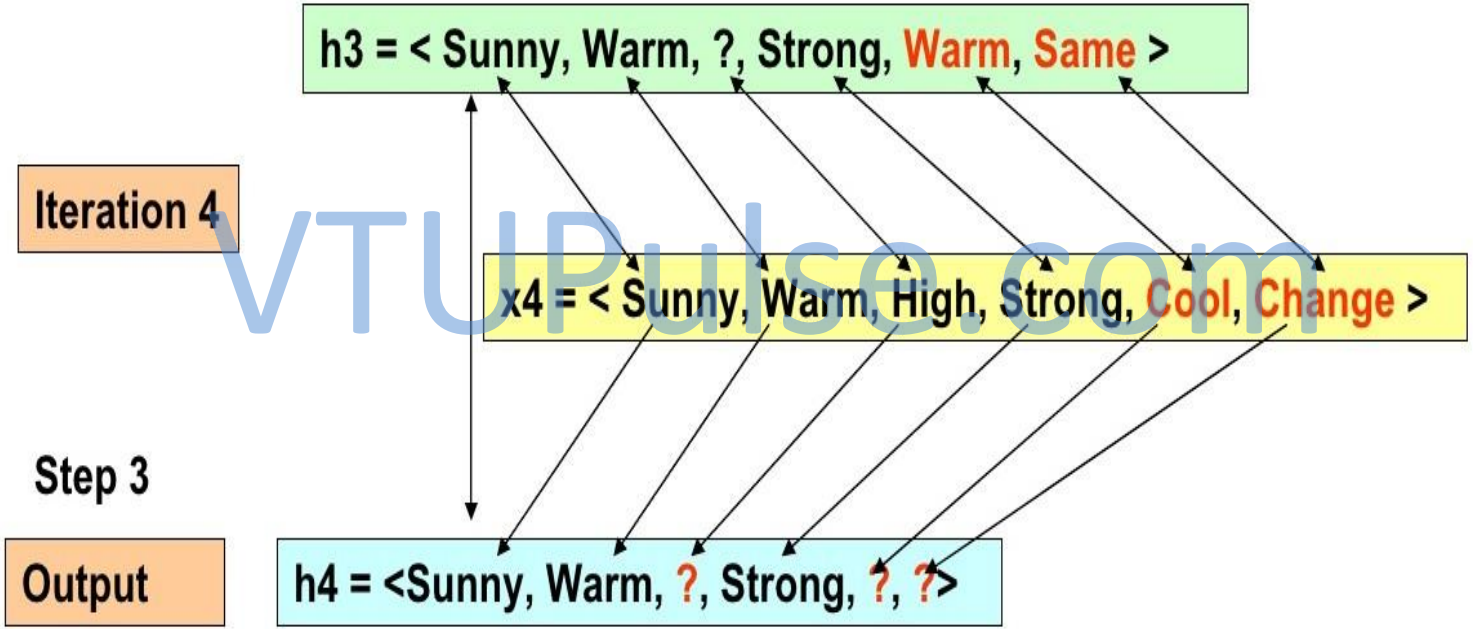


Simplest Neural Network



FIND-S: Step-2

Example	*Sky*	*AirTemp*	*Humidity*	*Wind*	*Water*	*Forecast*	*EnjoySport*
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes



Example	*Sky*	*AirTemp*	*Humidity*	*Wind*	*Water*	*Forecast*	*EnjoySport*
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

S_0: ⟨∅, ∅, ∅, ∅, ∅. ∅⟩

S_1: ⟨*Sunny,Warm, Normal, Strong, Warm, Same*⟩

S_2: S_3: ⟨*Sunny,Warm, ?, Strong, Warm, Same*⟩

S_4 ⟨*Sunny, Warm, ?, Strong, ?, ?*⟩

G_4: ⟨*Sunny, ?, ?, ?, ?, ?*⟩ ⟨*?, Warm, ?, ?, ?, ?*⟩

G_3: ⟨*Sunny,?,?,?,?,?*⟩ ⟨*?,Warm,?,?,?,?*⟩ ⟨*?,?,Normal,?,?,?*⟩ ⟨*?, ?,?,?,Cool,?*⟩ ⟨*?,?,?,?,?,Same*⟩

G_0: G_1: G_2: ⟨?, ?, ?, ?, ?, ?⟩

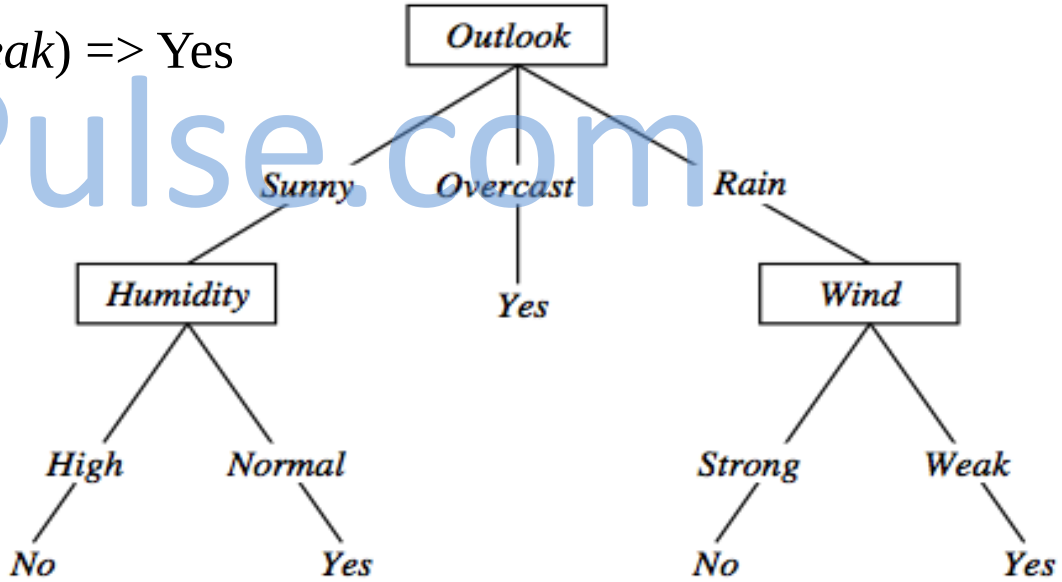
Decision Trees

- Decision trees represent a disjunction of conjunctions on constraints on the value of attributes:

  $(Outlook = Sunny \wedge Humidity = Normal) \Rightarrow \text{Yes}$

  $(Outlook = Overcast) \Rightarrow \text{Yes}$

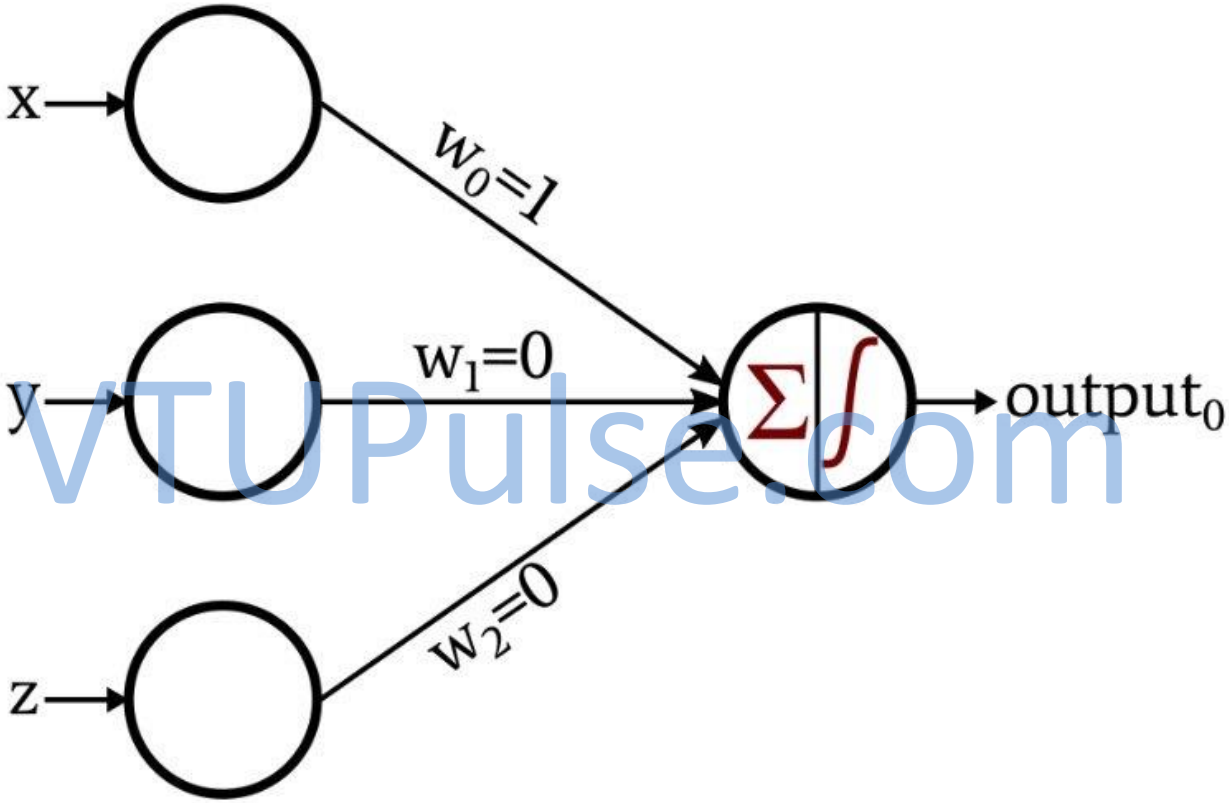
  $(Outlook = Rain \wedge Wind = Weak) \Rightarrow \text{Yes}$



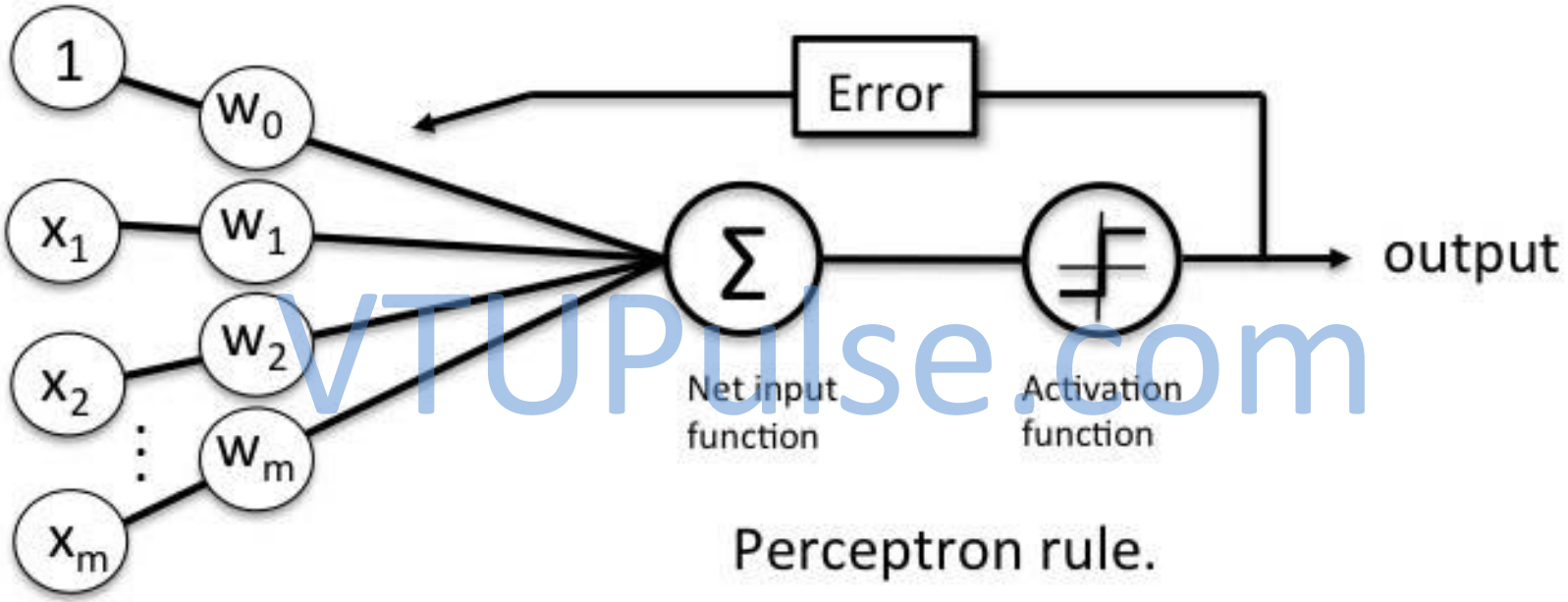
Artificial Neurons

- Artificial neurons are based on biological neurons.
- Each neuron in the network receives one or more inputs.
- An activation function is applied to the inputs, which determines the output of the neuron – the activation level.

Artificial Neurons

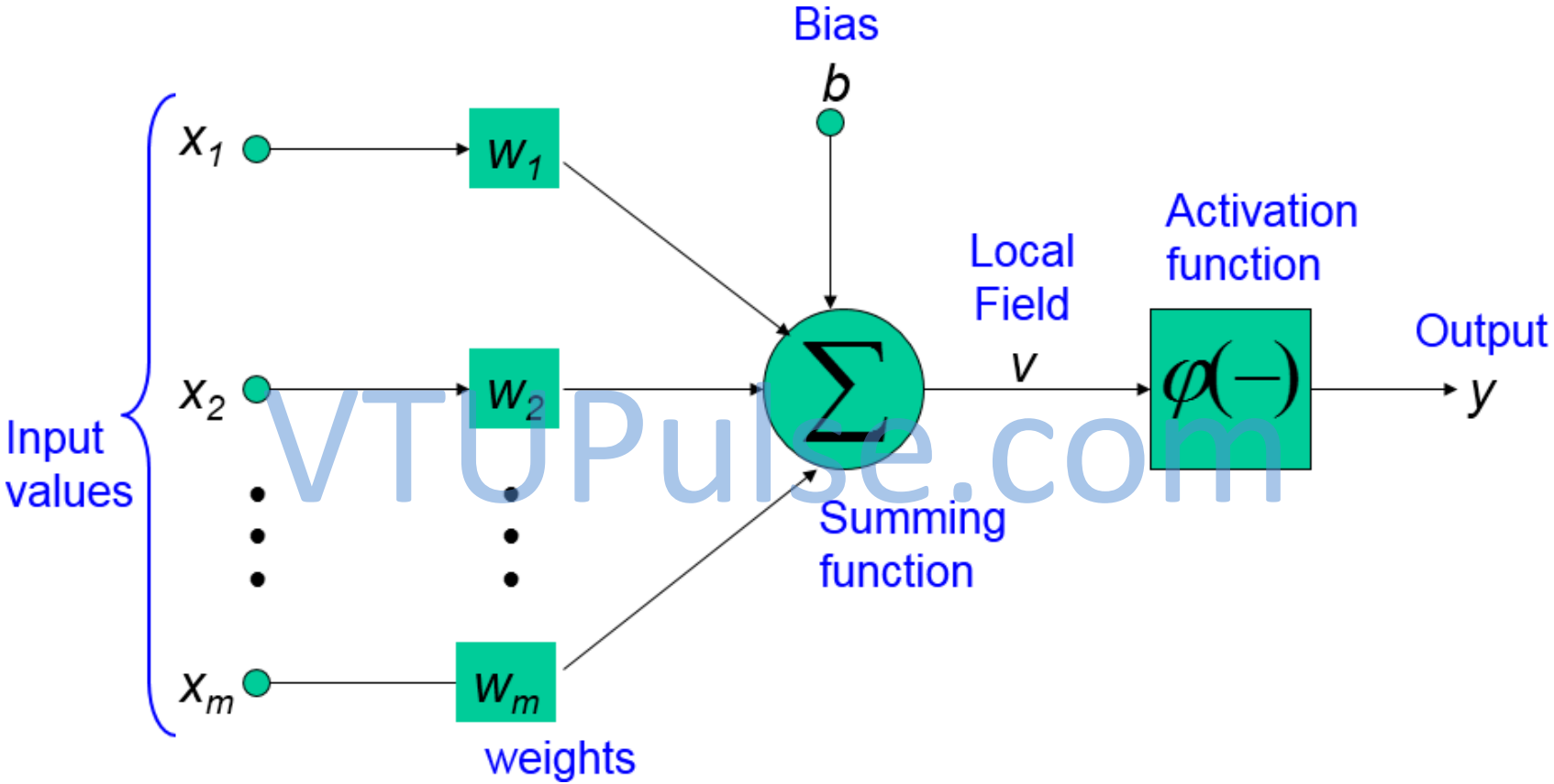


Artificial Neurons



Perceptron rule.

Artificial Neurons



Artificial Neurons

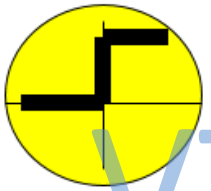
- A typical activation function works as follows:

$$X = \sum_{i=1}^{n} w_i x_i \qquad Y = \begin{cases} +1 & for\ X > t \\ 0 & for\ X \le t \end{cases}$$

- Each node *i* has a weight, w_i associated with it.
- The input to node *i* is x_i.
- *t* is the threshold.
- So if the weighted sum of the inputs to the neuron is above the threshold, then the neuron fires.

Artificial Neurons

- The charts on the right show three typical activation functions.

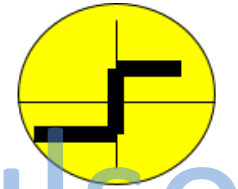


Step function

(Linear Threshold Unit)

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

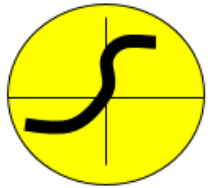
0, if x < threshold



Sign function

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

-1, if x < 0



Sigmoid function

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

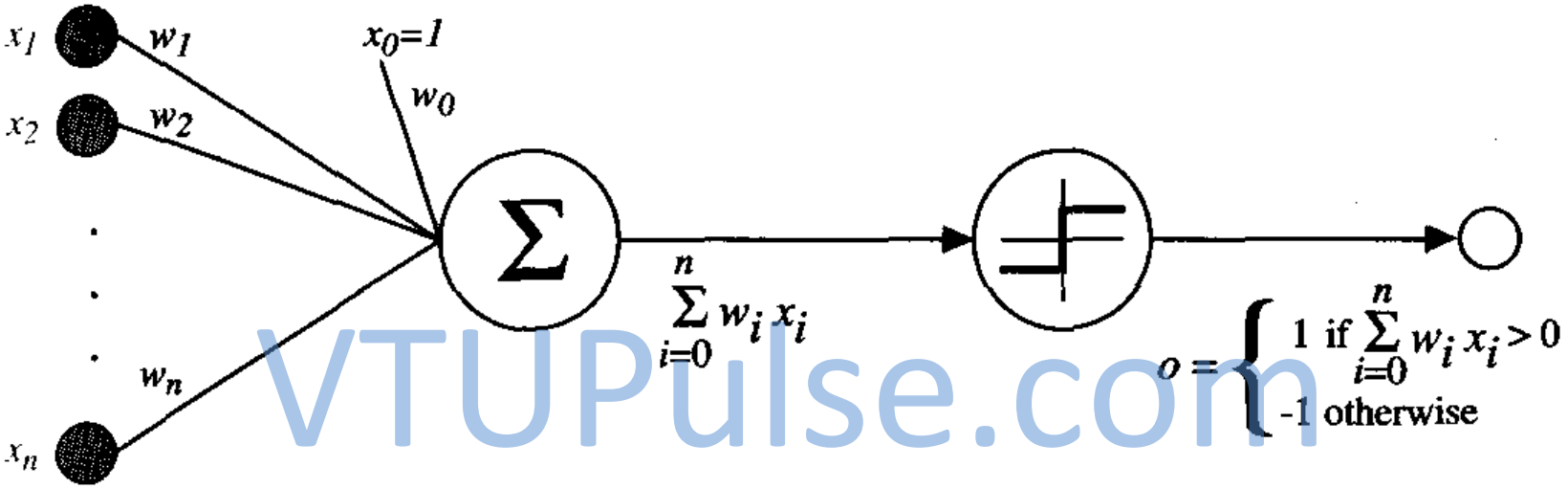
PERCEPTRON

- One type of ANN system is based on a unit called a perceptron.
- 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



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

PERCEPTRON

- For brevity, we will sometimes write the perceptron function as,

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

where

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

- Learning a perceptron involves choosing values for the weights ***wo,*** . . . , ***wn.***
- Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

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

The 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

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

The Perceptron Training Rule

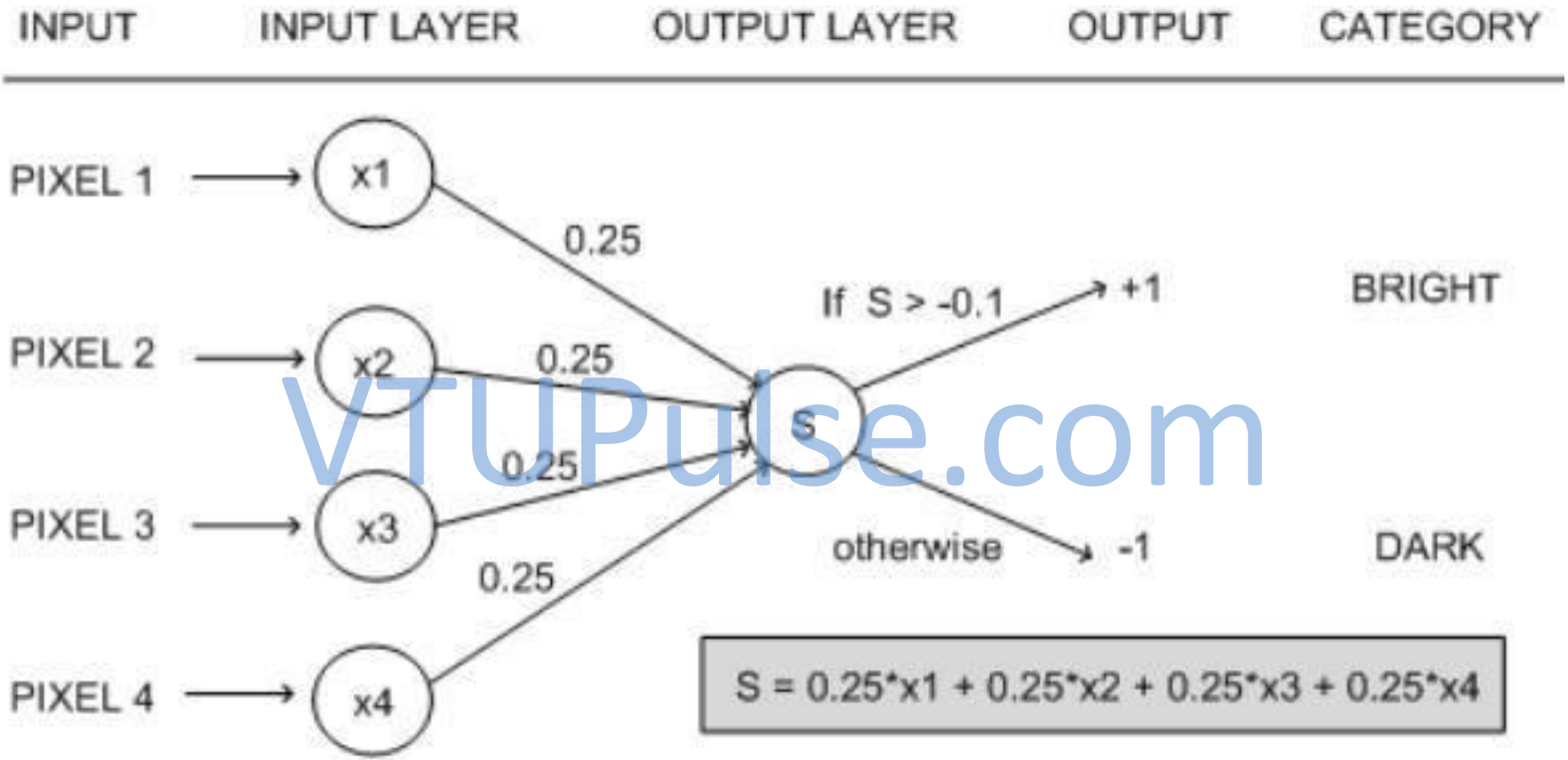
$$w_i \leftarrow w_i + \Delta w_i$$

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, and *n* 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.

The Perceptron Training Rule



The Perceptron Training Rule

- Why should this update rule converge toward successful weight values?
- To get an intuitive feel, consider some specific cases.
- Suppose the training example is correctly classified already by the perceptron. In this case, $(t - o)$ is zero, making Δw_i zero, so that no weights are updated.
- Suppose the perceptron outputs a -1, when the target output is + 1.
- To make the perceptron output a + 1 instead of - 1 in this case, the weights must be altered to increase the value of $\vec{w} \cdot \vec{x}$.
- For example, if $x_i > 0$, then increasing w_i will bring the perceptron closer to correctly classifying this example. Notice the training rule will increase w_i in this case, because $(t - o)$, n, and x_i are all positive.
- For example, if $x_i = .8$, $q = 0.1$, $t = 1$, and $o = -1$, then the weight update will be $\Delta w_i = q(t - o)x_i = 0.1(1 - (-1))0.8 = 0.16$.
- On the other hand, if $t = -1$ and $o = 1$, then weights associated with positive x_i will be decreased rather than increased.

The Perceptron Training Rule

A single perceptron can be used to represent many Boolean functions weights 0.6 and 0.6

AND function

$Threshhold\ \theta = 1$

Learning Rate $n = 0.5$

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

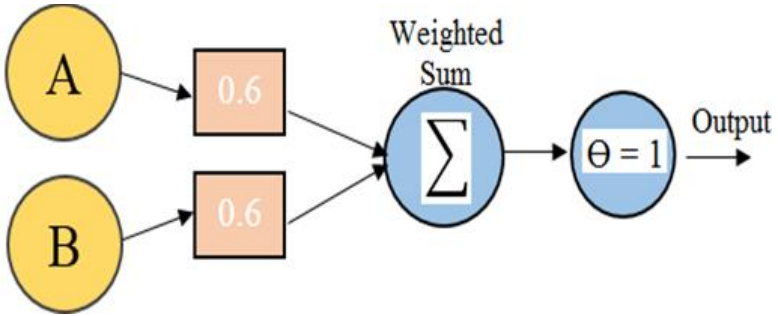
This is not greater than the threshold, so the output = 0

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

A single perceptron can be used to represent many Boolean functions weights 1.2 and 0.6

AND function

$Threshhold\ \theta = 1$

Learning Rate $n = 0.5$

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

If A=0 & B=0 → 0*1.2 + 0*0.6 = 0

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

If A=0 & B=1 → 0*1.2 + 1*0.6 = 0.6

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

If A=1 & B=0 → 1*1.2 + 0*0.6 = 1.2

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

But the expected output is 0

The Perceptron Training Rule

A single perceptron can be used to represent many Boolean functions weights 1.2 and 0.6

AND function

$Threshhold\ \theta = 1$

Learning Rate $n = 0.5$

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

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

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

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

The Perceptron Training Rule

A single perceptron can be used to represent many Boolean functions weights 1.2 and 0.6

AND function

$Threshhold\ \theta = 1$

Learning Rate $n = 0.5$

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

If A=0 & B=0 → 0*0.7 + 0*0.6 = 0

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

If A=0 & B=1 → 0*0.7 + 1*0.6 = 0.6

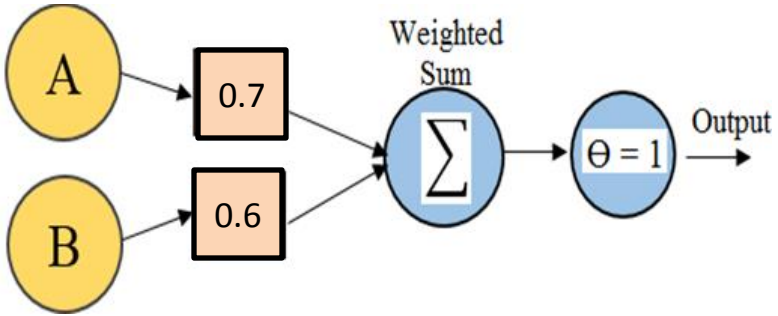
This is not greater than the threshold, so the output = 0

If A=1 & B=0 → 1*0.7 + 0*0.6 = 0.7

This is greater than the threshold, so the output = 0

If A=1 & B=0 → 1*0.7 + 1*0.6 = 1.3

This is greater than the threshold, so the output = 0



The Perceptron Training Rule

A single perceptron can be used to represent many Boolean functions with initial weights 0.6 and 0.6

$Threshhold\ \theta = 1$

Learning Rate $n = 0.5$

OR function

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

But the expected output is 1

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

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

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

The Perceptron Training Rule

A single perceptron can be used to represent many Boolean functions

OR function

$Threshhold\ \theta = 1$

Learning Rate $n = 0.5$

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

If A=0 & B=0 → 0*0.6 + 0*1.1 = 0

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

If A=0 & B=1 → 0*0.6 + 1*1.1 = 1.1

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

If A=1 & B= → 1*1.1 + 0*1.1 = 1.1

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

But the expected output is 1

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

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

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

The Perceptron Training Rule

A single perceptron can be used to represent many Boolean functions

OR function

$Threshhold\ \theta = 1$

Learning Rate $n = 0.5$

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

If A=0 & B=0 → 0*1.1 + 0*1.1 = 0

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

If A=0 & B=1 → 0*1.1 + 1*1.1 = 1.1

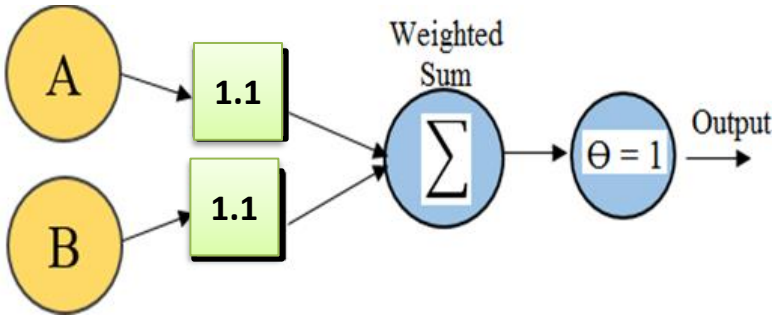
This is greater than the threshold, so the output = 1.

If A=1 & B= → 1*1.1 + 0*1.1 = 1.1

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

If A=1 & B= → 1*1.1 + 1*1.1 = 2.2

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



The Perceptron Training Rule

A single perceptron can be used to represent many Boolean functions

NOT function

The Perceptron Training Rule

Perceptron_training_rule (X, η)
initialize **w** (wi ← an initial (small) random value)
repeat
  for each training instance (**x**, *t***x**) ∈ X
    compute the real output *o***x**=*Summation(***w**.x*)*
    if (*t***x** ≠ *o***x**)
      for each wi
        wi ← wi + Δ*wi*
        Δ*wi* ← η(t**x**-*o***x**)xi
      end for
    end if
  end for
until all the training instances in X are correctly classified
return **w**

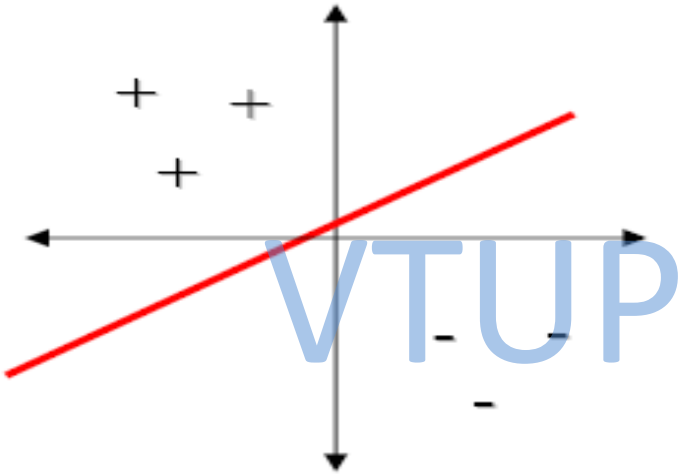
X: training data
η: learning rate (small positive constant, e.g., 0.1)

Examples
• **x** is correctly classified, *o***x**–*o***x**=0
→ no update
• *o***x**=-1 but *t***x**=1, t**x**-*o***x**>0
→ wi is increased if xi>0, decreased otherwise
→**w**.**x** is increased
• *o***x**=1, but *out***x**=-1, *out***x**-*o***x**<0
→wi is decreased if xi>0, increased otherwise
→**w**.□**x** is decreased

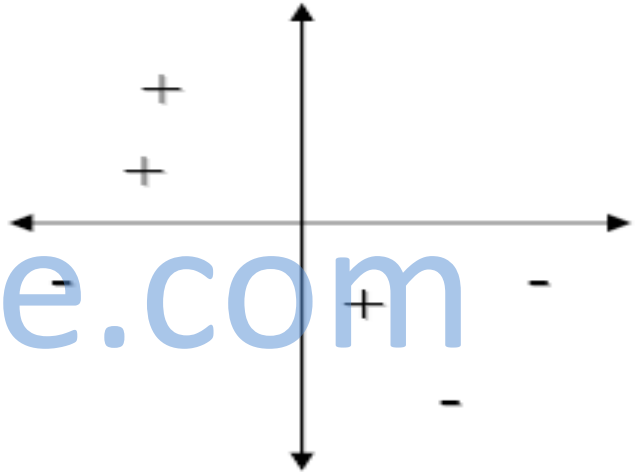
Representational Power of Perceptron's

- We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points).
- 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.
- The equation for this decision hyperplane $\vec{w}.\vec{x} = 0$.
- 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.

Representational Power of Perceptron's



Linearly separable



Non-linearly separable

Representational Power of Perceptron's

- 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 ***w1 = w2 = 0.5***.
- This perceptron can be made to represent the OR function instead by altering the threshold to ***wo = -0.3.***
- 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.
- Any m-of-n function is easily represented using a perceptron by setting all input weights to the same value (e.g., 0.5) and then setting the threshold *wo* accordingly.

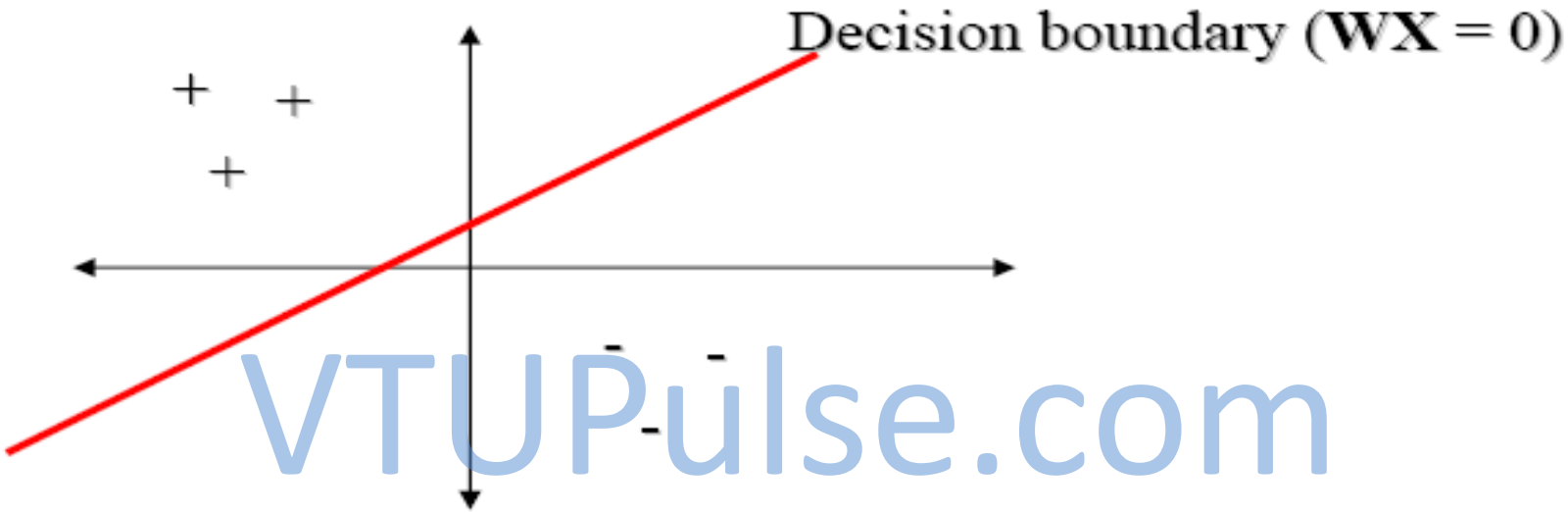
Representational Power of Perceptron's

- 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.**
- Note the set of linearly nonseparable training examples shown in Figure (b) corresponds to this XOR function.

Representational Power of Perceptron's

- The ability of perceptrons to represent AND, OR, NAND, and NOR is important because ***every*** boolean function can be represented by some network of interconnected units based on these primitives.
- In fact, every boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage.
- One way is to represent the boolean function in disjunctive normal form (i.e., as the disjunction (OR) of a set of conjunctions (ANDs) of the inputs and their negations).
- Note that the input to an AND perceptron can be negated simply by changing the sign of the corresponding input weight.
- Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units.

Representational Power of Perceptron's

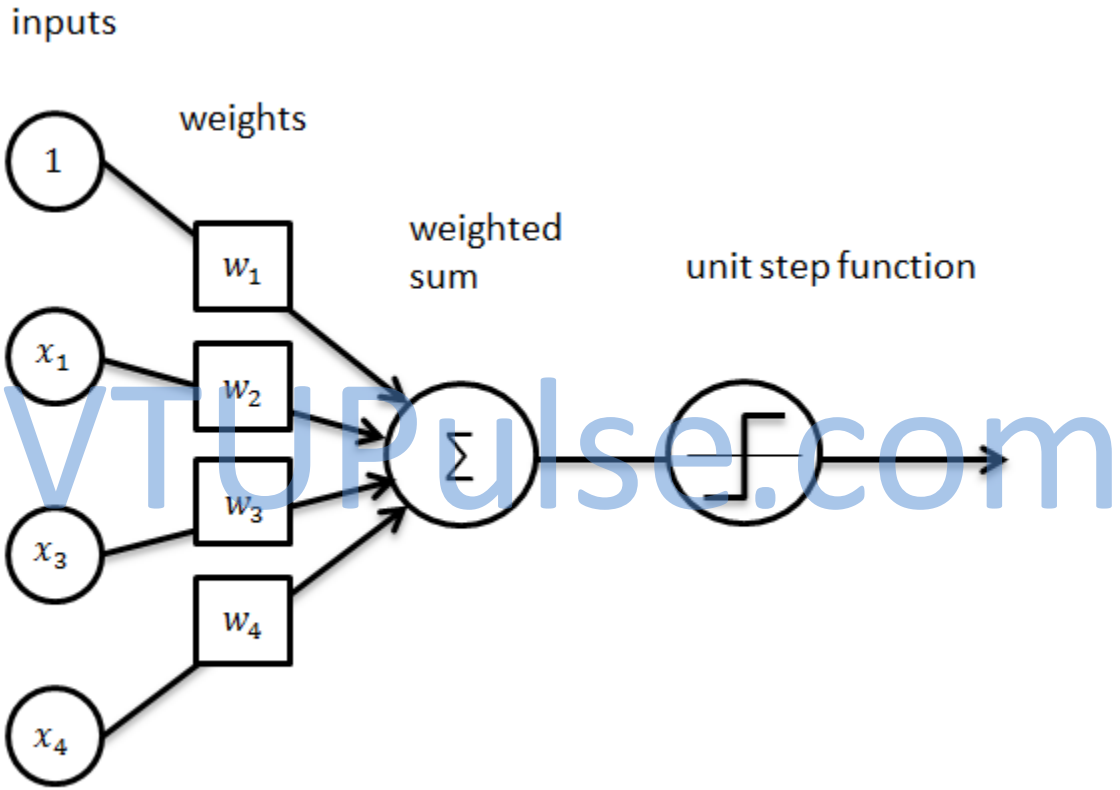


A perceptron can learn only examples that are called "linearly separable". These are examples that can be perfectly separated by a hyperplane.

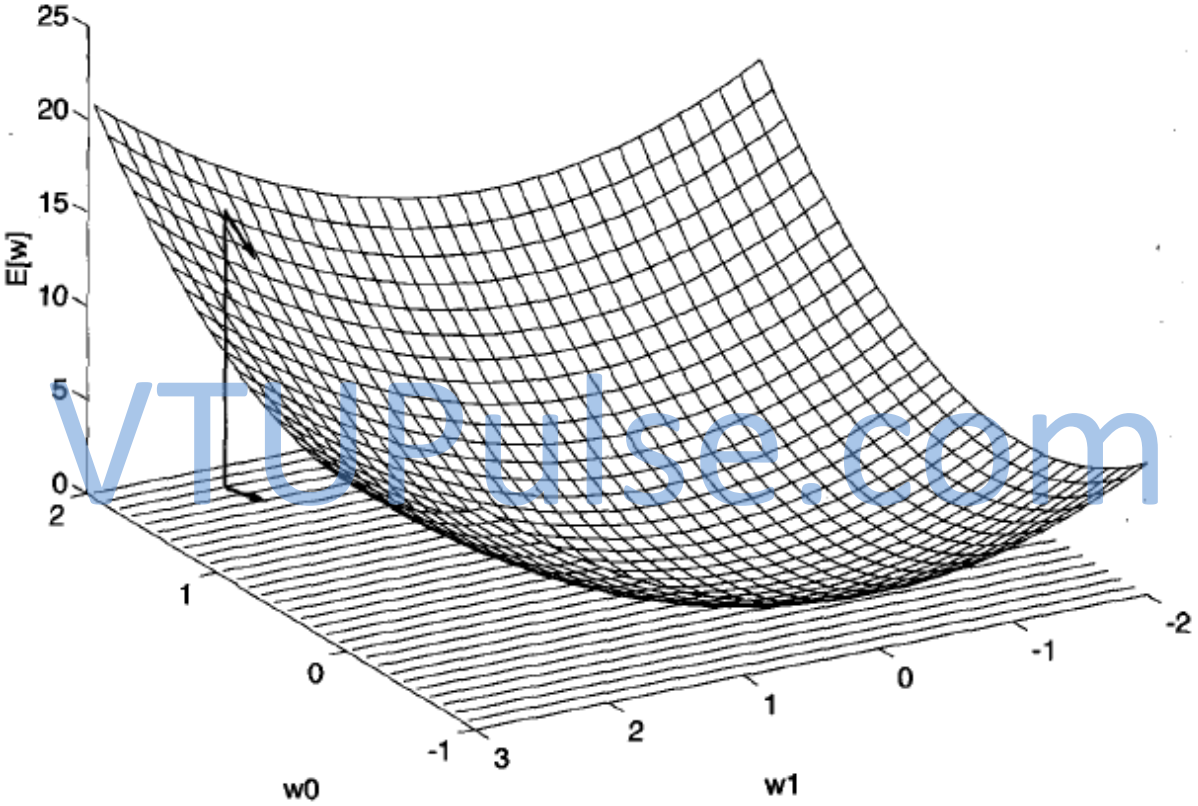
Gradient Descent and the Delta Rule

- Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable.
- **A** second training rule, called the ***delta rule,*** is designed to overcome this difficulty.
- 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.
- This rule is important because gradient descent provides the basis for the BACKPROPAGATON algorithm, which can learn networks with many interconnected units.
- It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

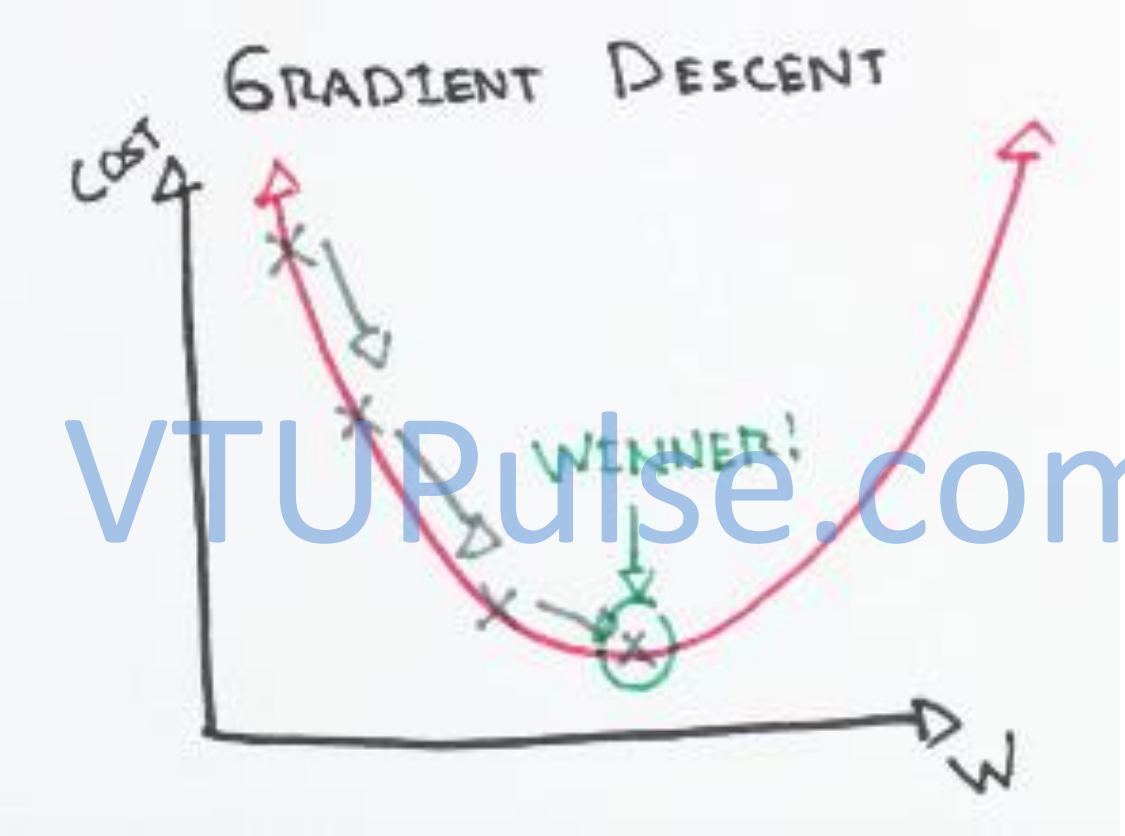
Gradient Descent and the Delta Rule



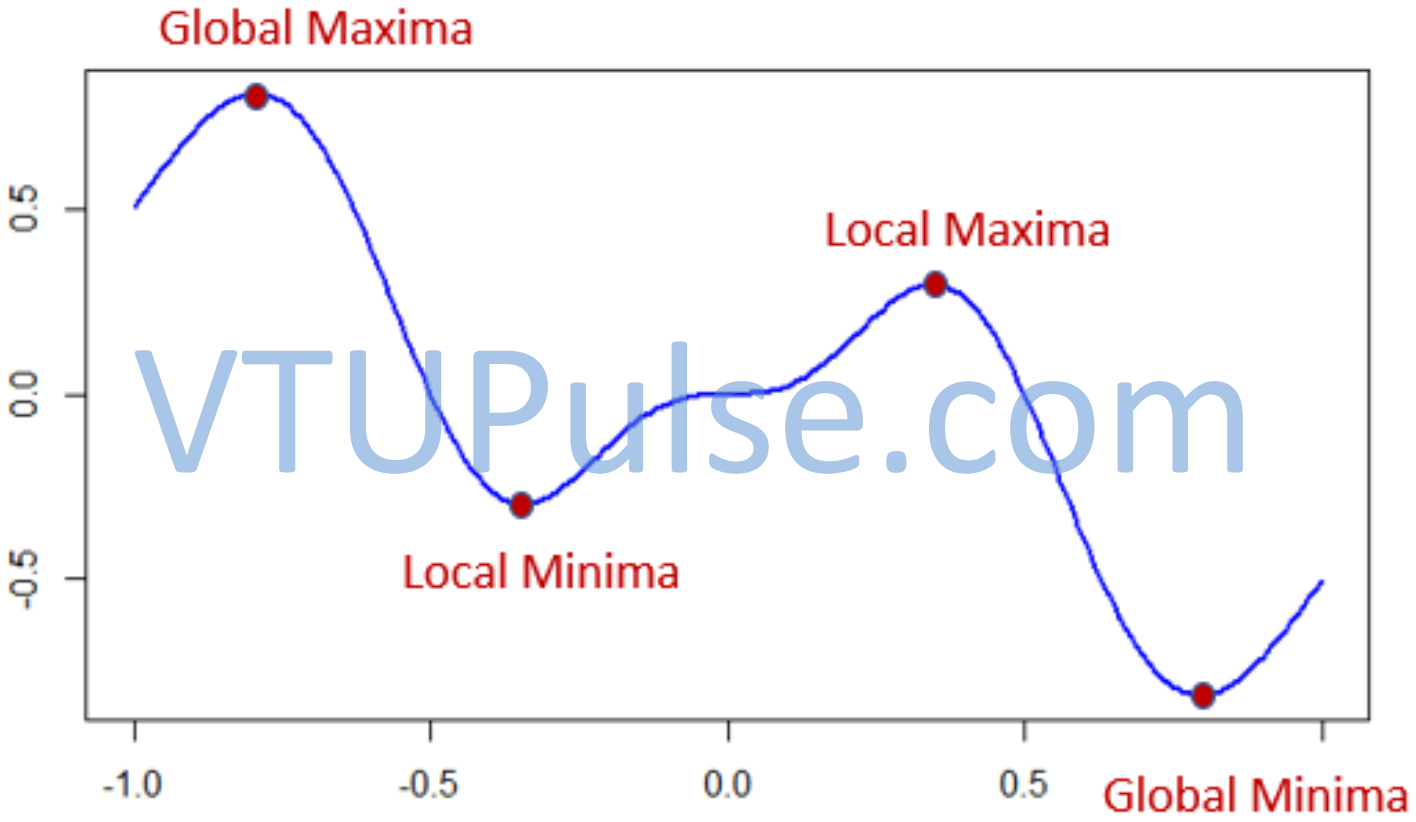
Gradient Descent and the Delta Rule



Gradient Descent and the Delta Rule



Gradient Descent and the Delta Rule



Gradient Descent and the Delta Rule

- The delta training rule is best understood by considering the task of training an ***unthresholded*** perceptron; that is, a ***linear unit*** for which the output ***o*** is given by

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

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

- Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.
- In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples.

- Although there are many ways to define this error, one common measure is

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

- where D is the set of training examples, ***td*** is the target output for training example *d*, and ***od*** is the output of the linear unit for training example d.

Derivation of Gradient Descent Rule

- How can we calculate the direction of steepest descent along the error surface?
- This direction can be found by computing the derivative of ***E*** with respect to each component of the vector $\vec{w}$.
- This vector derivative is called the ***gradient*** of ***E*** with respect to $\vec{w}$, written ***as*** $\nabla E(\vec{w})$.

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

- Since 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

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

$$\Delta\vec{w} = -\eta\nabla E(\vec{w})$$

Derivation of Gradient Descent Rule

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

where

$$\Delta\vec{w} = -\eta\nabla E(\vec{w})$$

- Here *n* 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

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

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

Derivation of Gradient Descent Rule

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

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

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

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

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

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

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

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

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

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

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

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

  - For each linear unit weight w_i, Do

$$w_i := w_i + \Delta w_i$$

STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

- 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 (e.g., the weights in a linear unit), and
  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 (i.e., it can require many thousands of gradient descent steps), and
  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

Stochastic 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

$$w_i := w_i + \eta(t - o)x_i$$

STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

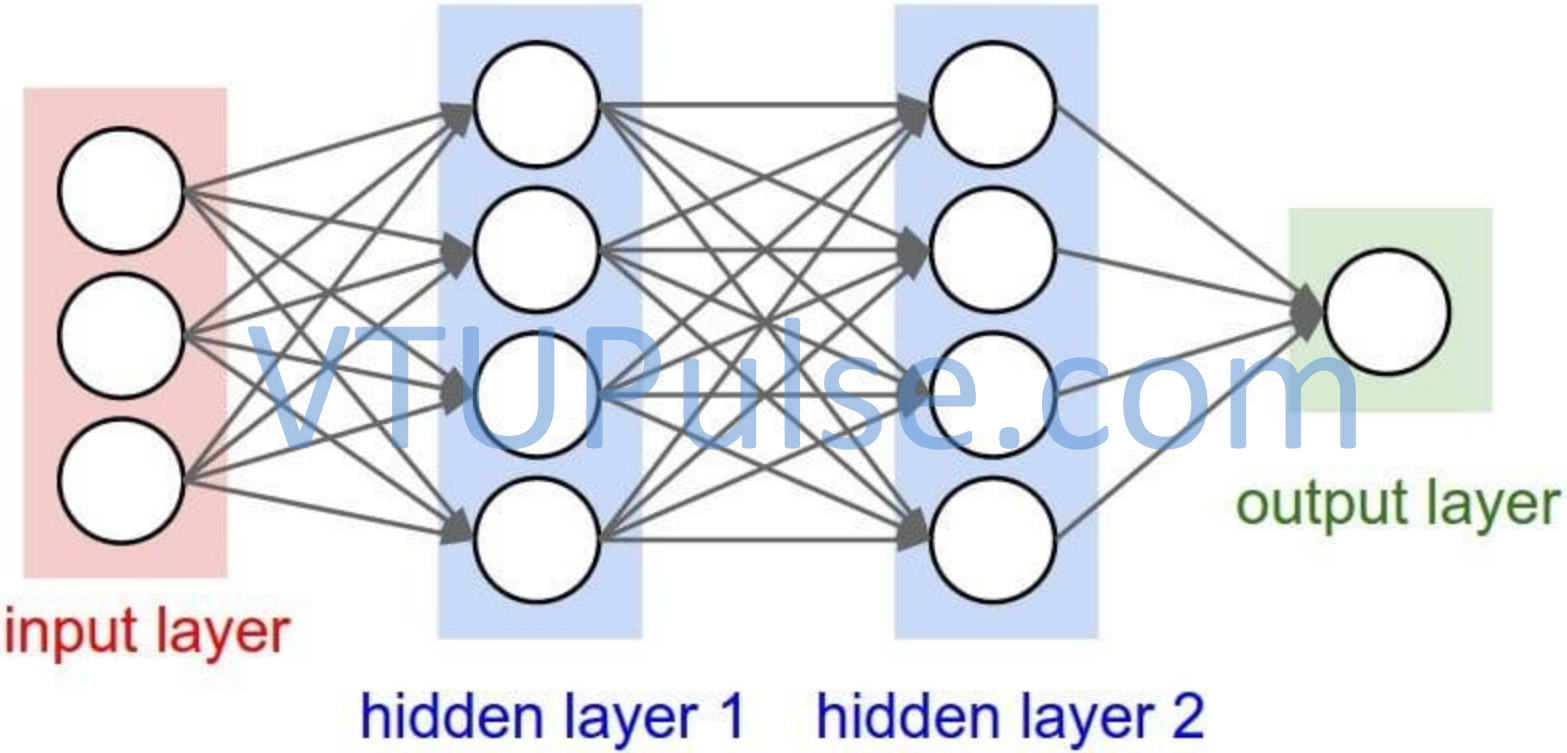
The key differences between standard gradient descent and stochastic gradient descent are:

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- 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 $E(\vec{w})$, 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.

MULTILAYER NETWORKS

- Multilayer neural networks can classify a range of functions, including non linearly separable ones.
- Each input layer neuron connects to all neurons in the hidden layer.
- The neurons in the hidden layer connect to all neurons in the output layer.

MULTILAYER NETWORKS



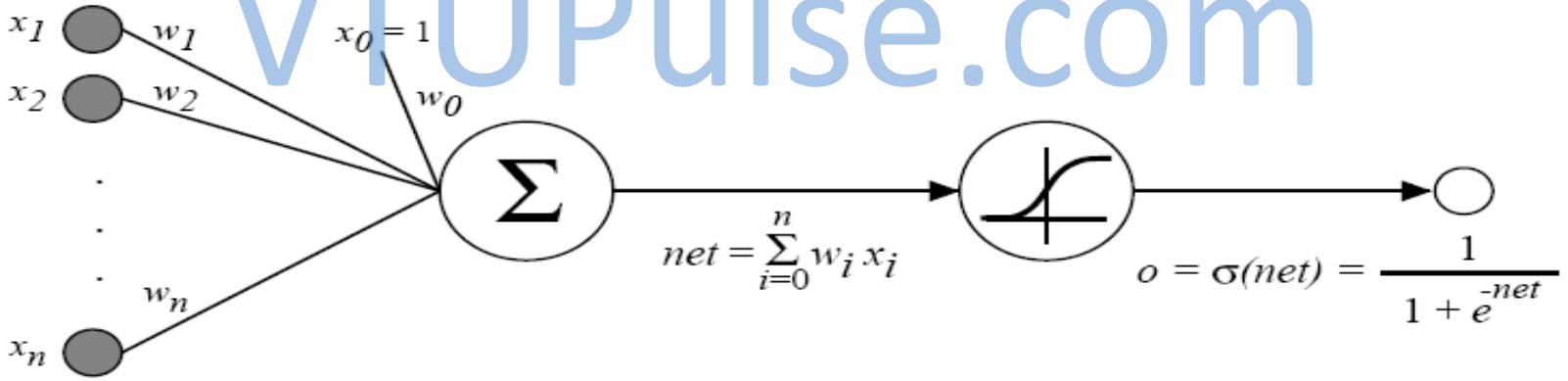
MULTILAYER NETWORKS

A Differentiable Threshold Unit

- What type of unit shall we use as the basis for constructing multilayer networks?
- At first we might be tempted to choose the linear units discussed in the previous section, for which we have already derived a gradient descent learning rule.
- However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions.
- The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent.
- What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs.
- One solution is the sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

MULTILAYER NETWORKS

- The sigmoid unit is illustrated in below Figure. Like the perceptron, 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.



MULTILAYER NETWORKS

More precisely, the sigmoid unit computes its output ***o*** as

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

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

$$\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$$

The BACKPROPAGATIAON Algorithm

- Multilayer neural networks learn in the same way as perceptrons.
- However, there are many more weights, and it is important to assign credit (or blame) correctly when changing weights.
- *E* sums the errors over all of the network output units

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

The BACKPROPAGATIAON Algorithm

x_{ji} = the ith input to unit j

w_{ji} = the weight associated with the ith input to unit j

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

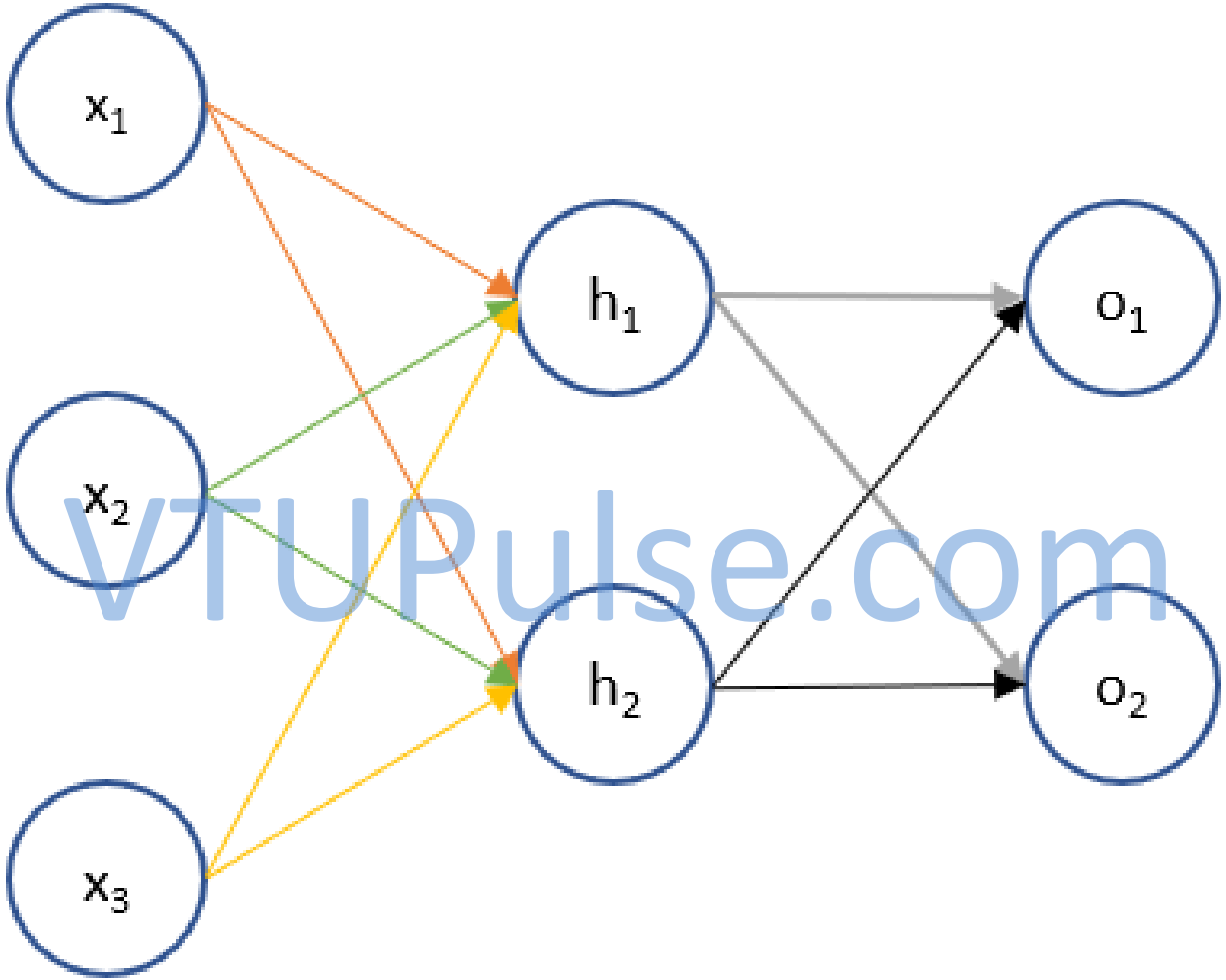
o_j = the output computed by unit j

t_j = the target output for unit j

σ = the sigmoid function

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

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



The BACKPROPAGATIAON Algorithm

- The learning problem faced by BACKPROPAGATION Algorithm search a large hypothesis space defined by **all** possible weight values for all the units in the network.

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $(\vec{x}, \vec{t})$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} 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_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
Until the termination condition is met, Do

- For each $\langle \vec{x}, \vec{t} \rangle$ 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

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

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

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

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

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

VTUPulse.com

Derivation of BACKPROPAGATIAON Rule

x_{ji} = the ith input to unit j

w_{ji} = the weight associated with the ith input to unit j

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

o_j = the output computed by unit j

t_j = the target output for unit j

σ = the sigmoid function

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

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

Derivation of BACKPROPAGATIAON Rule

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

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

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

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

Derivation of BACKPROPAGATIAON Rule

- The specific problem we address here is deriving the stochastic gradient descent rule, that stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error ***Ed*** with respect to this single example.
- In other words, for each training example d every weight ***wji*** is updated by adding to it Δw_{ij}

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

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

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

- Here outputs is the set of output units in the network, tk is the target value of unit k for training example d, and ok is the output of unit k given training example d.

Derivation of BACKPROPAGATIAON Rule

- To begin, notice that weight ***wji*** can influence the rest of the network only through ***netj.*** Therefore, we can use the chain rule to write,

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

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

- our remaining task is to derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$
- We 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.

Derivation of BACKPROPAGATIAON Rule

Case 1: Training Rule for Output Unit Weights

- Just **as *wji*** can influence the rest of the network only through ***net,, net,*** can influence the network only through ***oj.*** Therefore, we can invoke the chain rule again to write,

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

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

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

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

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

$$= -(t_j - o_j)$$

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

Derivation of BACKPROPAGATIAON Rule

Case 1: Training Rule for Output Unit Weights

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

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial net_j} x_{ji}$$

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

Derivation of BACKPROPAGATIAON Rule

Case 2: Training Rule for Hidden Unit Weights

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

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

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

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

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

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

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

$$\delta_k = (t_j - o_j)\, o_j(1 - o_j)$$

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

$$= o_j(1 - o_j)$$

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial net_j} x_{ji}$$

$$\Delta w_{ji} = -\eta \, o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj} \, x_{ji}$$

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

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

Neural Network Representations - ALVINN

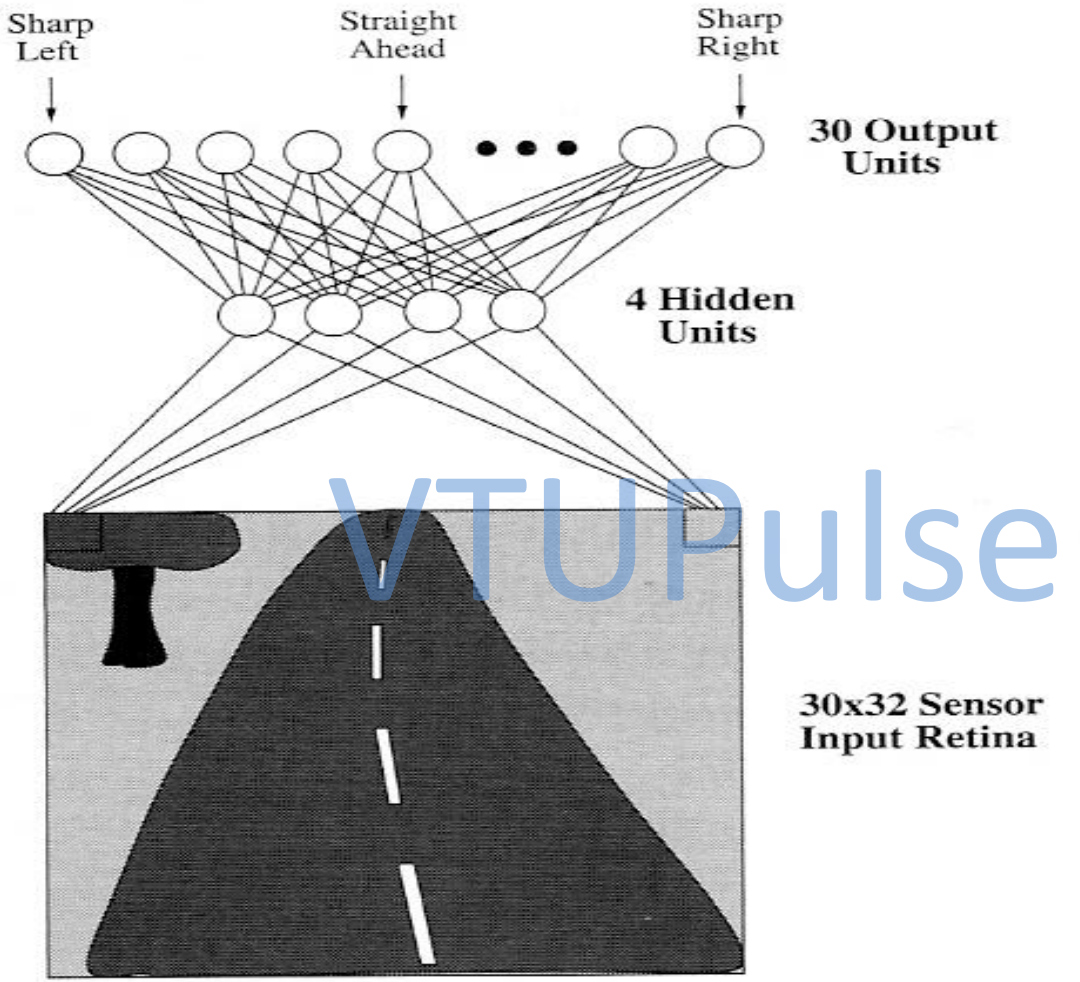
- **A** prototypical example of **ANN** learning which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways.
- The input to the neural network is a 30 x 32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle.
- The network output is the direction in which the vehicle is steered. The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes.
- ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways

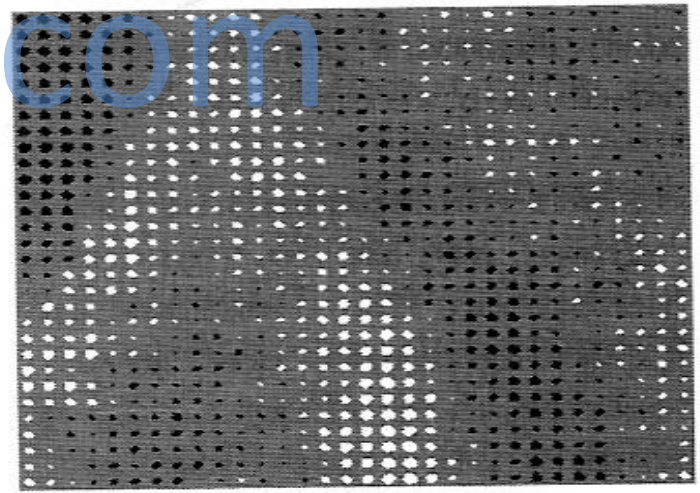
Neural Network Representations - ALVINN

- Figure illustrates the neural network representation used in one version of the ALVINN system, and illustrates the kind of representation typical of many ANN systems.
- 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.
- As can be seen, 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.

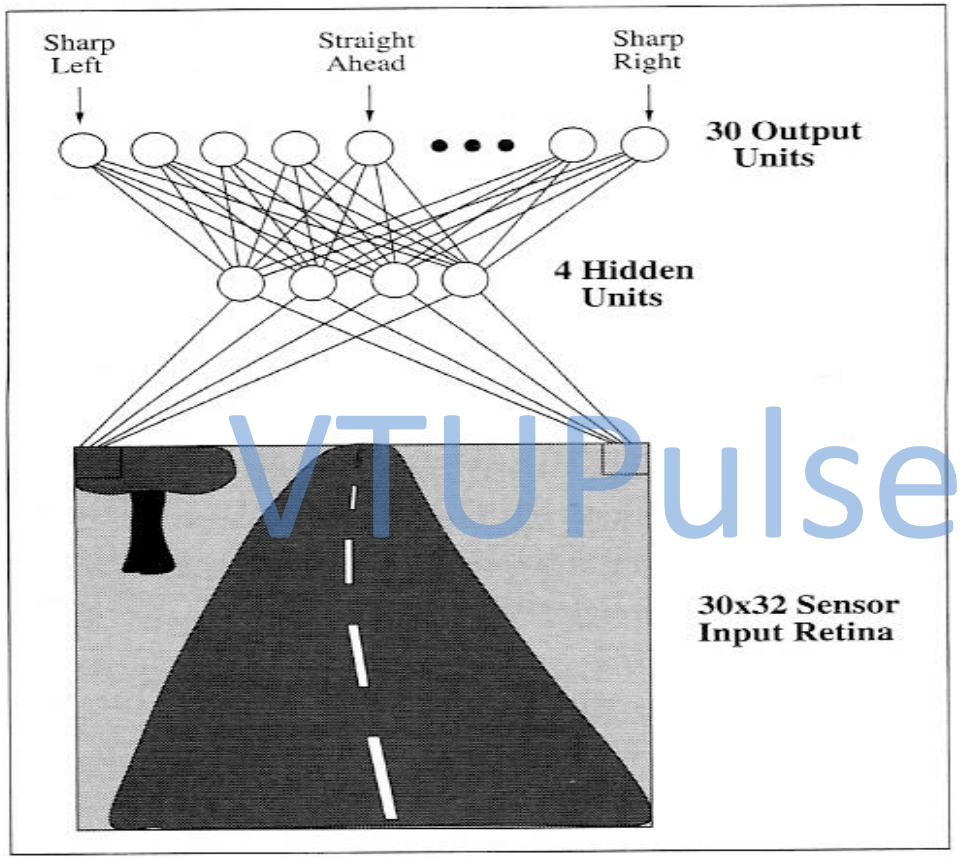
Neural Network Representations - ALVINN

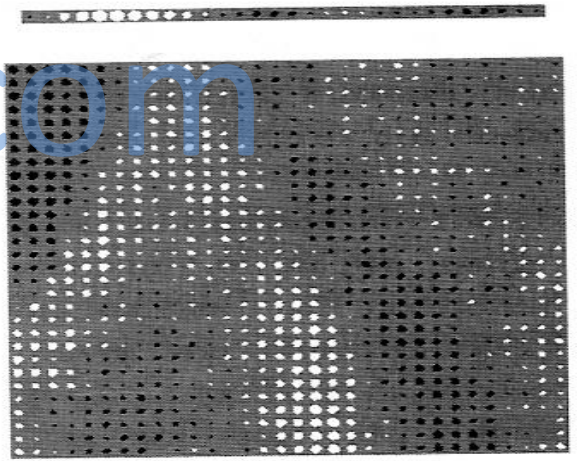
- The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN.
- The large matrix of black and white boxes on the lower right depicts the weights from the 30 x 32 pixel inputs into the hidden unit.
- Here, a white box indicates 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.





ALVINN





VTUPulse.com