Module3 Notes (Pointwise)

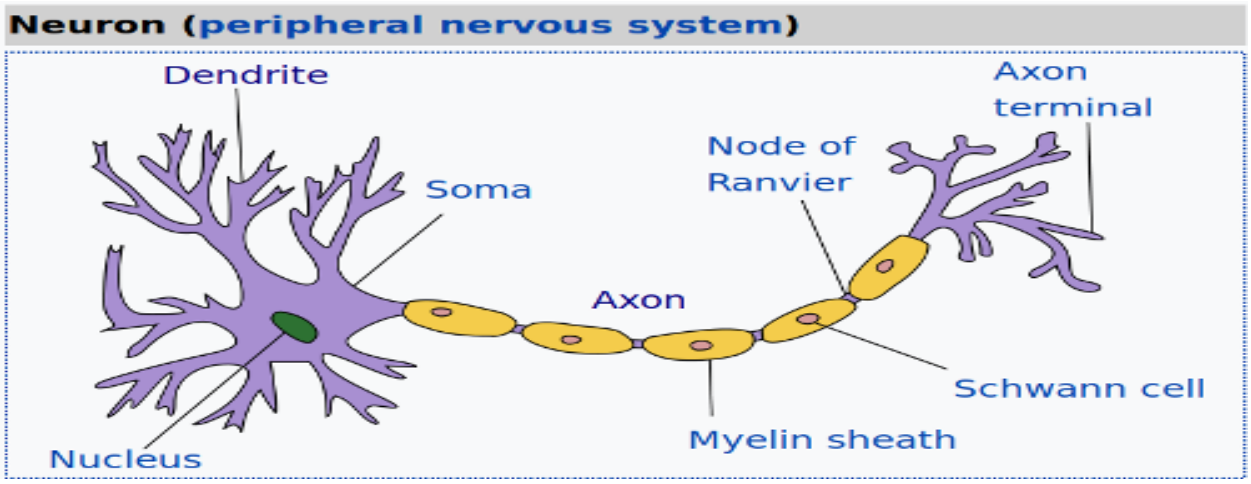
Syllabus: Artificial Neural Networks: Introduction, Neural Network representation, Appropriate problems, Perceptron, Backpropagation algorithm.

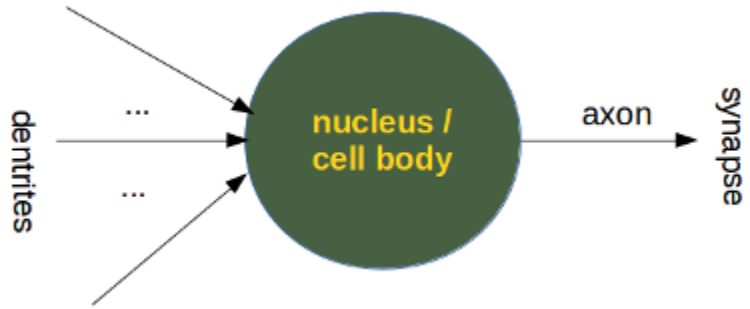
3.0 Introduction

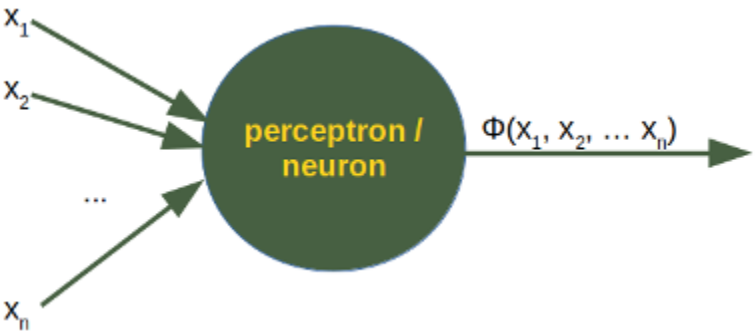
- Neural network learning methods provide a robust approach to approximating *real-valued, discrete-valued, and vector-valued target* functions.
- For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known.
- For example, the Backpropagation Algorithm described in this chapter has proven surprisingly successful in many practical problem such as learning to recognize
    - handwritten characters (LeCun et al. 1989),
    - learning to recognize spoken words (Lang et al. 1990), and
    - learning to recognize faces (Cottrell 1990).

3.1. 1 Biological Motivation

- When we say "Neural Networks", we mean artificial Neural Networks (ANN). The idea of ANN is based on biological neural networks like the brain.
- Brain is a complex web of interconnected neurons . There are 10^{11} $neurons\ and\ each\ neuron\ is\ connected\ to$ 10^{04} $neurons$
- The basic structure of a neural network is the neuron. A neuron in biology consists of three major parts: *the soma (cell body), the dendrites, and the axon*.
- The dendrites branch *off* from the soma in a tree-like way and getting thinner with every branch.
- They receive signals (impulses) from other neurons at synapses.

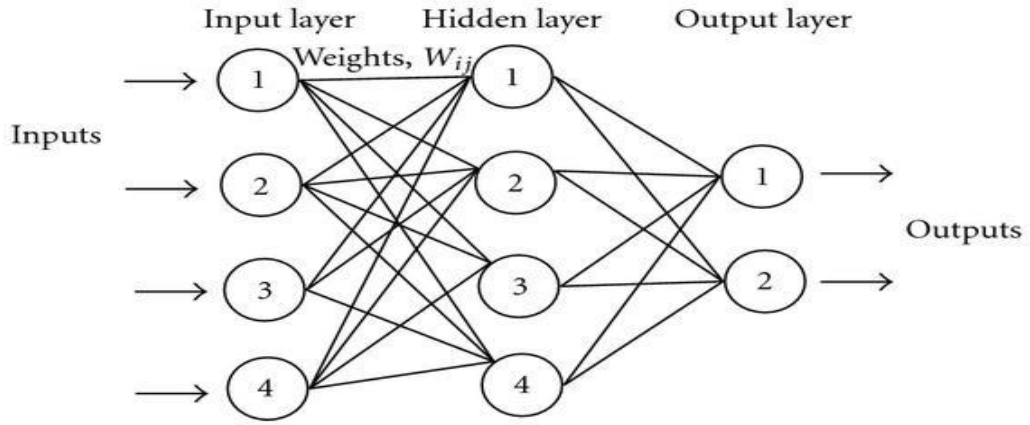

Neuron (peripheral nervous system)
Dendrite
Axon terminal
Node of Ranvier
Soma
Axon
Schwann cell
Myelin sheath
Nucleus


dentrites
...
...
nucleus / cell body
axon
synapse


x1
x2
...
xn
perceptron / neuron
Φ(x1, x2, ... xn)

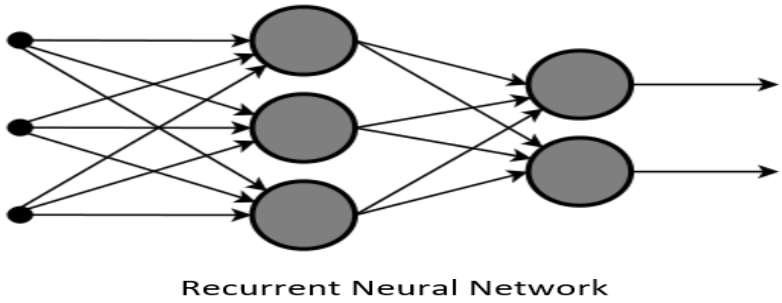
3.1.2 Artificial Neural Network

- A neural network is made up of simple processing unit called neurons. A neural net can be viewed as massively parallel distributed processor which acquires experimental knowledge from its environment through a learning process.
- The acquired knowledge is stored in the form of inter neuron connection strengths , known as synaptic weights

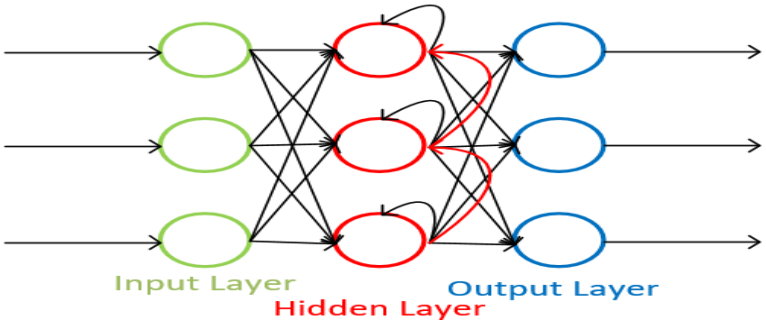


Types of ANN

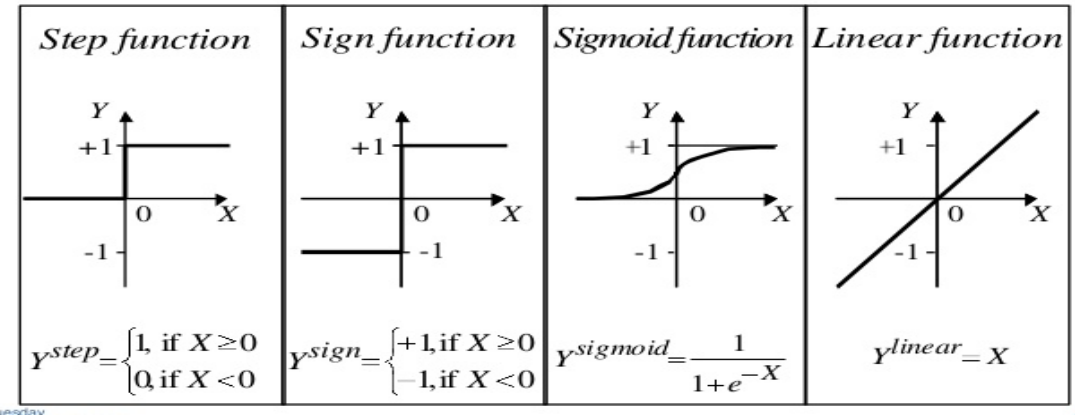
Feed Forward Neural Network



Recurrent Neural Network



Some Activation functions of a neuron



Biological Neural Network	Artificial Neural Network
Soma	Node
Dendrites	Input
Axon	Output
Synapse	Output
Slow Speed (Switching Speed : 10^{-3} seconds)	Fast Speed (Switching Speed :10^{-10} seconds)
10^{11} *neurons*	A dozen to 100 thousands neurons

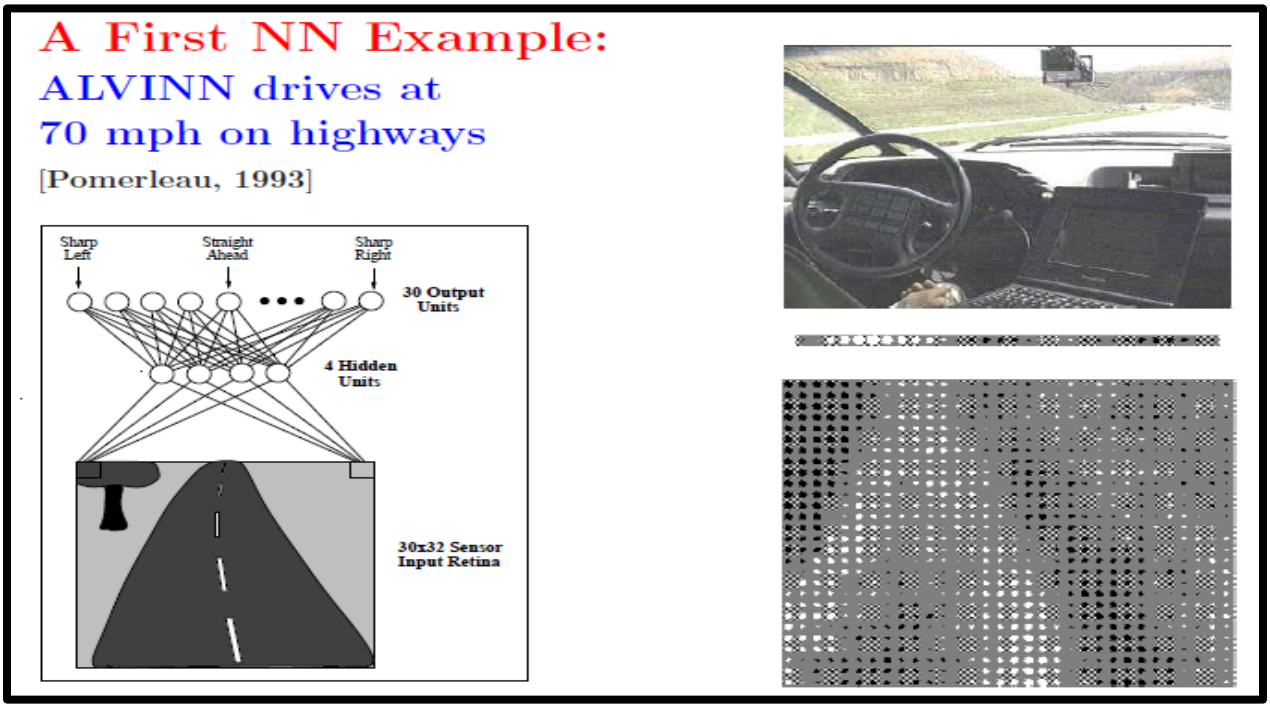
3.2. Neural Network Representation

- 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 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 (driving in the left lane of a divided public highway, with other vehicles present).

Example :

- Figure illustrates the neural network representation used in one version of the ALVINN system,
- 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.
- The network structure of ALVINN is typical of many ANNs. Here the individual units are interconnected in layers that form a directed acyclic graph. In general, ANNs can be graphs with many types of structures-acyclic or cyclic, directed or undirected.



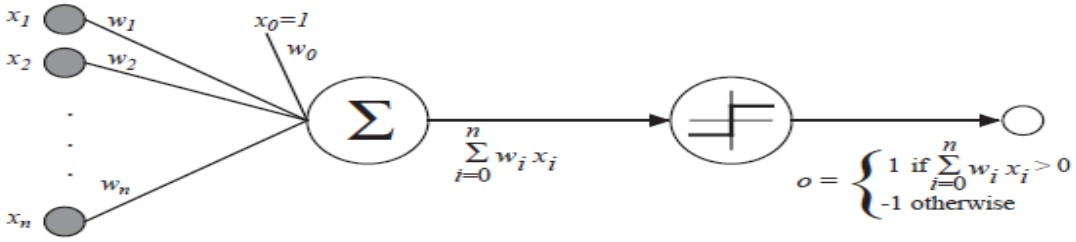
3.3. APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN learning is well-suited to problems in the following cases:

- *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.* ANN learning methods are quite robust to noise in the training data.
- *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.*

3.4. Perceptron

- A perceptron is a feedforward network with one output neuron that learns a separating hyper plane in a pattern space. The percepron is used where in the data is linearly separable.
- One type of ANN system is based on a unit called a perceptron, illustrated in Figure
- 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



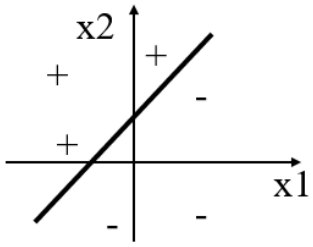
$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

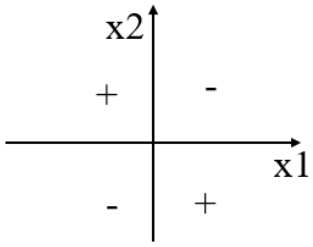
$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

3.4.1 Representational Power of Perceptron

- 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, as illustrated in Figure
- The equation for this decision hyperplane is w.x= 0.
- 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.
- Perceptrons can represent all of the primitive boolean functions AND, OR,NAND (NOT AND), and NOR (NOT OR).
- 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 x1# x2.



(a)

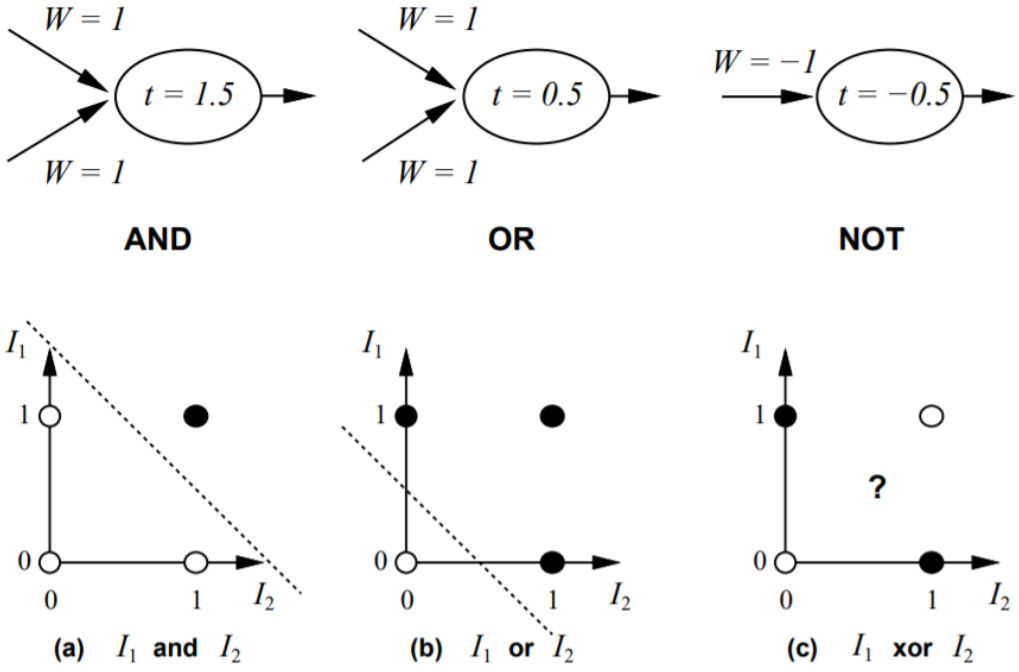


(b)

Linear Separable

Linearly Not Separable

Boolean Functions and Perceptrons

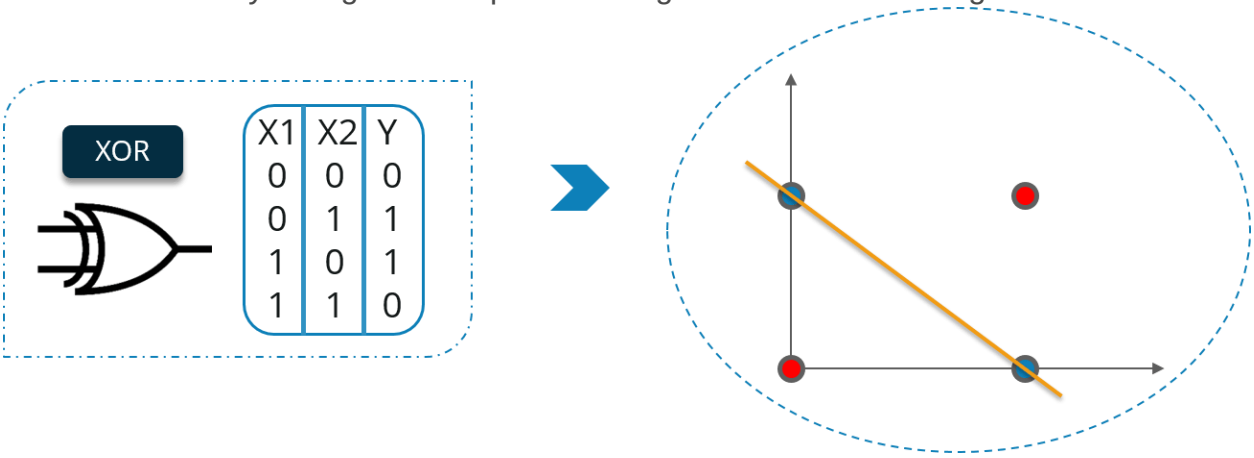


Limitations of Single-Layer Perceptron:

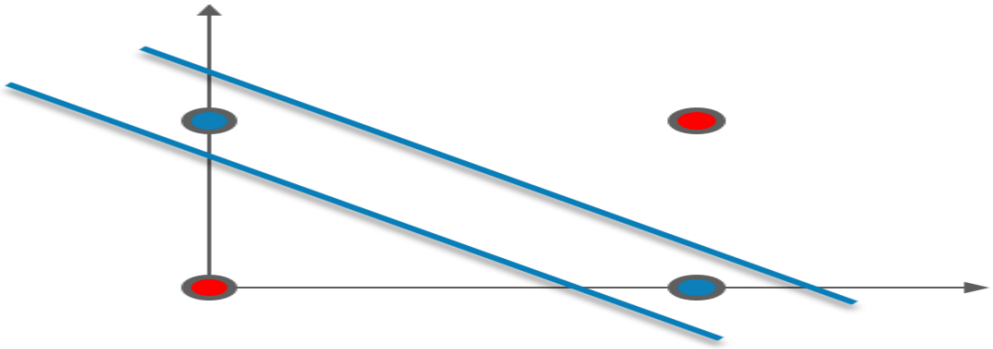
Well, there are two major problems:

- Single-Layer Percpetrons cannot classify non-linearly separable data points.
- Complex problems, that involve a lot of parameters cannot be solved by Single-Layer Perceptrons.

Single-Layer Perceptron's cannot classify non-linearly separable data points. Let us understand this by taking an example of XOR gate. Consider the diagram below:



Here, you cannot separate the high and low points with a single straight line. But, we can separate it by two straight lines. Consider the diagram below:



3.4.2 The Perceptron Training Rule

- **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 example. At each step the system weights are modified to reduced the error.
- **The Perceptron Learning Theorem (Rosenblatt, 1960):** Given enough training examples, there is an algorithm that will learn any linearly separable function.
- **Theorem 1 (Minsky and Papert, 1969):**The perceptron rule converges to weights that correctly classify all training examples provided the given data set represents a function that is linearly separable

Learning in Perceptron

Algorithm:

1.Randomly assign weights to initial network (usually values range[-0.5,0.5])
2.Repeat until all examples correctly predicated or stopping criterion is met
for each example e in training set do
i). **o** = neural-net-output(***network, e***)
ii). **t**= observed output values from **e**
iii). Update-weights in network based on **e, o, t**

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

Perceptron training rule

learning rate — error

where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small positive constant (e.g., .1) called *learning rate*

It will converge (proven by [Minsky & Papert, 1969])

- if the training data is linearly separable
- and η is sufficiently small.

4.3 Gradient Descent and the Delta Rule

- **The perceptron rule fail to converge if the examples are not linearly separable**
- **The second rule called 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**

Gradient Descent and the Delta Rule

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

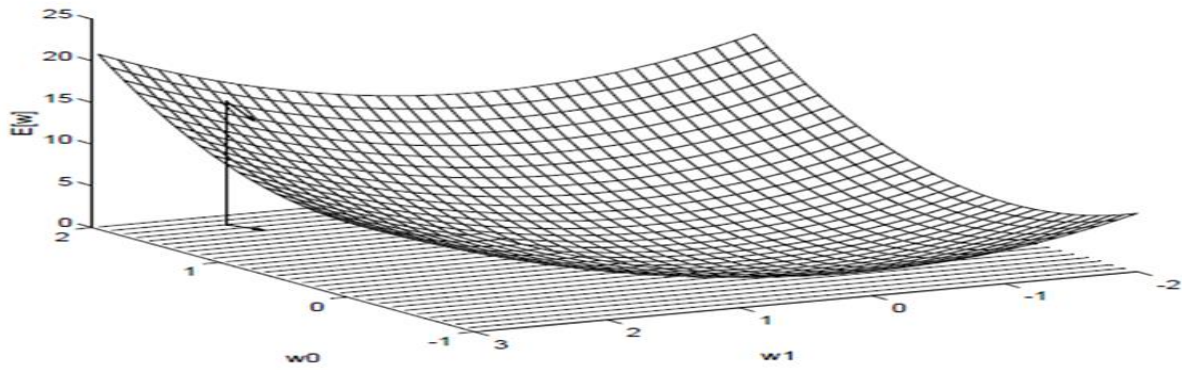
$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

- **The measure for the *training error* of a hypothesis (weight vector)**

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

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 illustrated in Figure .**



Gradient:

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

Training rule:

$$\vec{w} = \vec{w} + \Delta\vec{w},$$

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

Therefore,

$$w_i = w_i + \Delta w_i,$$

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

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

Therefore

$$\Delta w_i = \eta\sum_d (t_d - o_d)x_{i,d}$$

Convergence

[Hertz et al., 1991]

The gradient descent training rule used by the linear unit is guaranteed to converge to a hypothesis with minimum squared error

- given a sufficiently small learning rate η
- even when the training data contains noise
- even when the training data is not separable by H

Note: If η is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification of the algorithm is to gradually reduce the value of η as the number of gradient descent steps grows.

Remark

- **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

- One common variation on gradient descent intended to alleviate these difficulties is called *incremental gradient descent,* or alternatively *stochastic gradient descent.*
- *Modified training rule*

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

:

Where t, o and xi are the target value , unit output and the ith input for the training example in the question.

- *Error Function:*

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

Where td and od are the target value and the unit output value for training example d. Stochastic gradient descent iterates over the training example d in D at each iteration altering the weights according to the gradient w.r.t to Ed.

Standard Gradient descent and Stochastic Gradient Descent

Standard Gradient descent	**Stochastic Gradient Descent**
The error is summed over all examples before updating weights,	The weights are updated *upon examining each training example*
More computation per weight update step.	Less Computation per weight update Step
Step Size is Larger	Step Size is Smaller

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:	Incremental mode Gradient Descent:
Do until satisfied	Do until satisfied
1. Compute the gradient $\nabla E_D[\vec{w}]$	• For each training example d in D
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$	1. Compute the gradient $\nabla E_d[\vec{w}]$
	2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$
$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$	$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$

Covergence:

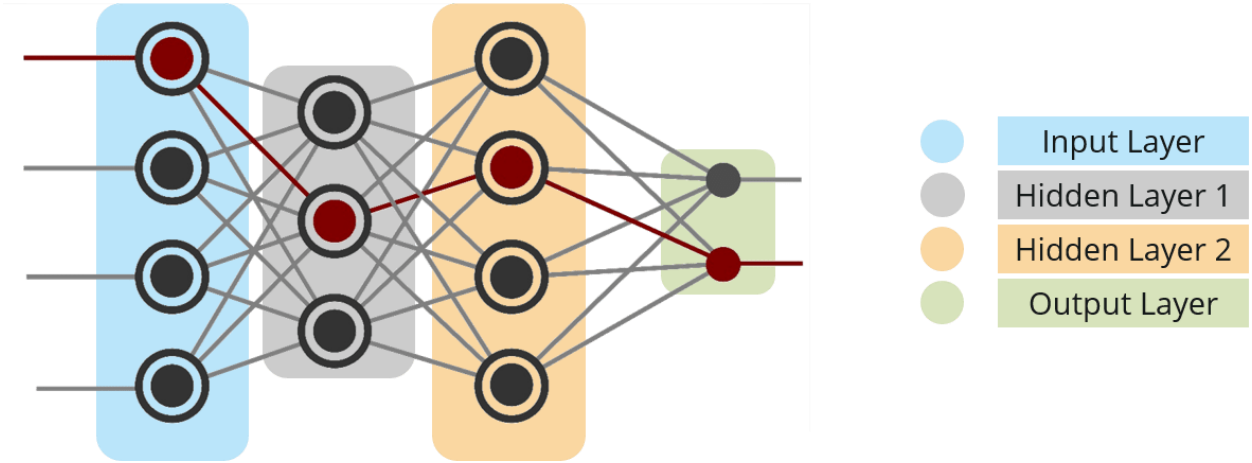
The *Incremental Gradient Descent* can approximate the *Batch Gradient Descent* arbitrarily closely if η is made small enough.

3.5. MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

What is Multi-Layer Perceptron?

As you know our brain is made up of millions of neurons, so a Neural Network is really just a composition of Perceptrons, connected in different ways and operating on different activation functions.

Consider the diagram below:



- Input Nodes – The Input nodes provide information from the outside world to the network and are together referred to as the “Input Layer”. No computation is performed in any of the Input nodes – they just pass on the information to the hidden nodes.
- Hidden Nodes – The Hidden nodes have no direct connection with the outside world (hence the name “hidden”). They perform computations and transfer information from the input nodes to the output nodes. A collection of hidden nodes forms a “Hidden Layer”. While a network will only have a single input layer and a single output layer, it can have zero or multiple Hidden Layers. A Multi-Layer Perceptron has one or more hidden layers.
- Output Nodes – The Output nodes are collectively referred to as the “Output Layer” and are responsible for computations and transferring information from the network to the outside world.

Suppose we have data of a football team, **Chelsea**. The data contains three columns. The last column tells whether Chelsea won the match or they lost it. The other two columns are about, goal lead in the first half and possession in the second half. Possession is the amount of time for which the team has the ball in percentage. So, if I say that a team has 50% possession in one half (45 minutes), it means that, the team had ball for 22.5 minutes out of 45 minutes.

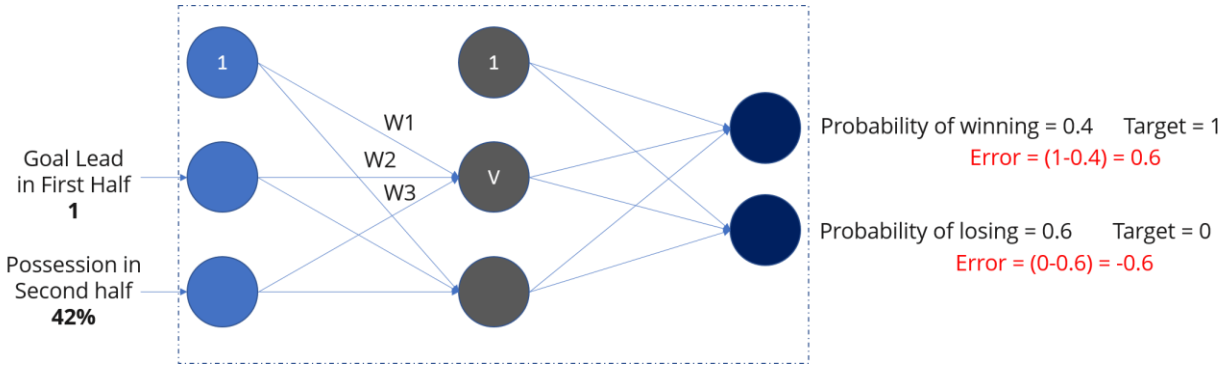
Goal Lead in First Half	Possession in Second Half	Won or Lost (1,0)?
0	80%	1
0	35%	0
1	42%	1
2	20%	0
-1	75%	1

The Final Result column, can have two values 1 or 0 indicating whether Chelsea won the match or not. For example, we can see that if there is a 0 goal lead in the first half and in next half Chelsea has 80% possession, then Chelsea wins the match.

Now, suppose, we want to predict whether Chelsea will win the match or not, if the goal lead in the first half is 2 and the possession in the second half is 32%.

This is a binary classification problem where a multi layer Perceptron can learn from the given examples (training data) and make an informed prediction given a new data point. We will see below how a multi layer perceptron learns such relationships. The process by which a Multi Layer Perceptron learns is called the **Backpropagation** algorithm

Consider the diagram below:



Forward Propagation:

Here, we will propagate forward, i.e. calculate the weighted sum of the inputs and add bias. In the output layer we will use the softmax function to get the probabilities of Chelsea winning or loosing.

If you notice the diagram, winning probability is 0.4 and loosing probability is 0.6. But, according to our data, we know that when goal lead in the first half is 1 and possession in the second half is 42% Chelsea will win. Our network has made wrong prediction.

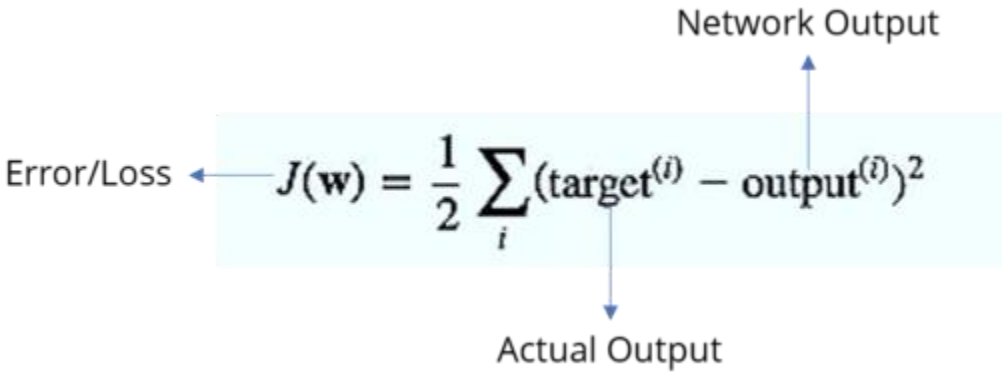
If we see the error (Comparing the network output with target), it is 0.6 and -0.6.

Backward Propagation and Weight Updation:

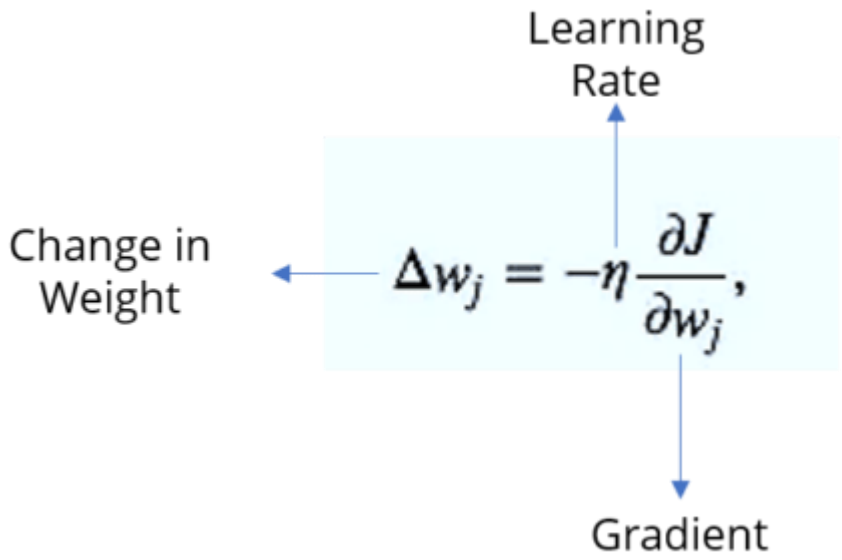
We calculate the total error at the output nodes and propagate these errors back through the network using Backpropagation to calculate the *gradients*. Then we use an optimization method such as *Gradient Descent* to 'adjust' **all** weights in the network with an aim of reducing the error at the output layer.

Let me explain you how the gradient descent optimizer works:

Step – 1: First we calculate the error, consider the equation below:

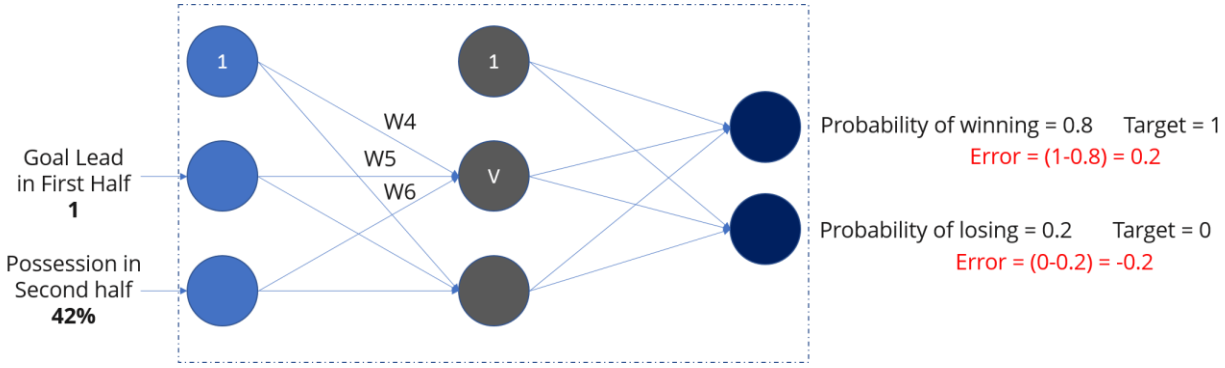


Step-2 : Based on the error we got, it will calculate the rate of change of error w.r.t change in the weights.



Step – 3: Now, based on this change in weight, we will calculate the new weight value.

If we now input the same example to the network again, the network should perform better than before since the weights have now have been adjusted to minimize the error in prediction. Consider the example below, As shown in Figure, the errors at the output nodes now reduce to [0.2, -0.2] as compared to [0.6, -0.4] earlier. This means that our network has learnt to correctly classify our first training example.

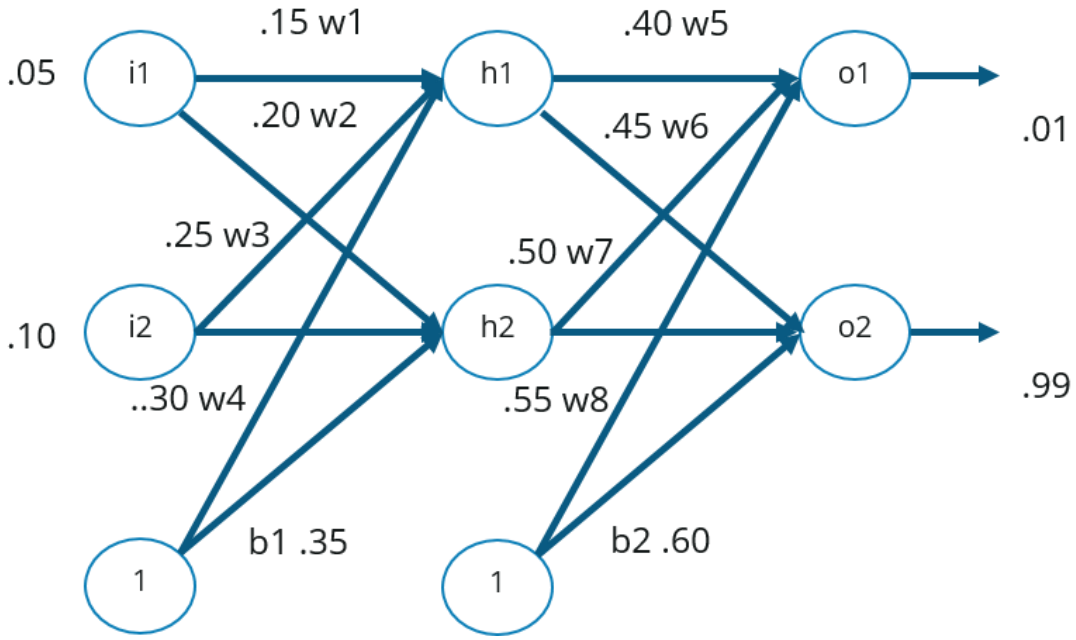


What is Backpropagation?

The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent. The weights that minimize the error function is then considered to be a solution to the learning problem.

How Backpropagation Works?

Consider the below Neural Network:



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 1: Forward Propagation
We will start by propagating forward.

Net Input For h1:

net h1 = w1*i1 + w2*i2 + b1*1 → net h1 = 0.15*0.05 + 0.2*0.1 + 0.35*1 = 0.3775

Output Of h1:

out h1 = $1/1+e^{-\text{net h1}}$ → $1/1+e^{.3775}$ = 0.593269992

Output Of h2:

out h2 = 0.596884378

We will repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

Output For o1:

net o1 = w5*out h1 + w6*out h2 + b2*1 → 0.4*0.593269992 + 0.45*0.596884378 + 0.6*1 = 1.105905967

Out o1 = $1/1+e^{-\text{net o1}}$ → $1/1+e^{-1.105905967}$ = 0.75136507

Output For o2:

Out o2 = 0.772928465

Now, let's see what is the value of the error:

Error For o1:

E o1 = $\Sigma 1/2(target - output)^2$ → ½ (0.01 – 0.75136507)² = 0.274811083

Error For o2:

E o2 = 0.023560026

Total Error:

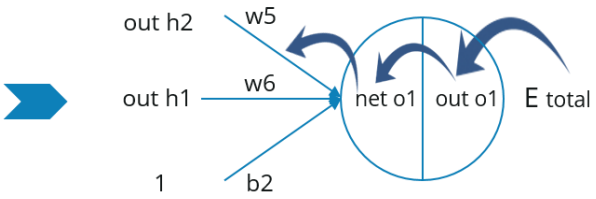
E_{total} = E o1 + E o2 → 0.274811083 + 0.023560026 = 0.298371109

Step – 2: Backward Propagation

Now, we will propagate backwards. This way we will try to reduce the error by changing the values of weights and biases.

Consider W5, we will calculate the rate of change of error w.r.t change in weight W5.

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5}$$



Since we are propagating backwards, first thing we need to do is, calculate the change in total errors w.r.t the output O1 and O2.

$$E_{total} = 1/2(\text{target o1} - \text{out o1})^2 + 1/2(\text{target o2} - \text{out o2})^2$$

$$\frac{\delta Etotal}{\delta out\ o1} = -(\text{target o1} - \text{out o1}) = -(0.01 - 0.75136507) = 0.74136507$$

Now, we will propagate further backwards and calculate the change in output O1 w.r.t to its total net input.

$$\text{out o1} = 1/1+e^{-neto1}$$

$$\frac{\delta out\ o1}{\delta net\ o1} = \text{out o1 (1 - out o1)} = 0.75136507\ (1 - 0.75136507) = 0.186815602$$

Let's see now how much does the total net input of O1 changes w.r.t W5?

$$\text{net o1} = w5 * \text{out h1} + w6 * \text{out h2} + b2 * 1$$

$$\frac{\delta net\ o1}{\delta w5} = 1 * \text{out h1}\ w5^{(1-1)} + 0 + 0 = 0.593269992$$

Step – 3: Putting all the values together and calculating the updated weight value

Now, let's put all the values together:

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5} \rightarrow 0.082167041$$

Let's calculate the updated value of W5:

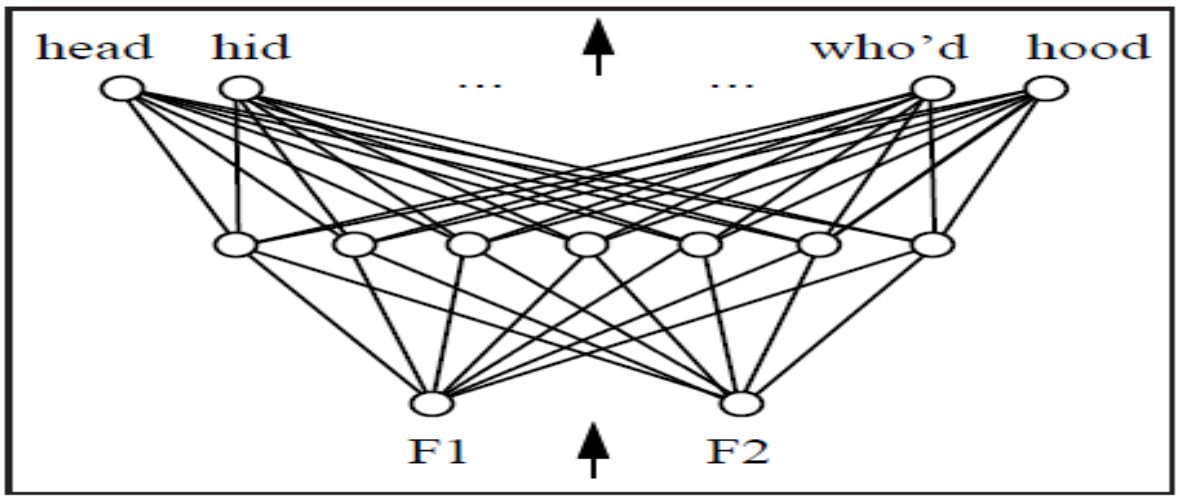
$$w5^{+} = w5 - n\frac{\delta Etotal}{\delta w5} \rightarrow w5^{+} = 0.4 - 0.5 * 0.082167041$$

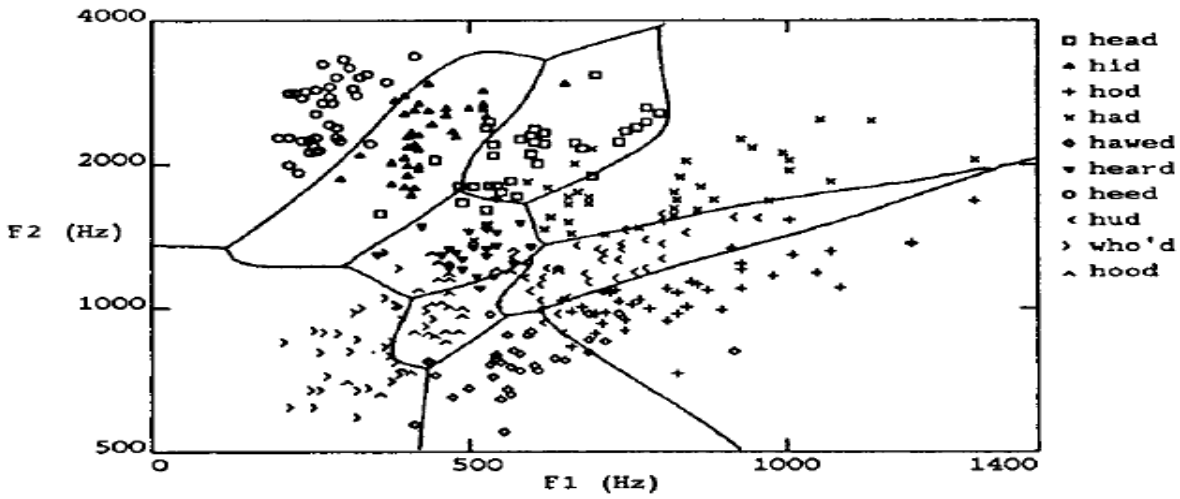
Updated w5 → 0.35891648

- Similarly, we can calculate the other weight values as well.
- After that we will again propagate forward and calculate the output. Again, we will calculate the error.
- If the error is minimum we will stop right there, else we will again propagate backwards and update the weight values.
- This process will keep on repeating until error becomes minimum.

An example

- For example, a typical multilayer network and decision surface is depicted in Figure
- Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h-d" (i.e., "hid," "had," "head," "hood," etc.).
- This network was trained to recognize 1 of 10 vowel sounds occurring in the context "h d" (e.g. "head", "hid").
- The inputs have been obtained from a spectral analysis of sound.
- The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is the highest.

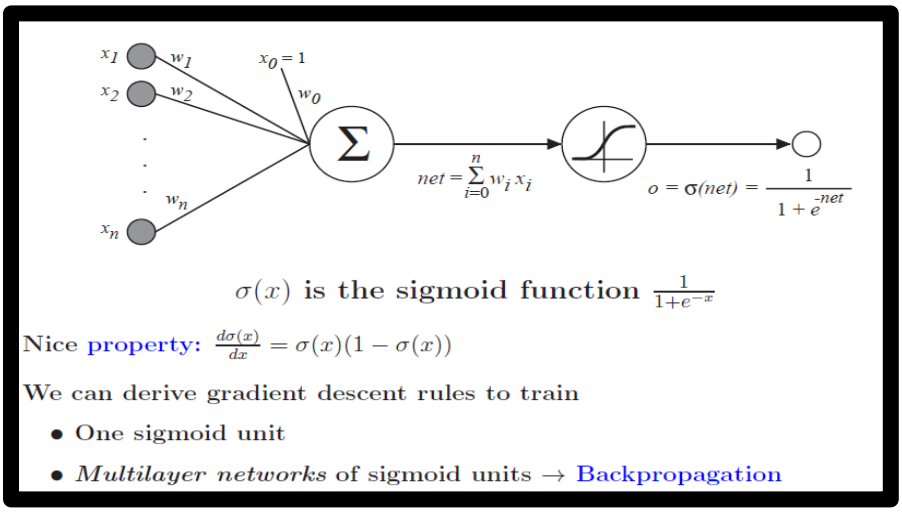




from [Haug & Lippmann, 1988]

- This plot illustrates the highly non-linear decision surface represented by the learned network.
- Points shown on the plot are test examples distinct from the examples used to train the network.

Sigmoid Threshold Unit



Error Gradient for the Sigmoid Unit

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\ &= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\ &= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\ &= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) \\ &= -\sum_d (t_d - o_d)\frac{\partial o_d}{\partial net_d}\frac{\partial net_d}{\partial w_i}\end{aligned}$$

where $net_d = \sum_{i=0}^{n} w_i x_{i,d}$

But

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\vec{w}\cdot\vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d\in D} o_d(1 - o_d)(t_d - o_d)x_{i,d}$$

Back Propagation Algorithm for Feed Forward networks

function BackProp (D, η, n_{in}, n_{hidden}, n_{out})

- D is the training set consists of m pairs: $\{(x_i, y_i)^m\}$
- η is the learning rate as an example (0.1)
- n_{in}, n_{hidden} e n_{out} are the numbero of imput hidden and output unit of neural network

Make a feed-forward network with n_{in}, n_{hidden} e n_{out} units
Initialize all the weight to short randomly number (es. [-0.05 0.05])
Repeat until termination condition are verifyed:
For any sample in D:

Forward propagate the network computing the output o_u of every unit u of the network

Back propagate the errors onto the network:

– For every output unit k, compute the error δ_k: $\delta_k = o_k(1-o_k)(t_k - o_k)$

– For every hidden unit h compute the error δ_h: $\delta_h = o_h(1-o_h)\sum_{k\in outputs} w_{kh}\delta_k$

– Update the network weight w_{ji}: $w_{ji} = w_{ji} + \Delta w_{ji}$, where $\Delta w_{ji} = \eta\delta_j x_{ji}$

(x_{ji} is the input of unit j from coming from unit i)

Derivation of the Backpropagation rule,

Notations:

x_{ji}: the ith input to unit j;
(j could be either hidden or output unit)

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

$net_j = \sum_i w_{ji} x_{ji}$

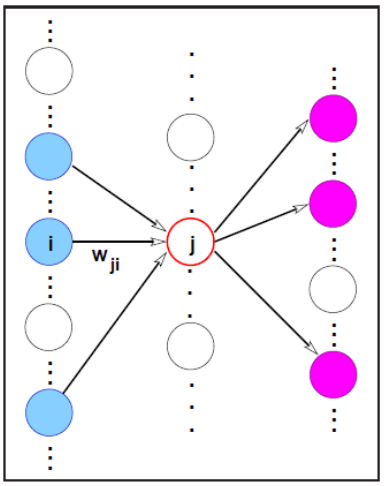
σ: the sigmoid function

o_j: the output computed by unit j; ($o_j = \sigma(net_j)$)

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

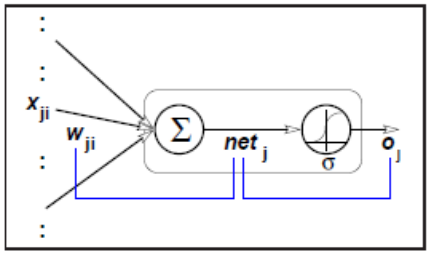
E_d: the training error on the example d (summing over all of the network output units)



Legend: in magenta color, units belonging to *Downstream(j)*

Preliminaries

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

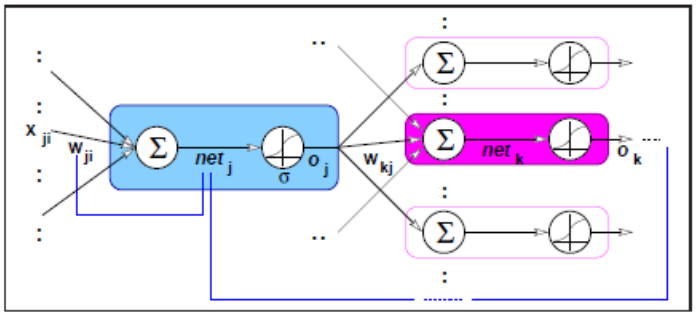


Common staff for both hidden and output units:

$$net_j = \sum_i w_{ji} x_{ji}$$

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

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



Note: In the sequel we will use the notation: $\delta_j = -\frac{\partial E_d}{\partial net_j} \Rightarrow \Delta w_{ji} = \eta \delta_j x_{ji}$

Stage/Case 1: Computing the increments (Δ) for output unit weights

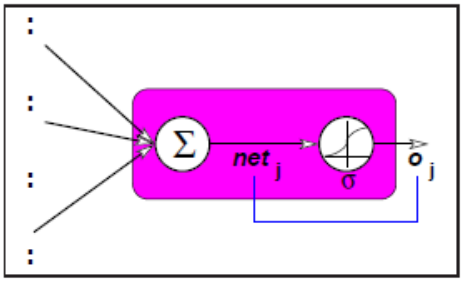
$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

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

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

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

$$= -(t_j-o_j)$$



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

$$\Rightarrow \delta_j \overset{not.}{=} -\frac{\partial E_d}{\partial net_j} = o_j(1-o_j)(t_j-o_j)$$

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

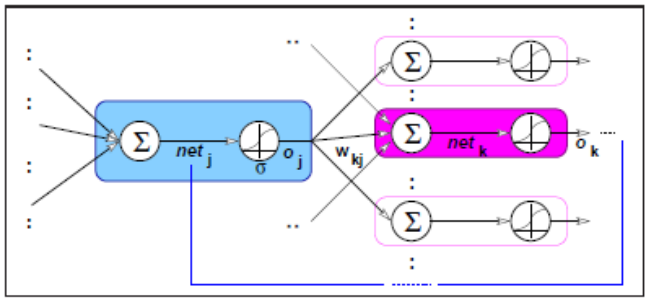
Stage/Case 2: Computing the increments (Δ) for hidden unit weights

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

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

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

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



Therefore:

$$\delta_j \overset{not}{=} -\frac{\partial E_d}{\partial net_j} = o_j(1-o_j)\sum_{k\in Downstream(j)}\delta_k w_{kj}$$

$$\Delta w_{ji} \overset{def}{=} -\eta\frac{\partial E_d}{\partial w_{ji}} = -\eta\frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} = -\eta\frac{\partial E_d}{\partial net_j}x_{ji} = \eta\delta_j x_{ji} = \eta\ [\ o_j(1-o_j)\sum_{k\in Downstream(j)}\delta_k w_{kj}\ ]\ x_{ji}$$

3.6. REMARKS ON THE BACKPROPAGATION ALGORITHM

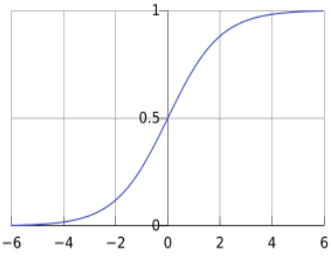
- Convergence and Local Minima
- Representational Power of Feedforward Networks (Boolean , Continous and Arbitrary)
- Hypothesis Space Search and Inductive Bias
- Hidden Layer Representations
- Generalization, Overfitting, and Stopping Criterion

Convergence of Backpropagation for NNs of Sigmoid units

Nature of convergence

- The weights are initialized near zero; therefore, initial decision surfaces are near-linear.

  Explanation: o_j is of the form $\sigma(\vec{w} \cdot \vec{x})$, therefore $w_{ji} \approx 0$ for all i, j; note that the graph of σ is approximately liniar in the vecinity of 0.



- Increasingly non-linear functions are possible as training progresses
- Will find a local, not necessarily global error minimum. In practice, often works well (can run multiple times).

More on Backpropagation

- Easily generalized to arbitrary directed graphs
- Training can take thousands of iterations → slow!
- Often include weight momentum α

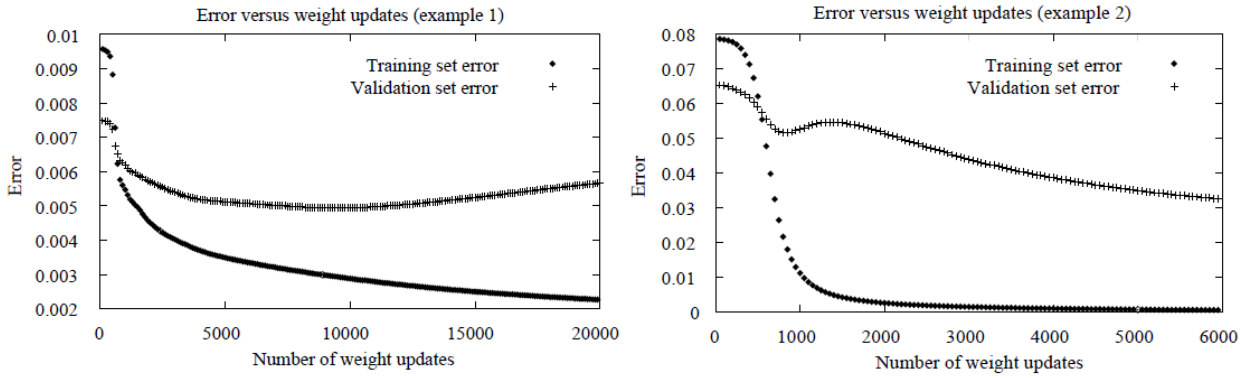
$$\Delta w_{i,j}(n) = \eta \delta_j x_{ij} + \alpha \Delta w_{ij}(n-1)$$

Effect:

  - speed up convergence (increase the step size in regions where the gradient is unchanging);
  - "keep the ball rolling" through local minima (or along flat regions) in the error surface

- Using network after training is very fast
- Minimizes error over *training* examples;
  Will it generalize well to subsequent examples?

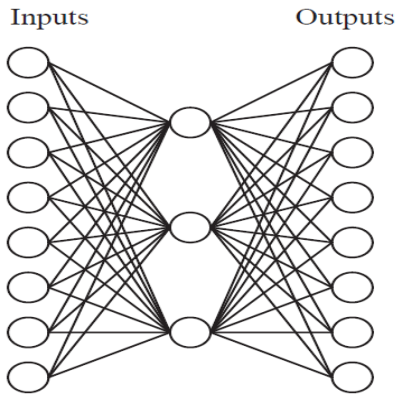
Stopping Criteria when Training ANNs and Overfitting



Plots of the error E, as a function of the number of weights updates, for two different robot perception tasks.

Learning Hidden Layer Representations

An Example: Learning the identity function $f(\vec{x}) = \vec{x}$

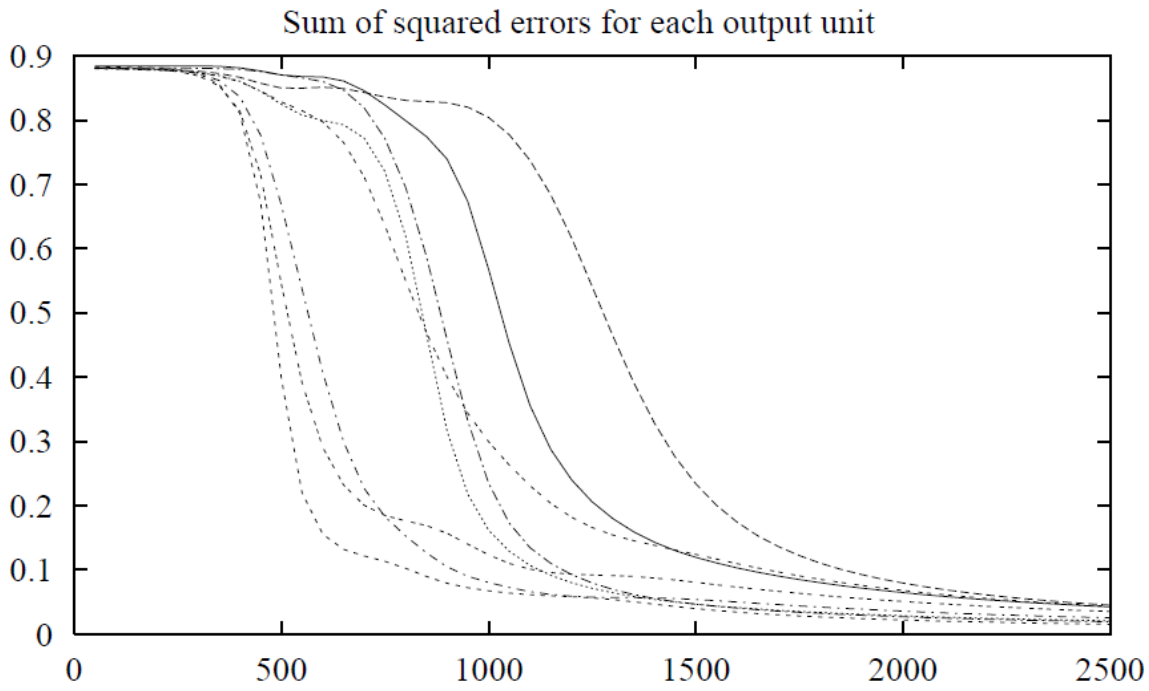


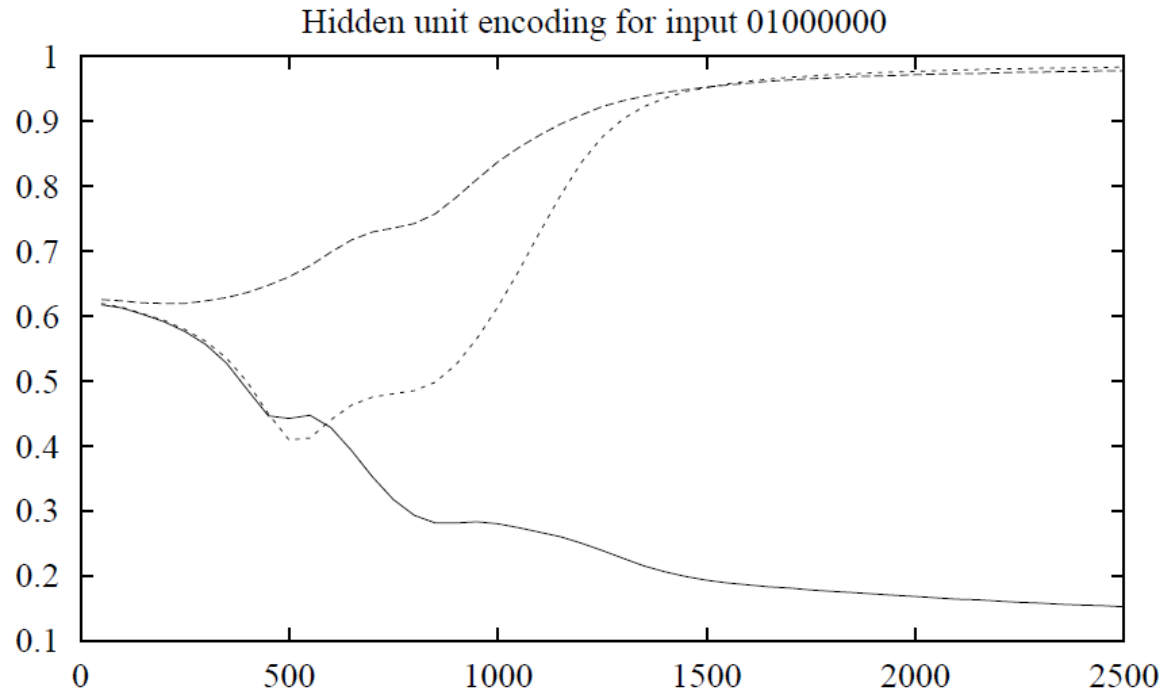
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

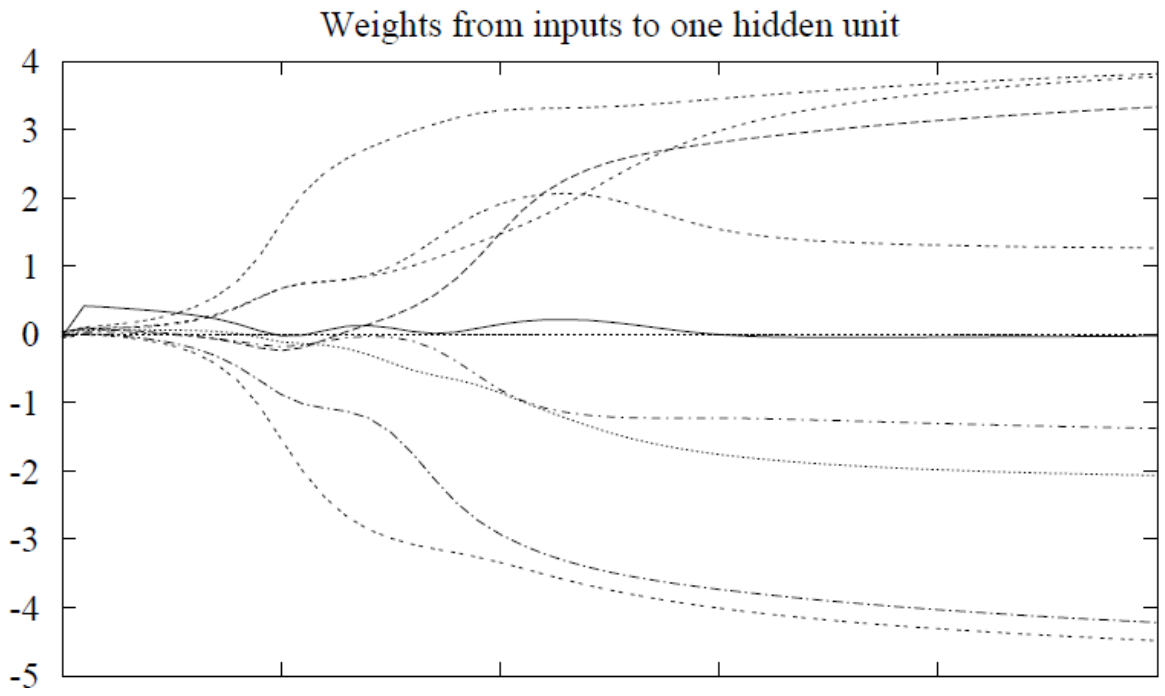
Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

After 8000 training epochs, the 3 hidden unit values encode the 8 distinct inputs. Note that if the encoded values are rounded to 0 or 1, the result is the standard binary encoding for 8 distinct values (however not the usual one, i.e. $1 \rightarrow 001$, $2 \rightarrow 010$, etc).


Sum of squared errors for each output unit
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500


Hidden unit encoding for input 01000000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
500
1000
1500
2000
2500



7. Alternative Error Functions

- Penalize large weights;

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

- Train on target slopes as well as values

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

- Tie together weights: e.g., in phoneme recognition network;
- Minimizing the cross entropy (see next 3 slides):

$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log(1 - o_d)$$

where o_d, the output of the network, represents the estimated probability that the training instance x_d is associated the label (target value) 1.

Question Bank

Module -3 Questions.

1) What is Artificial Neural Network?
2) What are the type of problems in which Artificial Neural Network can be applied.
3) Explain the concept of a Perceptron with a neat diagram.
4) Discuss the Perceptron training rule.
5) Under what conditions the perceptron rule fails and it becomes necessary to apply the delta rule
6) What do you mean by Gradient Descent?
7) Derive the Gradient Descent Rule.
8) What are the conditions in which Gradient Descent is applied.
9) What are the difficulties in applying Gradient Descent.
10) Differentiate between Gradient Descent and Stochastic Gradient Descent
11) Define Delta Rule.
12) Derive the Backpropagation rule considering the training rule for Output Unit weights and Training Rule for Hidden Unit weights
13) Write the algorithm for Back propagation.
14) Explain how to learn Multilayer Networks using Gradient Descent Algorithm.
15) What is Squashing Function?

Question Bank Set 2

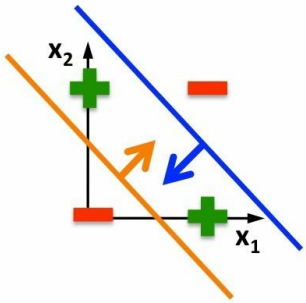
1. Define ANN. Explain the different types of ANN. Give some examples of the Applications of ANN.
2. Compare the Biological Neurons (Neural Network) with Artificial Neurons (Neural Network).
3. Explain the working ALVINN System.
4. What are the appropriate problems of Neural Network?
5. **Explain the following with examples:**
    a. **Artificial Neural Networks**
    b. **Perceptron**
    c. **Single and Multilayer Perceptron (NN)**
    d. **Activation Function**
    e. **Sigmoid Threshold Unit**
    f. **Error Gradient for Sigmoid unit**
6. When one must consider Neural Network
7. Describe "Neural Network Representation"
8. With neat diagram explain the following:
    a. Perceptron
    b. Representational Power of Perceptron

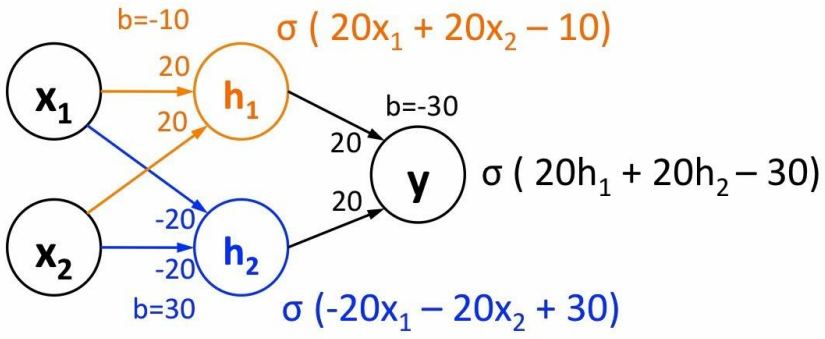
c. Perceptron Training rule and Learning in Perceptron.
d. Gradient Descent and Delta Rule

9. Derive the Gradient Descent Rule for the linear unit
10. Discuss "*Gradient Descent Algorithm for the linear unit*".
11. Write a note on Convergence of "*Gradient Descent Training Rule* "
12. Discuss Remark on Gradient Descent Training Rule. What are the practical difficulties in applying gradient descent.
13. Explain Stochastic approximation to gradient descent.
14. Differentiate between Standard Gradient Descent and Stochastic Gradient Descent.
15. Explain with example Multilayer Neural Networks (Multilayer Perceptron).
16. What is Sigmoid Threshold Unit? Derive the relation for Error gradient for Sigmoid Unit.
17. Write and explain Back Propagation Algorithm. Derive the following of the Backpropagation Rule:
a. Error at the output unit
b. Error at the hidden unit
c. Weight to be updated
18. Discuss all remarks of Backpropagation Algorithm.
19. What is linearly in separable problem? Design a two-layer network of perceptron to implement A OR B , A AND B & NOT A.
20. Consider a multilayer feed forward neural network. Enumerate and explain steps in back propagation algorithm use to train network
21. What are the steps in Back propagation algorithm? Why a Multilayer neural network is required?
22. What is Multilayer perception? How is it trained using Back propagation? What is linear separability issue? What is the role of hidden layer?
23. Explain how back propagation algorithm works for multilayer feed forward network.
24. Explain perceptron and Delta training rule.
25. Explain the differential sigmoid threshold unit.
26. Consider two perceptron's defined by the threshold expression $w0+w1x1+w2x2>0$, perceptron **A** has weight values $w0=1$, $w1= 2$ $w2 = 1$ and perceptron **B** has weight values $w0 = 0$, $w2 = 2$ and $w2 = 1$.
a. TRUE/FALSE: Perceptron A is more general than perceptron B.
27. Explain the back-propagation algorithm. Why is not likely to be trapped in local minima.
28. Explain Stochastic approximation to gradient Descent
29. What are the advantages and limitations of gradient descent.
30. Derive the Following:
a. Gradient Descent Rule
b. Back Propagation Rule

Solving XOR with a Neural Net

Linear classifiers
cannot solve this





σ(20*0 + 20*0 – 10) ≈ 0
σ(20*1 + 20*1 – 10) ≈ 1
σ(20*0 + 20*1 – 10) ≈ 1
σ(20*1 + 20*0 – 10) ≈ 1

σ (-20*0 – 20*0 + 30) ≈ 1
σ (-20*1 – 20*1 + 30) ≈ 0
σ (-20*0 – 20*1 + 30) ≈ 1
σ (-20*1 – 20*0 + 30) ≈ 1

σ (20*0 + 20*1 – 30) ≈ 0
σ (20*1 + 20*0 – 30) ≈ 0
σ (20*1 + 20*1 – 30) ≈ 1
σ (20*1 + 20*1 – 30) ≈ 1