

# EE5904/ME5404:

## Lecture Three

### Multi-layer Perceptron: Back Propagation



**Xiang Cheng**

Associate Professor

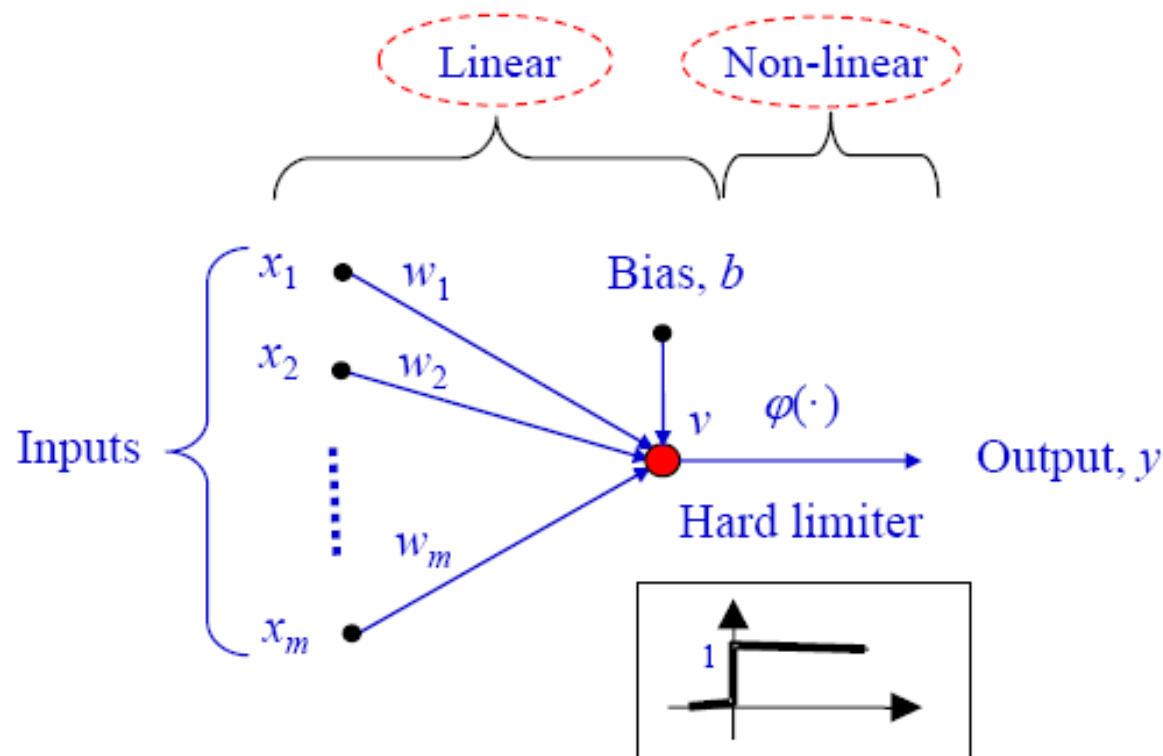
Department of Electrical & Computer Engineering  
The National University of Singapore

Phone: 65166210 Office: Block E4-08-07

Email: [elexc@nus.edu.sg](mailto:elexc@nus.edu.sg)

Perceptron is built by Frank Rosenblatt in 1958.

## A Simple Perceptron



$$v = \sum_{i=1}^m w_i x_i + b$$

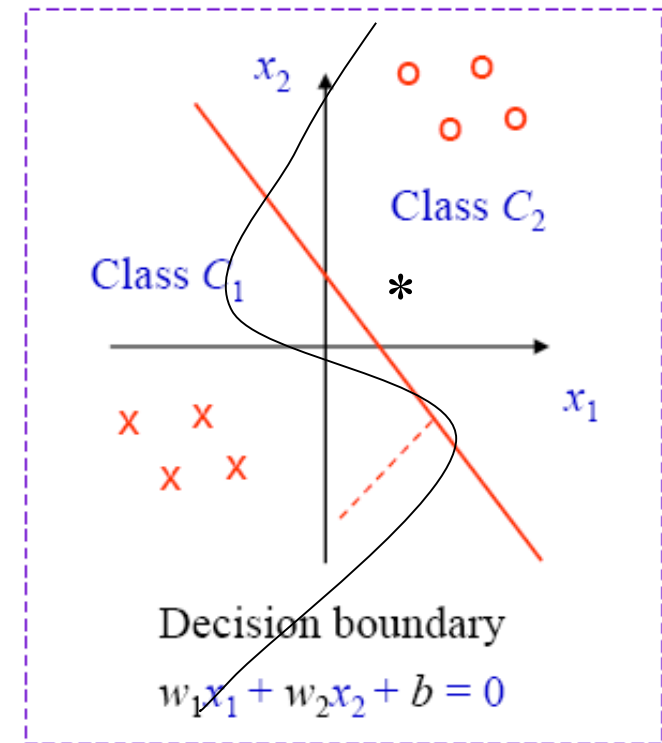
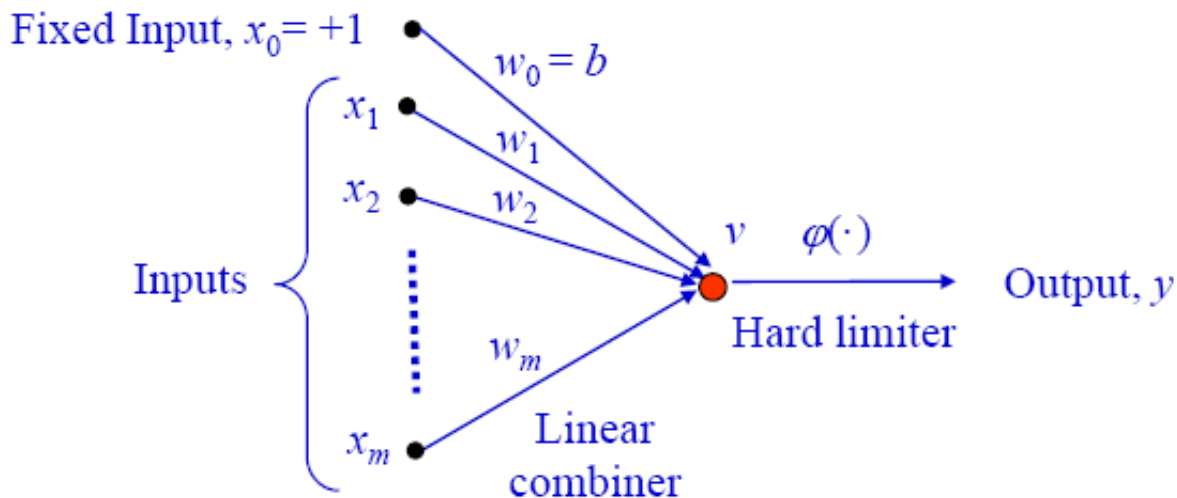
What is the difference between McCulloch-Pitts model and Perceptron?

Learning!

# What type of problem is suitable for classical Perceptron? Pattern Recognition or Regression? And why?

Pattern Recognition:

Goal: To correctly classify the set of externally applied stimuli  $x_1, x_2, \dots, x_n$  into one of two classes,  $C_1$  and  $C_2$ .



What is the equation describing the decision boundary produced by the perceptron?

$$v = w_1x_1 + w_2x_2 + \dots + w_mx_m + b = 0$$

What is the geometrical meaning of this equation?

Hyper-plane.

Is it possible for perceptron to produce a nonlinear decision boundary?

No.

## Can perceptron separate any two classes of patterns?

### Linearly Separable

If two classes can be separated by one line (plane or hyper-plane in higher dimensional space).

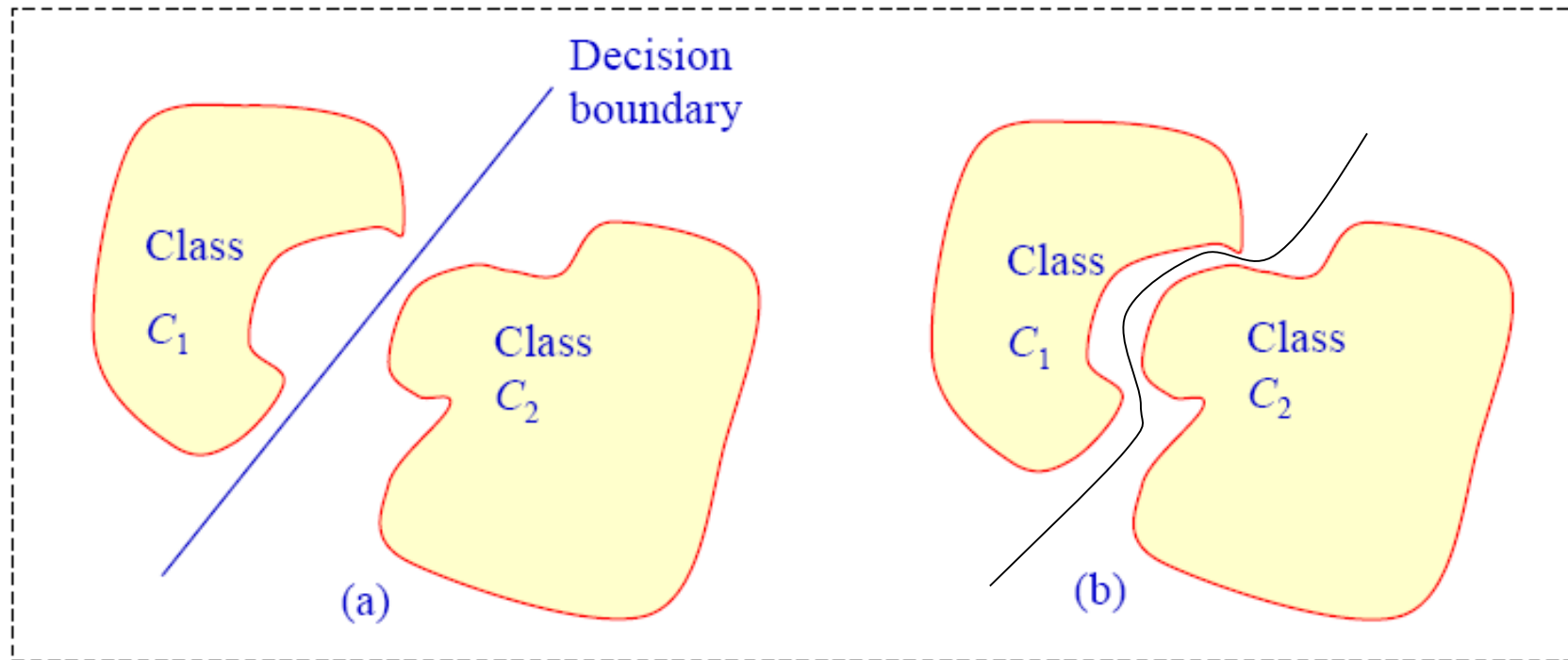


Figure: (a) A pair of linearly separable patterns; (b) A pair of non-linearly separable patterns.

Two classes are *linearly separable* if and only if there exists a weight vector  $w$  based on which the perceptron can correctly perform the classification.

## How to choose the proper weights (the proper decision boundary)?

By off-line calculation of weights (without learning) if the problem is relatively simple in lower dimensional space.

If the problem is more complex, we can use

### Perceptron learning algorithm

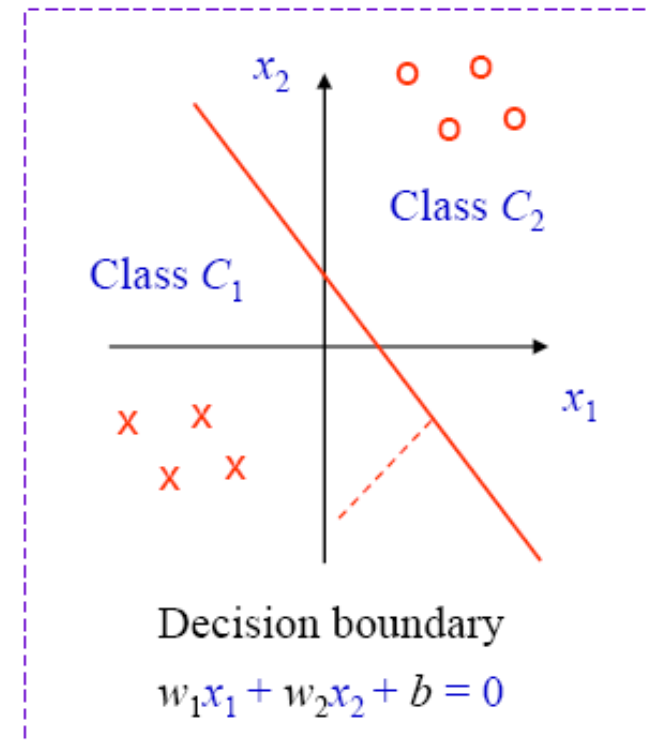
Start with a randomly chosen weight vector  $w(I)$ ;

Update the weight vector by the error-correction-learning rule

$$w(n+1) = w(n) + \eta e(n)x(n)$$

$$e(n) = d(n) - y(n)$$

$$\text{where } \eta > 0 \text{ and } d = \begin{cases} 1 & \text{if } x \text{ belongs to class } C_1 \\ 0 & \text{if } x \text{ belongs to class } C_2 \end{cases}$$

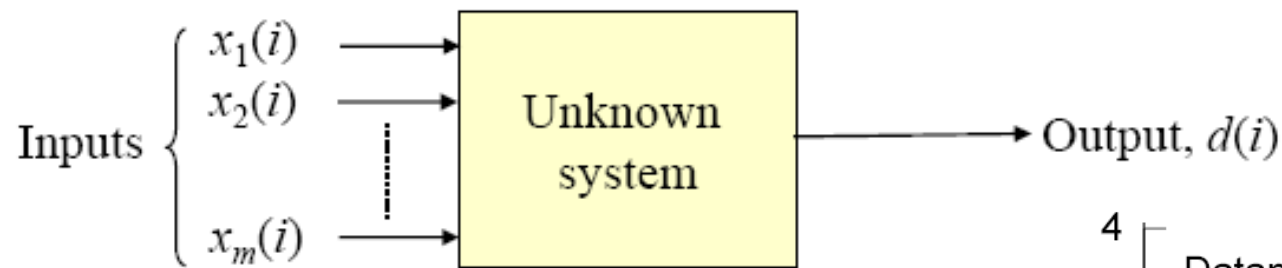


## Does it always converge to the correct solution?

If the patterns are linearly separable, then the weights will converge properly in finite steps.

## Regression Problem

Consider a multiple input-single output system whose mathematical characterization is unknown:



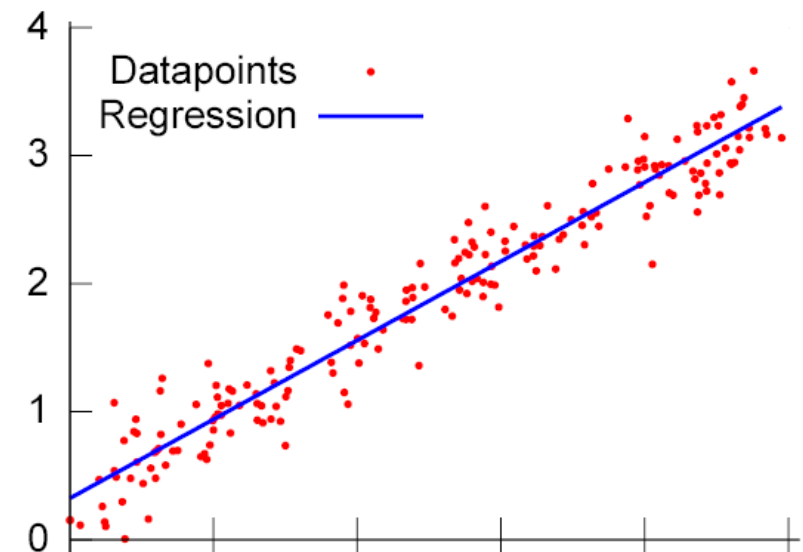
Given a set of observations of input-output data:

$$T: \{\mathbf{x}(i), d(i); i = 1, 2, \dots, n\}$$

$$\text{where } \mathbf{x}(i) = [x_1(i), x_2(i), \dots, x_m(i)]^T$$

$m$  = dimensionality of the input space;  $i$  = time index.

**How to design a model for the unknown system?**



*Optimization problem: Minimize the cost function!*

**How to evaluate the fitting results? Which one is better? Red or Black?**

**What is the most common cost function?**

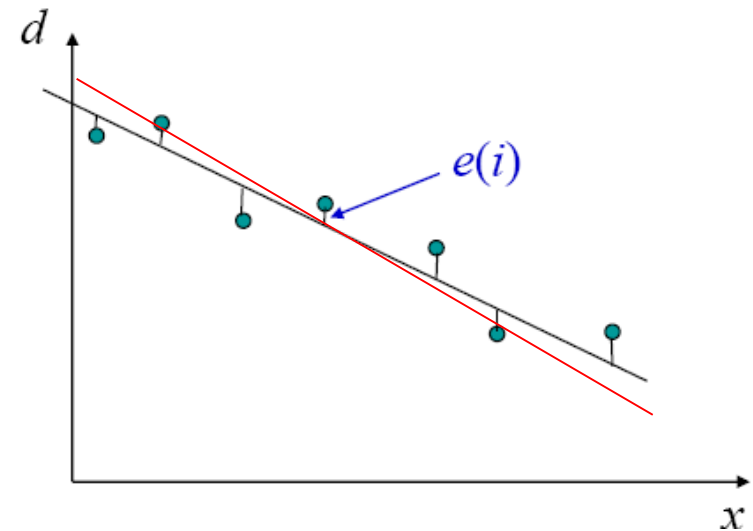
$$E(w) = \sum_{i=1}^n e(i)^2 = \sum_{i=1}^n (d(i) - y(i))^2$$

**What is the Optimality Condition?**

$$\frac{\partial E}{\partial w} = 0$$

There are two ways to solve the problem:

If the model is simple, then directly solve  $\frac{\partial E}{\partial w} = 0$



Iterative descent algorithm: Starting with an initial guess denoted by  $w(0)$ , generate a sequence of weight vectors  $w(1), w(2), \dots$ , such that the cost function

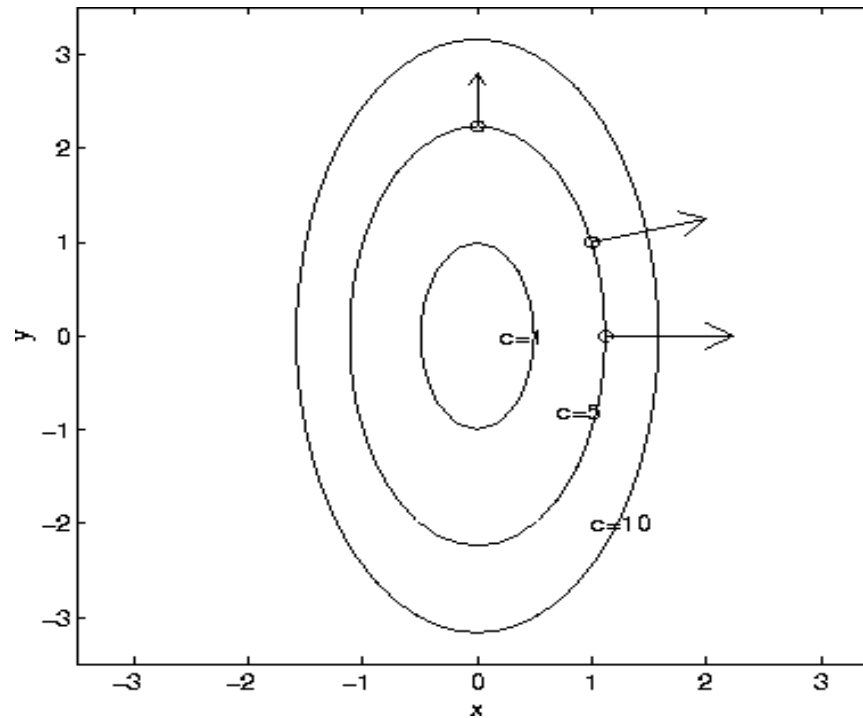
$E(w)$  is reduced at each iteration of the algorithm,

$$E(w(n+1)) < E(w(n))$$

Big question: How to choose the iterative algorithm such that the cost is always decreasing ?

**What is the simplest way if the gradient is known?**

# Method of Steepest Descent (Gradient Descent)



Successive adjustment applied to the weight vector  $\mathbf{w}$  are in the direction of steepest descent (a direction opposite to the gradient vector  $\nabla E(\mathbf{w})$ ).

Let  $\mathbf{g}(n) = \nabla E(\mathbf{w}(n))$ , steepest descent algorithm is formally described by

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta \mathbf{g}(n)$$

where  $\eta$  is a positive constant called the stepsize or **learning-rate**.

Is there any condition on the **learning-rate** to make the algorithm work?

**Sufficiently Small!**



## Linear Regression Problem

Consider that we are trying to fit a linear model to a set of input-output pairs  $(\mathbf{x}(1), d(1)), (\mathbf{x}(2), d(2)) \dots, (\mathbf{x}(n), d(n))$  observed in an interval of duration  $n$ .

$$y(\mathbf{x}) = w_1 x_1 + w_2 x_2 + \dots + w_m x_m + b$$

Cost Function:

$$E(w) = \sum_{i=1}^n e(i)^2 = \sum_{i=1}^n (d(i) - y(i))^2$$

Of course, the answer can be easily found by solving

$$\frac{\partial E}{\partial w} = 0$$

### Standard Linear Least Squares

$$w = (X^T X)^{-1} X^T d$$

where

$$d = [d(1) \quad d(2) \quad \dots \quad d(n)]^T$$

Regression matrix:

$$X = \begin{bmatrix} x(1)^T \\ x(2)^T \\ \vdots \\ x(n)^T \end{bmatrix}$$

## Linear Regression Problem

Consider that we are trying to fit a linear model to a set of input-output pairs  $(\mathbf{x}(1), d(1)), (\mathbf{x}(2), d(2)) \dots, (\mathbf{x}(n), d(n))$  observed in an interval of duration  $n$ .

$$y(\mathbf{x}) = w_1 x_1 + w_2 x_2 + \dots + w_m x_m + b$$

Can we directly use Rosenblatt's perceptron to solve this linear regression problem?

**No.** The output of the perceptron is either 1 or 0 due to the hard limiter.

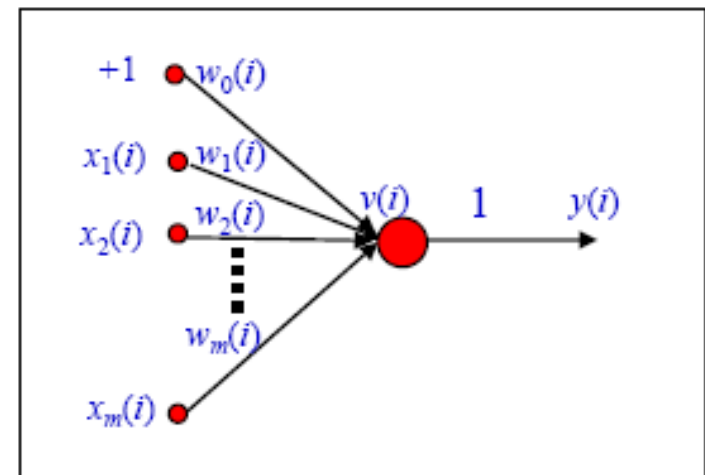
What is the simplest way to make the range of the output continuous instead of binary?

*Linear Neuron*  
(single-layer perceptron without squash function)

$$y(i) = v(i) = \sum_{k=0}^m w_k(i) x_k(i) = \mathbf{w}^T(i) \cdot \mathbf{x}(i)$$

Can the linear neuron learn the function by itself just like the perceptron?

**Yes.**



## The Least-Mean-Square algorithm:

Given  $n$  training samples:  $\{\mathbf{x}(i), d(i)\}, i = 1, 2, \dots, n$

$$e(n) = d(n) - \mathbf{w}^T(n)\mathbf{x}(n)$$

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta e(n)\mathbf{x}(n)$$

Does it take the same form as that for the Perceptron?

Yes.

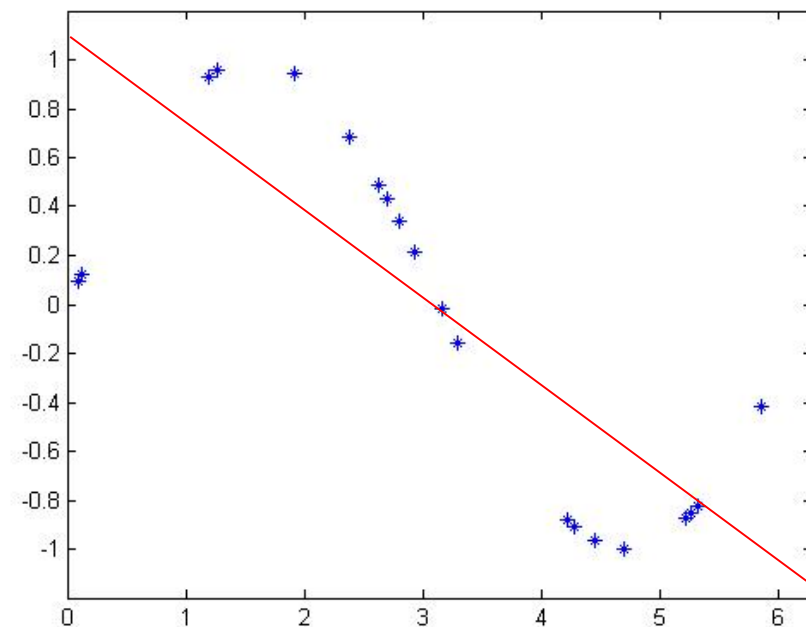
What is the advantage of LMS compared to standard linear least squares?

It requires much less memory space, and can be updated with new data easily.

Can linear neuron solve the regression problem where the underlying process is nonlinear?

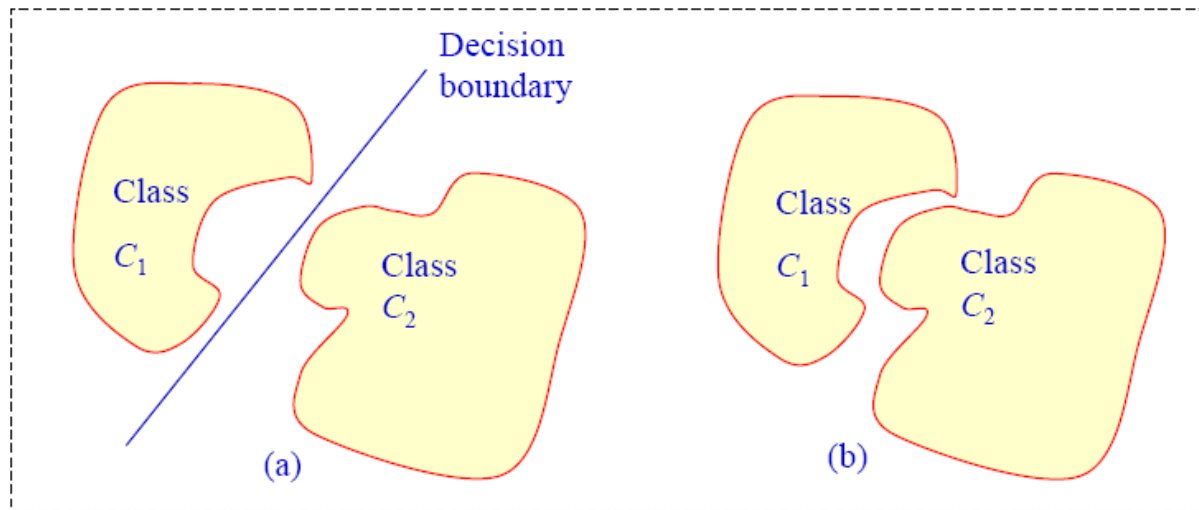
Example: Sinusoid function:  
 $y = \sin(x)$  (unknown to you), only  
sampling points are provided.

Single layer perceptron can only  
solve linear regression problem!

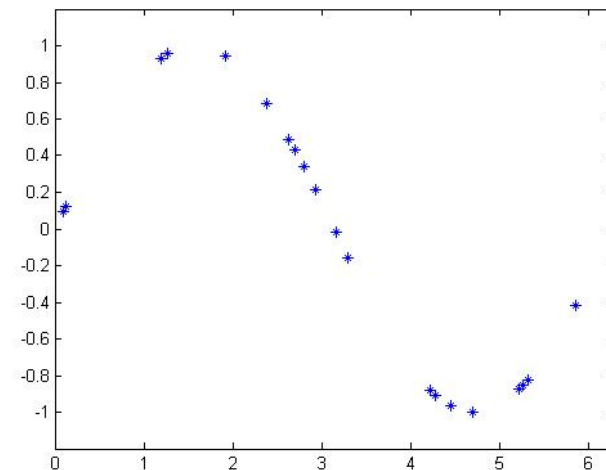
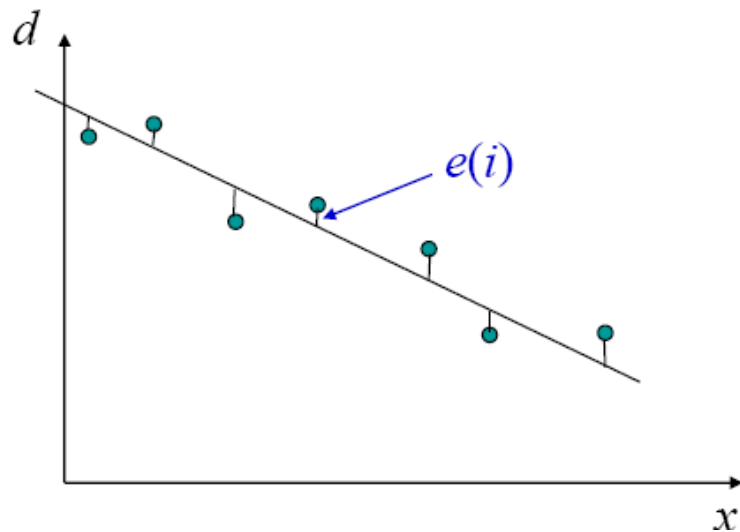


## The fundamental limits of Single Layer Perceptrons

### For Pattern Recognition Problem: Linearly Separable



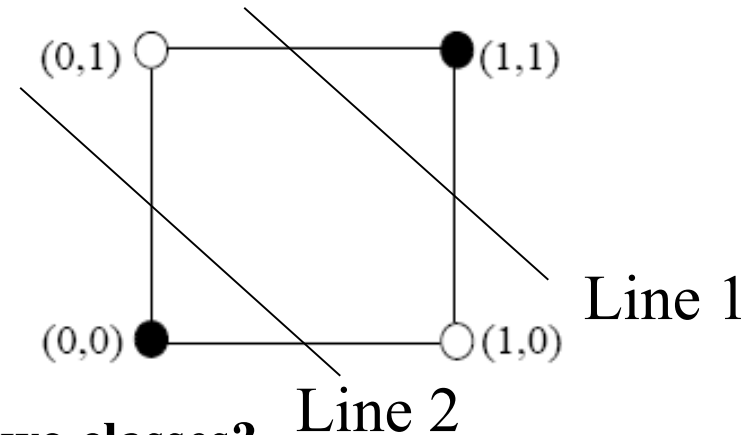
For Regression Problem: The process has to be close to a linear model!



# What is the simple logic gate problem that **killed** perceptron?

Let's consider the XOR truth table:

Input		Output
$x_1$	$x_2$	
0	0	0
1	0	1
0	1	1
1	1	0



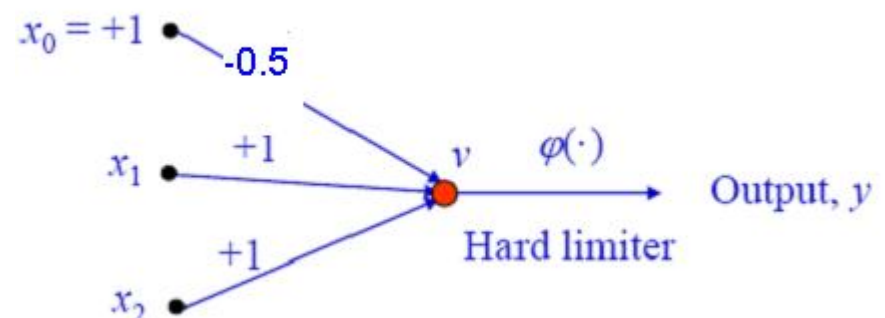
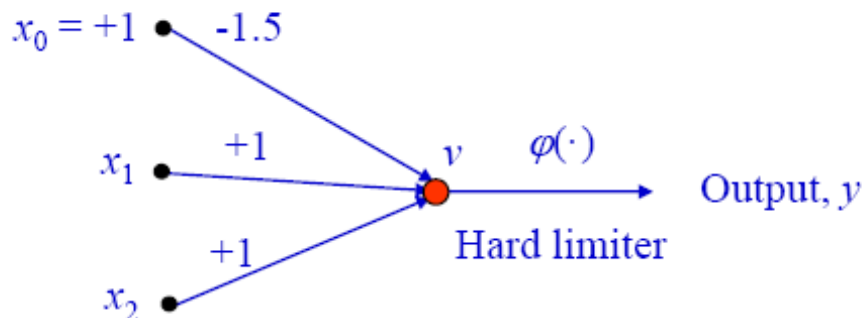
How many lines do we need to separate these two classes?

Let's write down the equations for these two lines:

$$\text{Line 1} \quad x_2 = -x_1 + 1.5 \quad \Longrightarrow \quad x_1 + x_2 - 1.5 = 0$$

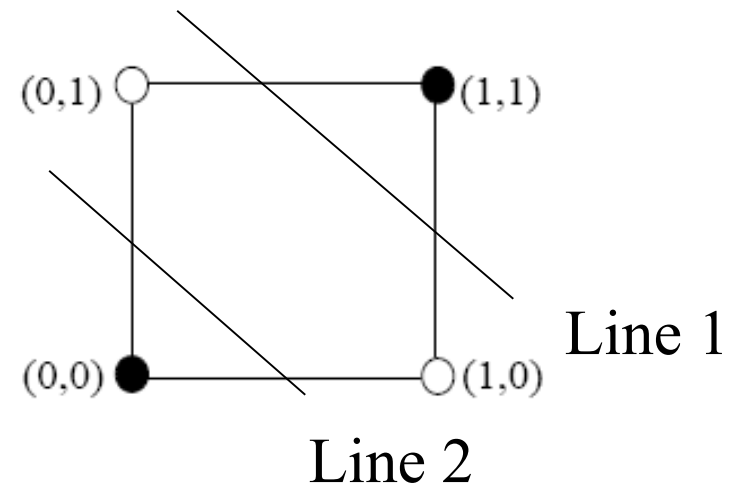
$$\text{Line 2} \quad x_2 = -x_1 + 0.5 \quad \Longrightarrow \quad x_1 + x_2 - 0.5 = 0$$

Can we construct the corresponding perceptrons for these two decision lines separately?

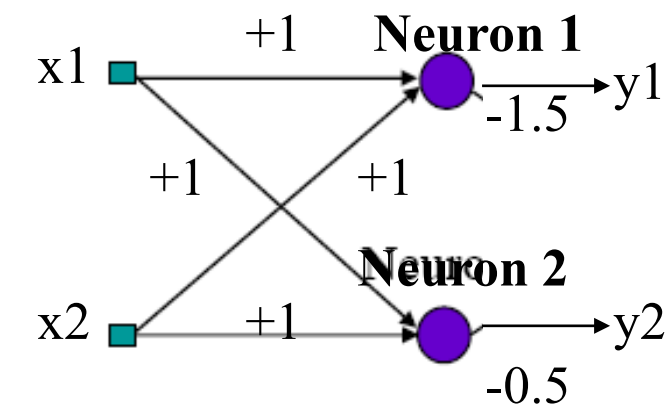


# Single Layer Perceptron with two output neurons

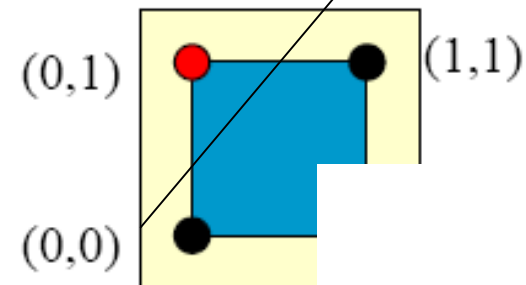
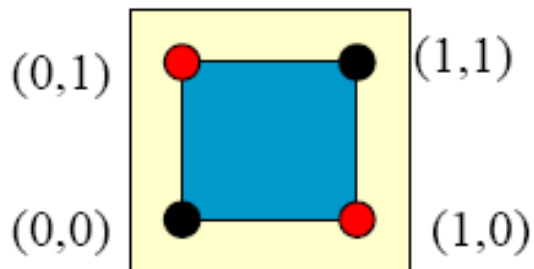
Input		Output
$x_1$	$x_2$	
0	0	0
1	0	1
0	1	1
1	1	0



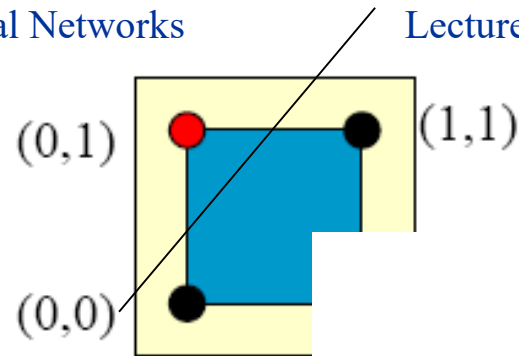
What would happen if we combine the two perceptrons together?



<b>x1</b>	0	0	1	1
<b>x2</b>	0	1	0	1
<b>y1</b>	0	0	0	1
<b>y2</b>	0	1	1	1

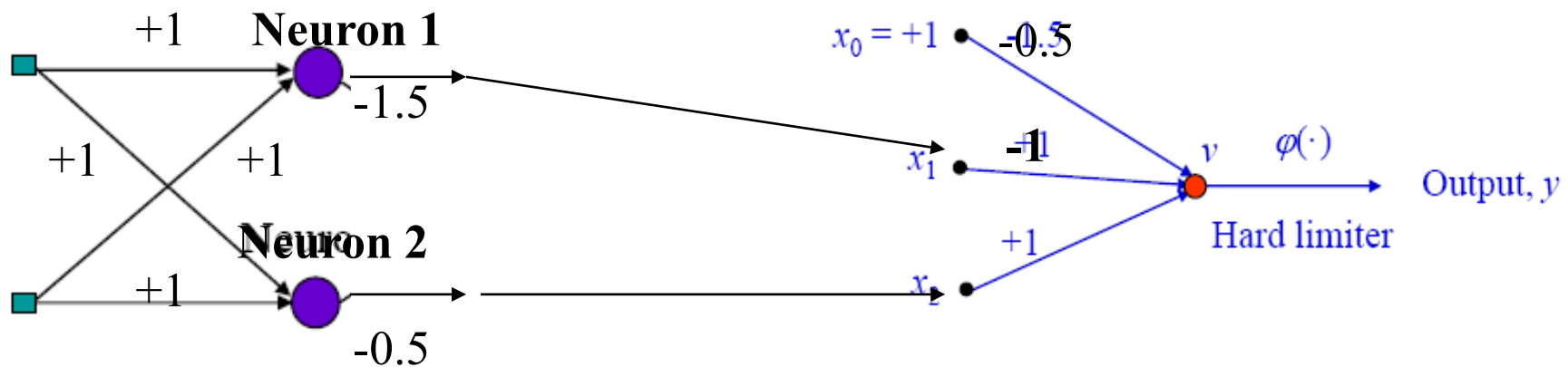


Can you find a line to separate the two classes in the output space (y1,y2) ?



Let's construct the perceptron to separate the two classes.

What is the equation for this line?  $x_2 = x_1 + 0.5 \implies -x_1 + x_2 - 0.5 = 0$

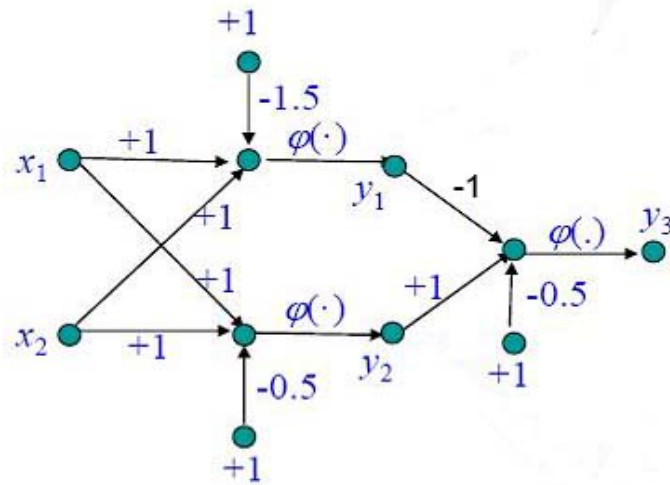


Now, how to combine this perceptron with the previous two neurons?

What are the inputs to this perceptron?

The outputs of the previous two perceptrons serve as the inputs to the output neuron!

## The complete solution to XOR problem



How many layers?

Two layers!

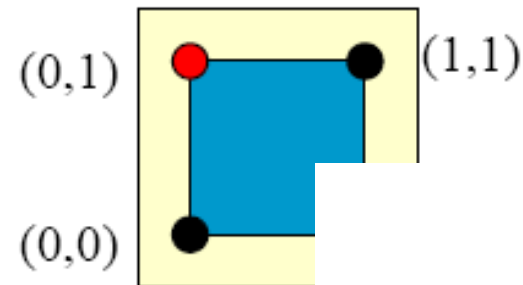
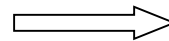
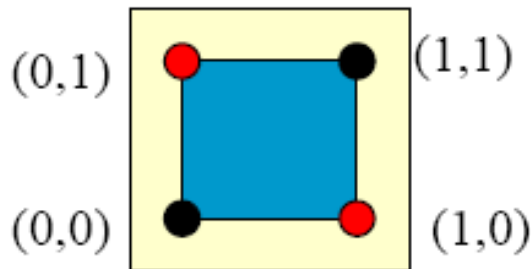
What is the magic?

In the original input space ( $x_1, x_2$ ), is it linearly separable?

No!

In the output space ( $y_1, y_2$ ) of the hidden layer, is it linearly separable?

Yes!



The inputs are transformed into another space ( $y_1, y_2$ ) such that they become linearly separable!

Could Frank Rosenblatt find out this solution and answer Minsky's attack if he had survived the boating accident?

Yes. He could! Unfortunately, we have to wait another 15 years after his tragic death in 1971.



# Multilayer Perceptron (MLP) and Back Propagation Algorithm

David Rumelhart and the PDP (Parallel Distributed Processing) group, 1986



David Rumelhart  
(1942-2011)

He obtained his B.A. in psychology and mathematics in 1963 at the University of South Dakota. He received his Ph. D. in mathematical psychology at Stanford University in 1967. From 1967 to 1987 he served on the faculty of the Department of Psychology at the University of California, San Diego.

The PDP group was led by David Rumelhart and Jay McClelland at UCSD. They became dissatisfied with symbol-processing machines, and embarked on a more ambitious “connectionist” program.

The 1986 PDP book was a big success. The book was read eagerly not only by brain theorists and psychologists but by mathematicians, physicists, engineers and even by people working in Artificial Intelligence.

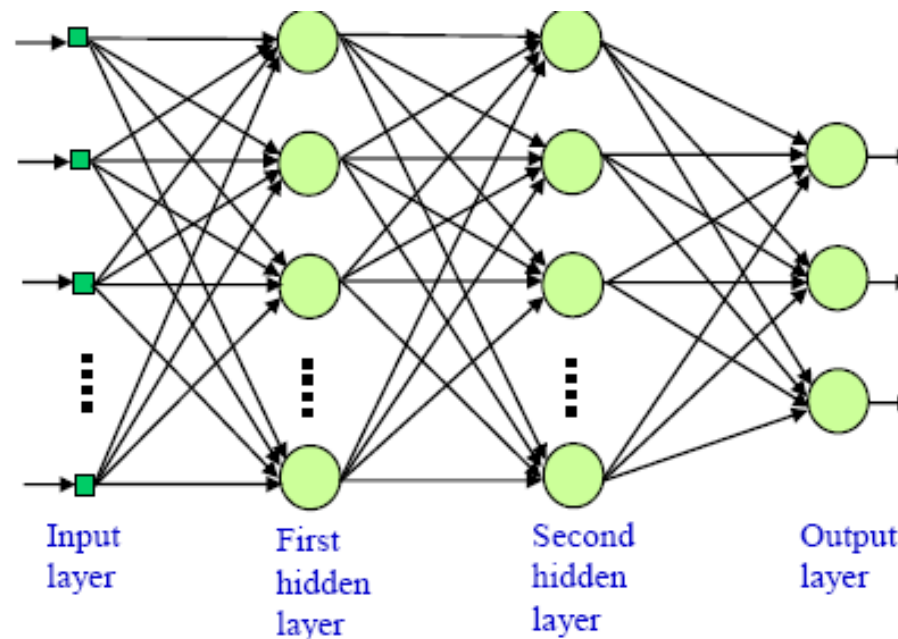
In 1987, Rumelhart moved to Stanford University, serving as Professor there until 1998.

The *Robert J. Glushko and Pamela Samuelson Foundation* created the David E. Rumelhart Prize for Contributions to the Theoretical Foundations of Human Cognition in 2000.

**Francis Crick** was also a member of the PDP group. He joked later, “Almost my only contribution to their efforts was to insist that they stop using the word *neurons* for the units of their networks.”

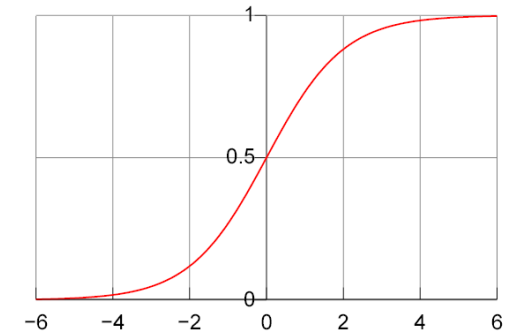
# Multilayer Perceptrons

- Multilayer perceptrons (MLPs)
  - ♦ Generalization of the single-layer perceptron
- Consists of
  - ♦ An input layer
  - ♦ One or more hidden layers of computation nodes
  - ♦ An output layer of computation nodes
- Architectural graph of a multilayer perceptron with two hidden layers:



MLP generally adopts a smooth nonlinear activation function, such as the following logistic function:

$$y_j = \frac{1}{1 + \exp(-v_j)}$$



where  $v_j$  is the induced local field (weighted sum of all synaptic inputs plus the bias) of neuron  $j$ ,  $y_j$  is the output of the neuron.

What would happen if all the neurons are linear neurons? Would it behave differently from single layer perceptrons?

No.

We already showed that MLP can solve the XOR problem by geometrical construction.

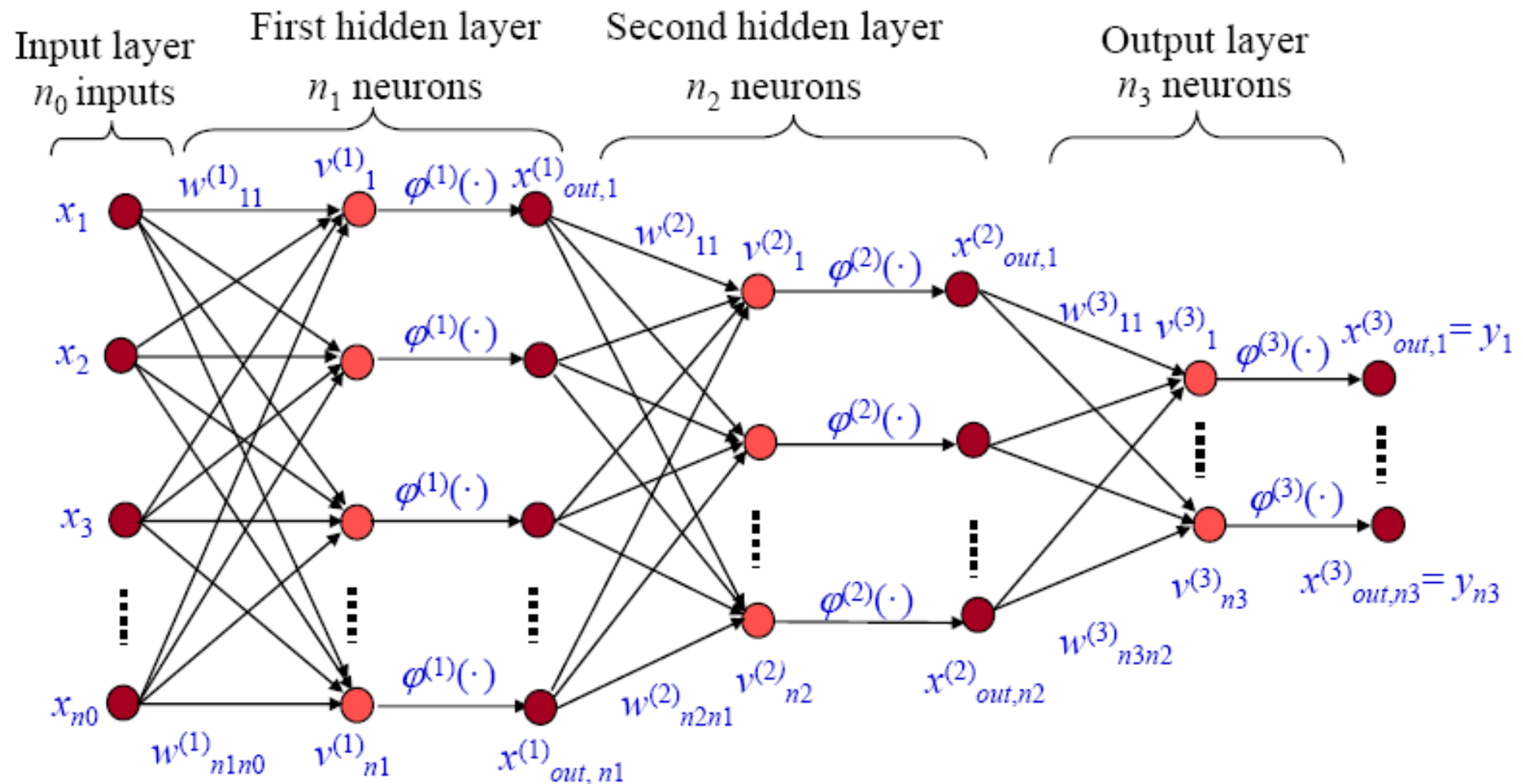
Can MLP solve the XOR problem by learning?

Training Algorithm

- ♦ Back-Propagation (BP) algorithm

## Back-Propagation Algorithm

- Consider a multilayer perceptron neural network having three layers of neurons (one output layer and two hidden layers).



Let's try to figure out the Back-Propagation (BP) algorithm step by step.

The MLP is fed with an input vector  $x(n)$ , and produces an output vector  $y(n)$ .

Let  $d(n)$  denote the desired network output, and the error is then

$$e(n) = d(n) - y(n) = d(n) - x_{out}^{(3)}(n)$$

We will only look at the error at this step!

How to use this error signal to adjust the synaptic weights  $w(n)$ ?

$$w(n+1) = w(n) + \Delta w(n)$$

Did we solve a similar problem in lecture 2?

Yes. The LMS is derived by minimizing the instantaneous errors!

What is the cost function used in LMS?

$$E(n) = \frac{1}{2} e(n)^2$$

Since we may have multiple outputs for MLP, so we define the cost function as

$$\Rightarrow E(n) = \frac{1}{2} \sum_{j=1}^{n_3} e_j(n)^2 = \frac{1}{2} \sum_{j=1}^{n_3} (d_j(n) - x_{out,j}^{(3)}(n))^2$$

$$E(n) = \frac{1}{2} \sum_{j=1}^{n_3} e_j(n)^2 = \frac{1}{2} \sum_{j=1}^{n_3} (d_j(n) - x_{out,j}^{(3)}(n))^2$$

**What is the simplest iterative algorithm to solve the minimization problem?**

Steepest Descent (Gradient Descent):

$$w(n+1) = w(n) - \eta g(n)$$

Similar to LMS algorithm, the learning rule for a network weight is:

$$\Delta w(n) = -\eta g(n)$$

$$g^T(n) = \frac{\partial E(n)}{\partial w(n)}$$

$$\Rightarrow \Delta w_{ji}^{(s)}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}^{(s)}(n)}$$

where  $s = 1, 2, 3$  designates the appropriate network layer,  $\eta > 0$  is the corresponding learning rate parameter.

All we need to do is trying to figure out how to compute the derivatives for all the weights!

**For Output Layer (neuron  $j$  for output layer):**

$$\Delta w_{ji}^{(3)}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}^{(3)}(n)}$$

$$E(n) = \frac{1}{2} \sum_{j=1}^{n_3} e_j(n)^2 = \frac{1}{2} \sum_{j=1}^{n_3} (d_j(n) - x_{out,j}^{(3)}(n))^2$$

Do you know how to calculate  $\frac{\partial E(n)}{\partial x_{out,j}^{(3)}(n)}$  ?

$$\frac{\partial E(n)}{\partial x_{out,j}^{(3)}(n)} = (d_j(n) - x_{out,j}^{(3)}(n)) \bullet (-1) = -e_j(n)$$

How to calculate the outputs of the network,  $x_{out,j}^{(3)}(n)$  ?

$$y_j(n) = x_{out,j}^{(3)}(n) = \varphi^{(3)}(v_j^{(3)}(n))$$

How to calculate  $\frac{\partial x_{out,j}^{(3)}(n)}{\partial v_j^{(3)}(n)}$  ?

$$\frac{\partial x_{out,j}^{(3)}(n)}{\partial v_j^{(3)}(n)} = \varphi^{(3)'}(v_j^{(3)}(n))$$

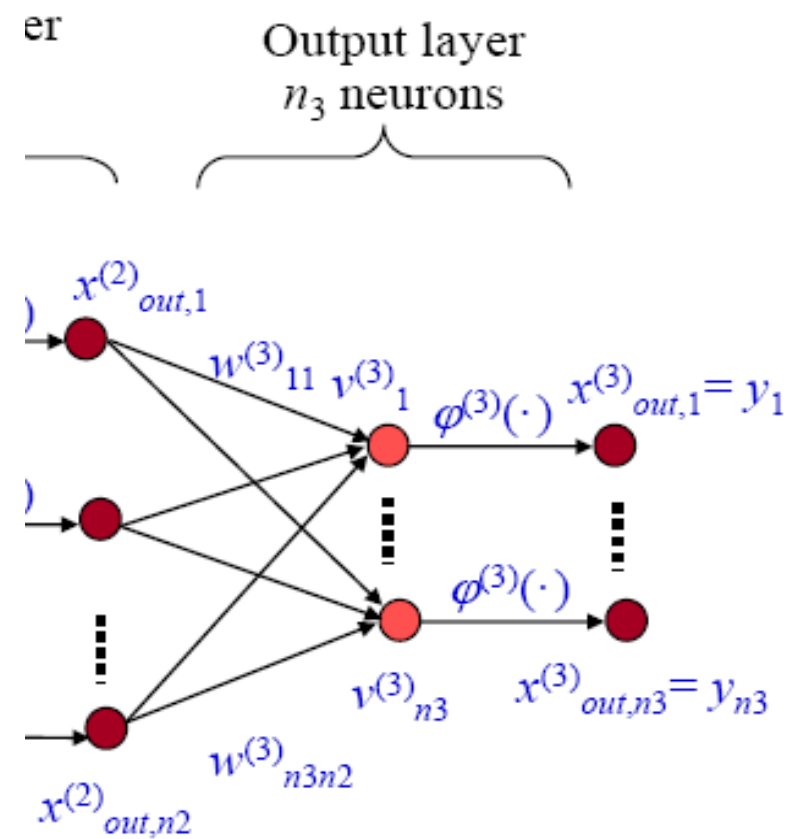
How to compute the induced local fields  $v_j^{(3)}(n)$  ?

$$v_j^{(3)}(n) = \sum_{i=1}^{n_2} w_{ji}^{(3)}(n) x_{out,i}^{(2)}(n)$$

How to calculate  $\frac{\partial v_j^{(3)}(n)}{\partial w_{ji}^{(3)}(n)}$  ?

$$\frac{\partial v_j^{(3)}(n)}{\partial w_{ji}^{(3)}(n)} = x_{out,i}^{(2)}(n)$$

Now the big question is: how to calculate  $\frac{\partial E(n)}{\partial w_{ji}^{(3)}(n)}$  ?



**Chain Rule!**

## For Output Layer (*neuron j for output layer*):

By chain rule,  $g(f(x))' = g'(f(x))f'(x)$ .

$$\frac{\partial E(n)}{\partial w_{ji}^{(3)}(n)} = \frac{\partial E(n)}{\partial x_{out,j}^{(3)}(n)} \frac{\partial x_{out,j}^{(3)}(n)}{\partial v_j^{(3)}(n)} \frac{\partial v_j^{(3)}(n)}{\partial w_{ji}^{(3)}(n)}$$

Now,

$$\frac{\partial E(n)}{\partial x_{out,j}^{(3)}(n)} = -e_j(n)$$

$$\frac{\partial x_{out,j}^{(3)}(n)}{\partial v_j^{(3)}(n)} = \phi^{(3)'}(v_j^{(3)}(n))$$

$$\frac{\partial v_j^{(3)}(n)}{\partial w_{ji}^{(3)}(n)} = x_{out,i}^{(2)}(n)$$

So we have

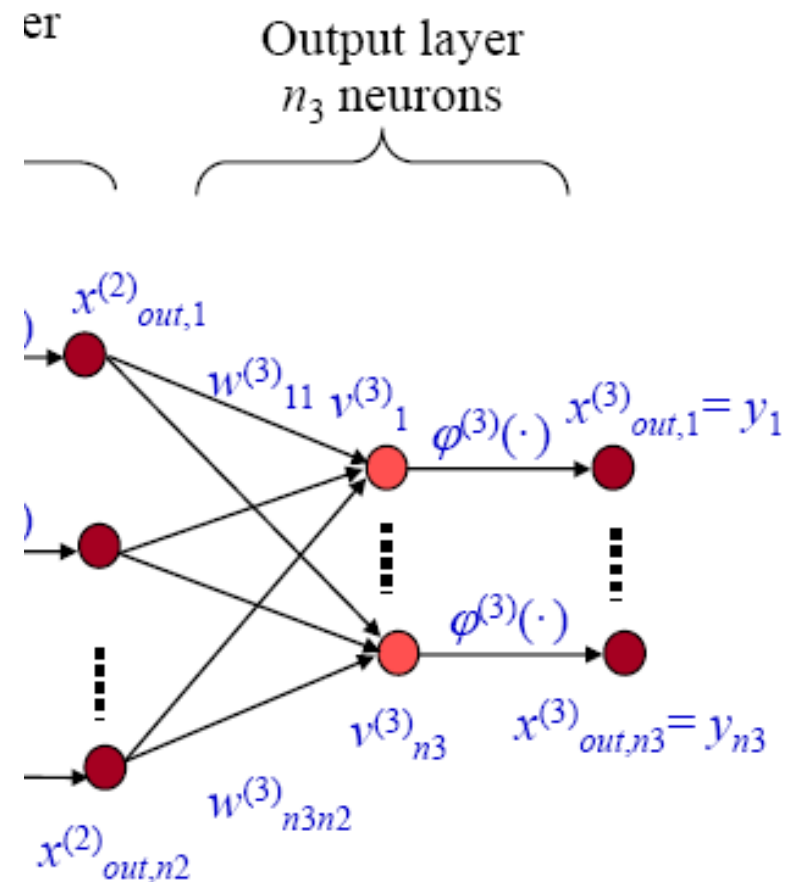
$$\frac{\partial E(n)}{\partial w_{ji}^{(3)}(n)} = -e_j(n) \phi^{(3)'}(v_j^{(3)}(n)) x_{out,i}^{(2)}(n)$$

To simplify the notation, **let's define**

$$\delta_j^{(3)}(n) = e_j(n) \phi^{(3)'}(v_j^{(3)}(n))$$

Then

$$\frac{\partial E(n)}{\partial w_{ji}^{(3)}(n)} = -\delta_j^{(3)}(n) x_{out,i}^{(2)}(n)$$





**For Output Layer (*neuron  $j$  for output layer*):**

$$\frac{\partial E(n)}{\partial w_{ji}^{(3)}(n)} = -\delta_j^{(3)}(n) x_{out,i}^{(2)}(n)$$

The learning rule for the weights in the output layer is

$$\Delta w_{ji}^{(3)}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}^{(3)}(n)}$$

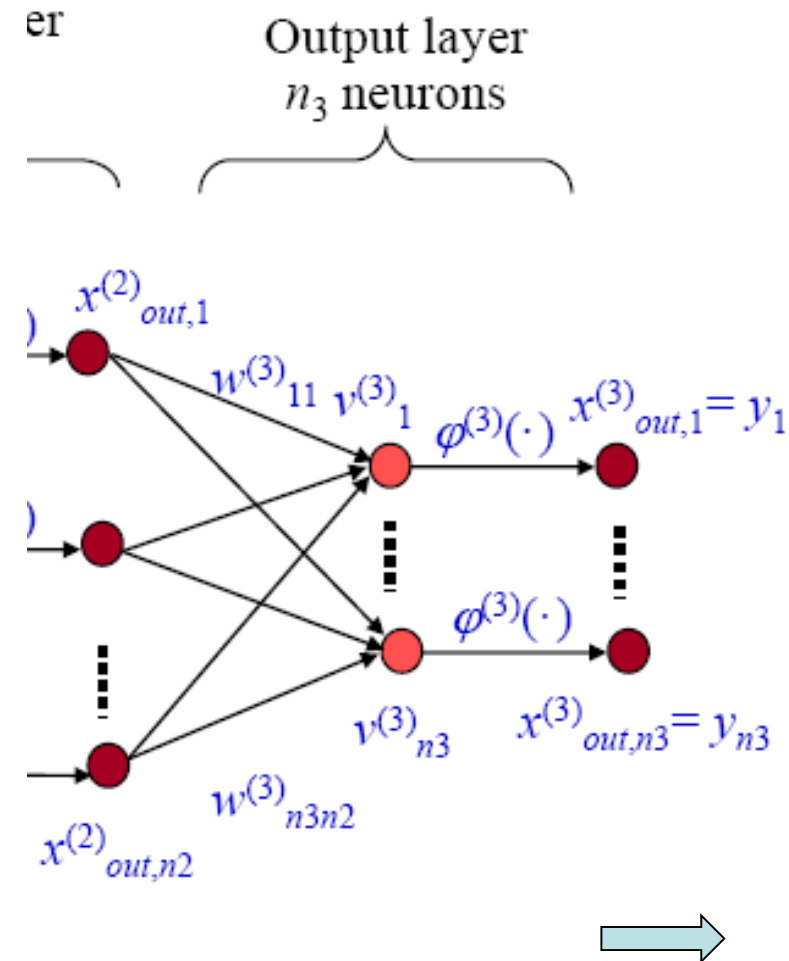
$$\Delta w_{ji}^{(3)}(n) = \eta \delta_j^{(3)}(n) x_{out,i}^{(2)}(n)$$

or

$$w_{ji}^{(3)}(n+1) = w_{ji}^{(3)}(n) + \eta \delta_j^{(3)}(n) x_{out,i}^{(2)}(n)$$

Output Error

Input Signal



Is it similar to the LMS algorithm  $w(n+1) = w(n) + \eta e(n)x(n)$  ?

Yes.

## Derivatives of cost function with respect to the weights in the second hidden layer:

$$E(n) = \frac{1}{2} \sum_{j=1}^{n_3} e_j(n)^2 = \frac{1}{2} \sum_{j=1}^{n_3} (d_j(n) - x_{out,j}^{(3)}(n))^2$$

Let's first figure out how the **error signal** is related to the **synaptic weights** in the **second hidden layer**.

What do the **error signals** depend upon?

The **outputs of the network**.

What do the **network outputs** depend upon?

The **induced local fields** of the output neurons.

What do induced local fields of the output neurons depend upon?

The **outputs of the hidden neurons** and the **synaptic weights** in the output layer.

What do the **outputs of the hidden neurons** depend upon?

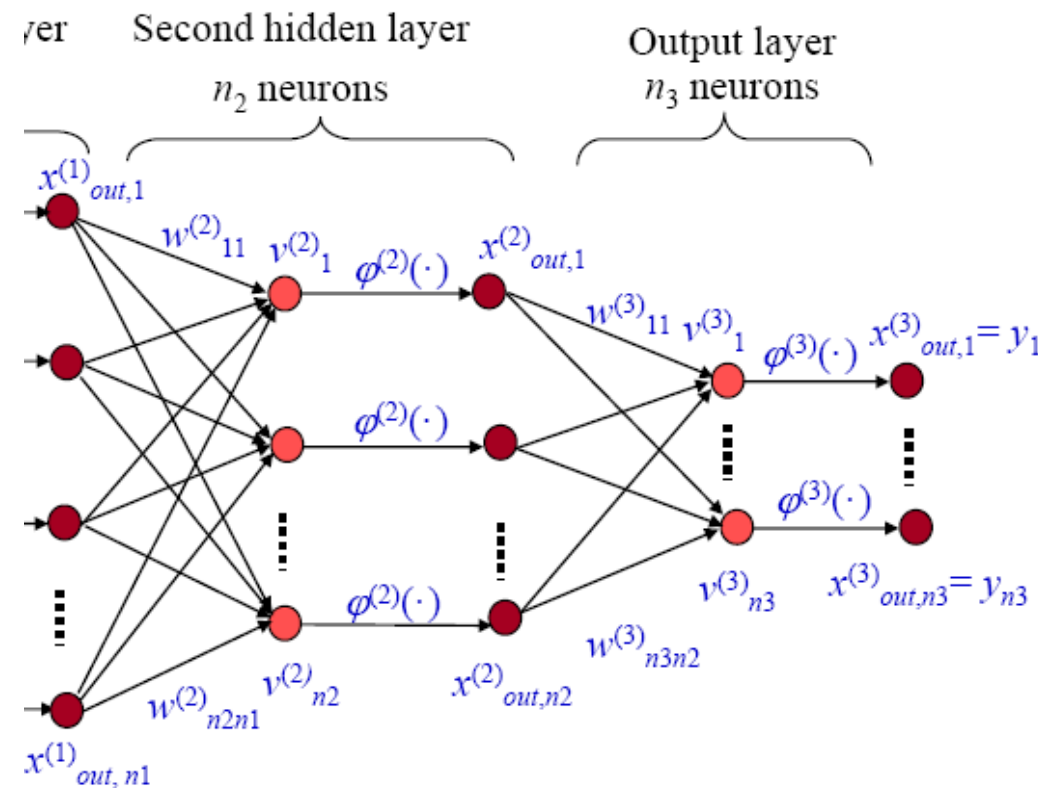
The **induced local fields** of the hidden neurons.

What do the **induced local fields** of the hidden neurons depend upon ?

The **synaptic weights** in the second hidden layer.

There are five levels of dependence between the error signal and the synaptic weights in the second hidden layer. Which rule shall we use to compute the derivatives?

The chain rule, of course!



## Derivatives of cost function with respect to the weights in the second hidden layer:

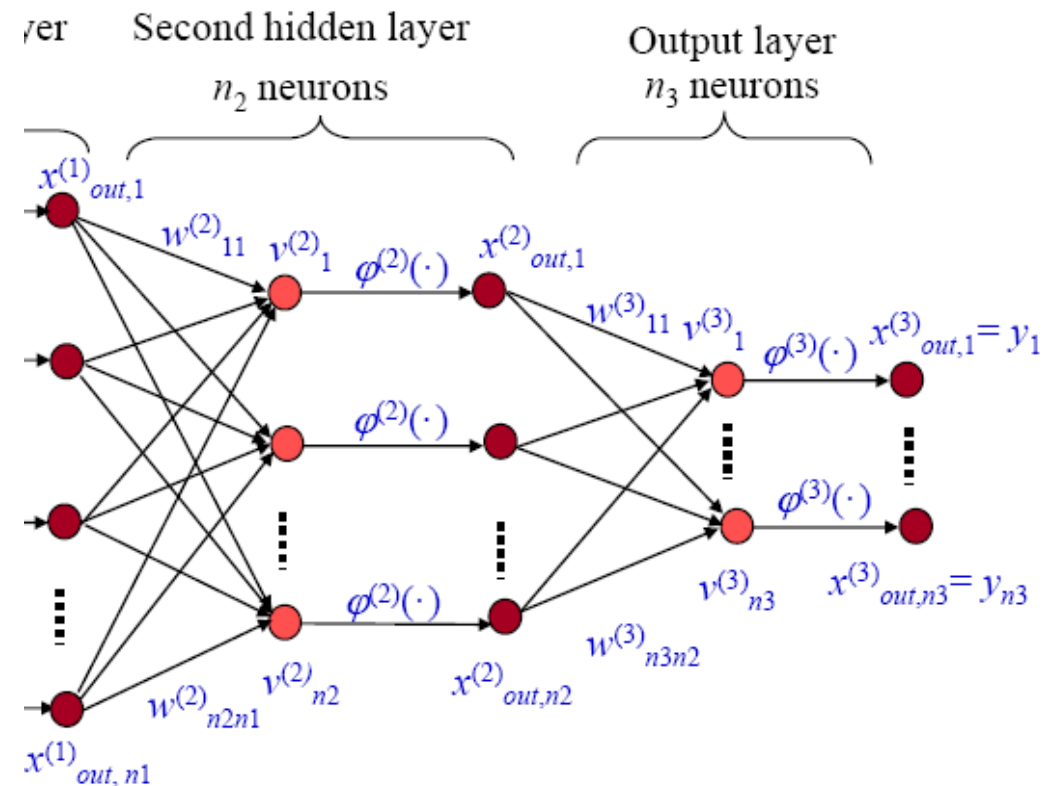
By chain rule, we need to compute the derivatives for every level and then put them together.

The first level is from the outputs of the network to cost function  $E(n)$

$$E(n) = \frac{1}{2} \sum_{j=1}^{n_3} (d_j(n) - x_{out,j}^{(3)}(n))^2$$

So we have ,

$$\frac{\partial E(n)}{\partial x_{out,k}^{(3)}(n)} = -(d_k(n) - x_{out,k}^{(3)}(n)) = -e_k(n)$$



## Derivatives of cost function with respect to the weights in the second hidden layer:

The second level is from the induced local fields of the output neuron to the output of the network.

$$x_{out,k}^{(3)}(n) = \phi^{(3)}(v_k^{(3)}(n))$$

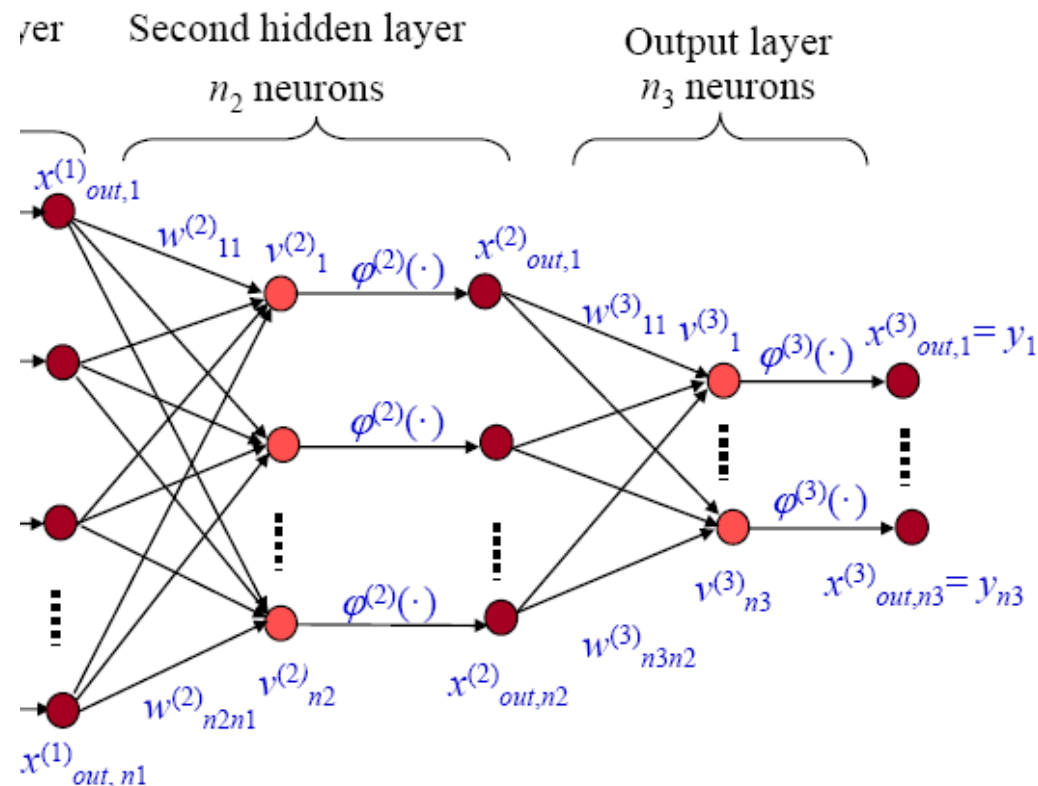
So we have , 
$$\frac{\partial x_{out,k}^{(3)}}{\partial v_k^{(3)}(n)} = \phi^{(3)'}(v_k^{(3)}(n))$$

The third level is from the output of the hidden neurons to the induced local fields of the output neuron.

$$v_k^{(3)}(n) = \sum_{j=1}^{n_2} w_{kj}^{(3)}(n) x_{out,j}^{(2)}(n)$$

Easily we obtain

$$\frac{\partial v_k^{(3)}(n)}{\partial x_{out,j}^{(2)}} = w_{kj}^{(3)}(n)$$



## Derivatives of cost function with respect to the weights in the second hidden layer:

The fourth level is from the induced local fields of the hidden neurons to the outputs of the hidden neurons.

$$x_{out,j}^{(2)}(n) = \phi^{(2)}(v_j^{(2)}(n))$$

Easily we obtain

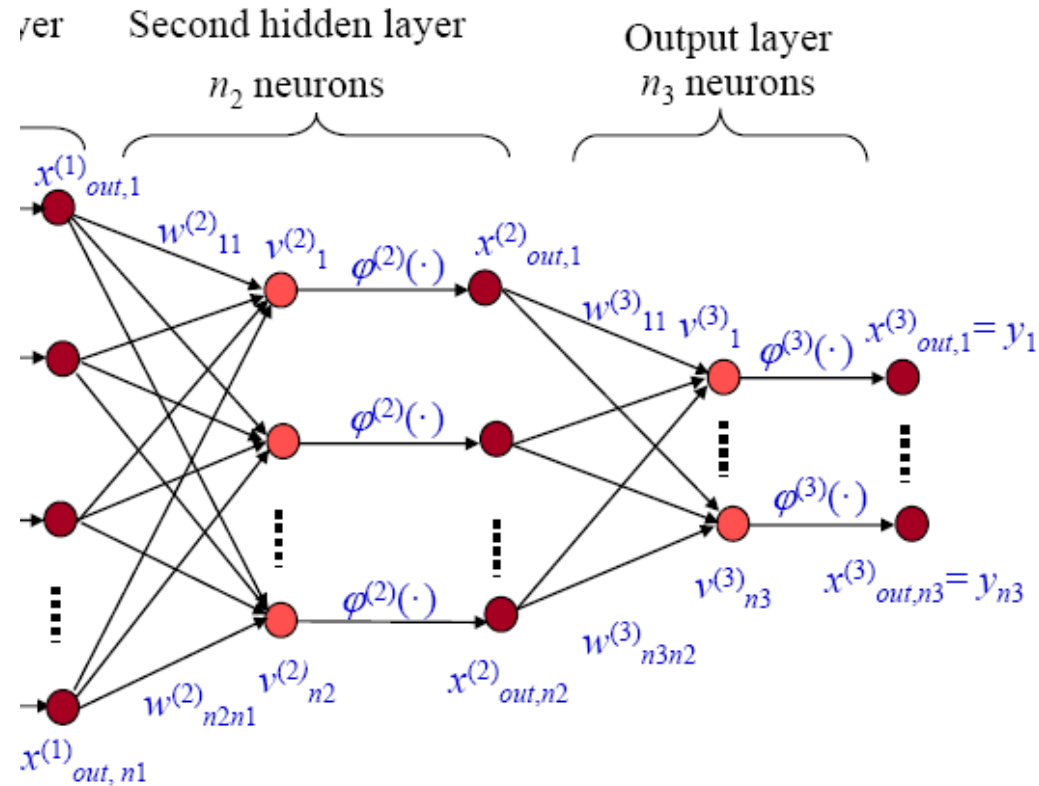
$$\frac{\partial x_{out,j}^{(2)}(n)}{\partial v_j^{(2)}(n)} = \phi^{(2)'}(v_j^{(2)}(n))$$

The fifth level is from the synaptic weights of the hidden layer to the induced local fields of the hidden neurons.

$$v_j^{(2)}(n) = \sum_{k=1}^{n_1} w_{jk}^{(2)}(n) x_{out,k}^{(1)}(n)$$

So we have ,

$$\frac{\partial v_j^{(2)}(n)}{\partial w_{ji}^{(2)}(n)} = x_{out,i}^{(1)}(n)$$



## Derivatives of cost function with respect to the weights in the second hidden layer:

In summary, we have

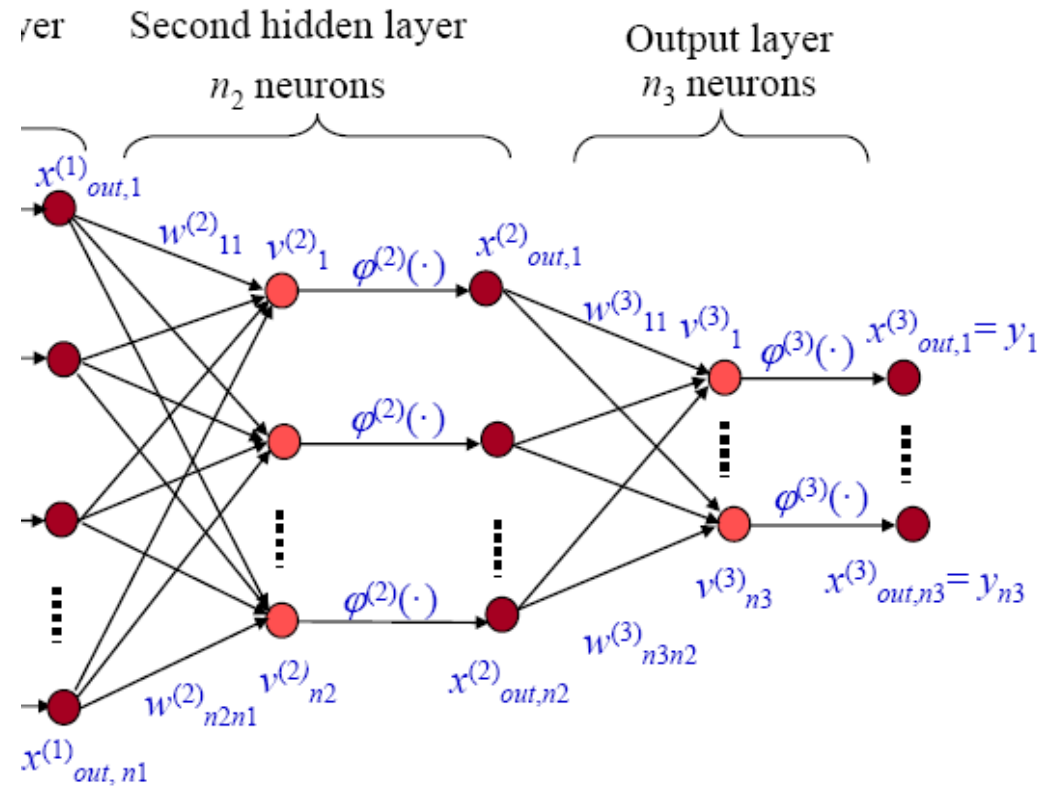
First level: 
$$\frac{\partial E(n)}{\partial x_{out,k}^{(3)}(n)} = -(d_k(n) - x_{out,k}^{(3)}(n)) = -e_k(n)$$

Second level: 
$$\frac{\partial x_{out,k}^{(3)}}{\partial v_k^{(3)}(n)} = \phi^{(3)'}(v_k^{(3)}(n))$$

Third level: 
$$\frac{\partial v_k^{(3)}(n)}{\partial x_{out,j}^{(2)}(n)} = w_{kj}^{(3)}(n)$$

Fourth level: 
$$\frac{\partial x_{out,j}^{(2)}(n)}{\partial v_j^{(2)}(n)} = \phi^{(2)'}(v_j^{(2)}(n))$$

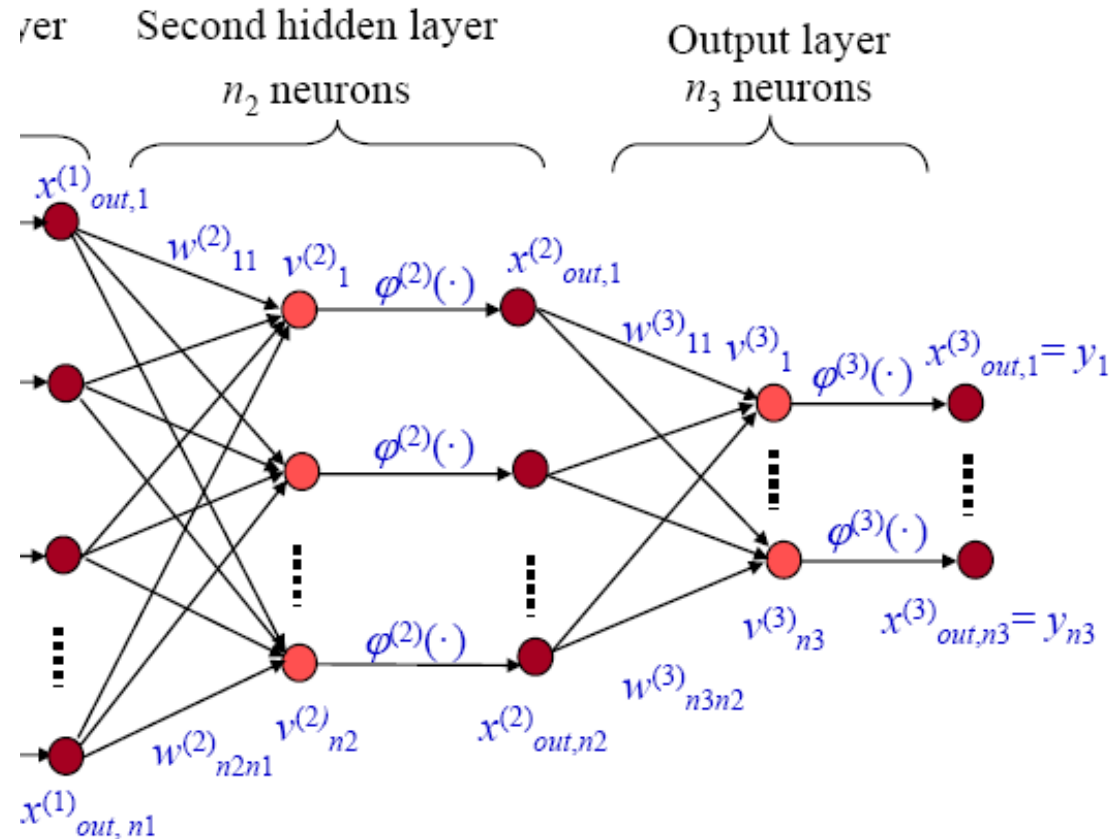
Fifth level: 
$$\frac{\partial v_j^{(2)}(n)}{\partial w_{ji}^{(2)}(n)} = x_{out,i}^{(1)}(n)$$



## Derivatives of cost function with respect to the weights in the second hidden Layer:

$$E(n) = \frac{1}{2} \sum_{j=1}^{n_3} (d_j(n) - x_{out,j}^{(3)}(n))^2$$

Now let's apply the chain rule:



$$\frac{\partial E(n)}{\partial w_{ji}^{(2)}(n)} = \frac{\partial E(n)}{\partial x_{out,k}^{(3)}(n)} \frac{\partial x_{out,k}^{(3)}(n)}{\partial v_k^{(3)}(n)} \frac{\partial v_k^{(3)}(n)}{\partial x_{out,j}^{(2)}(n)} \frac{\partial x_{out,j}^{(2)}(n)}{\partial v_j^{(2)}(n)} \frac{\partial v_j^{(2)}(n)}{\partial w_{ji}^{(2)}(n)}$$

Is this the correct way to apply the chain rule? Did we miss anything?

No, we only consider the k-th output in above calculation. The weights in the hidden layer affect all the outputs of the network.

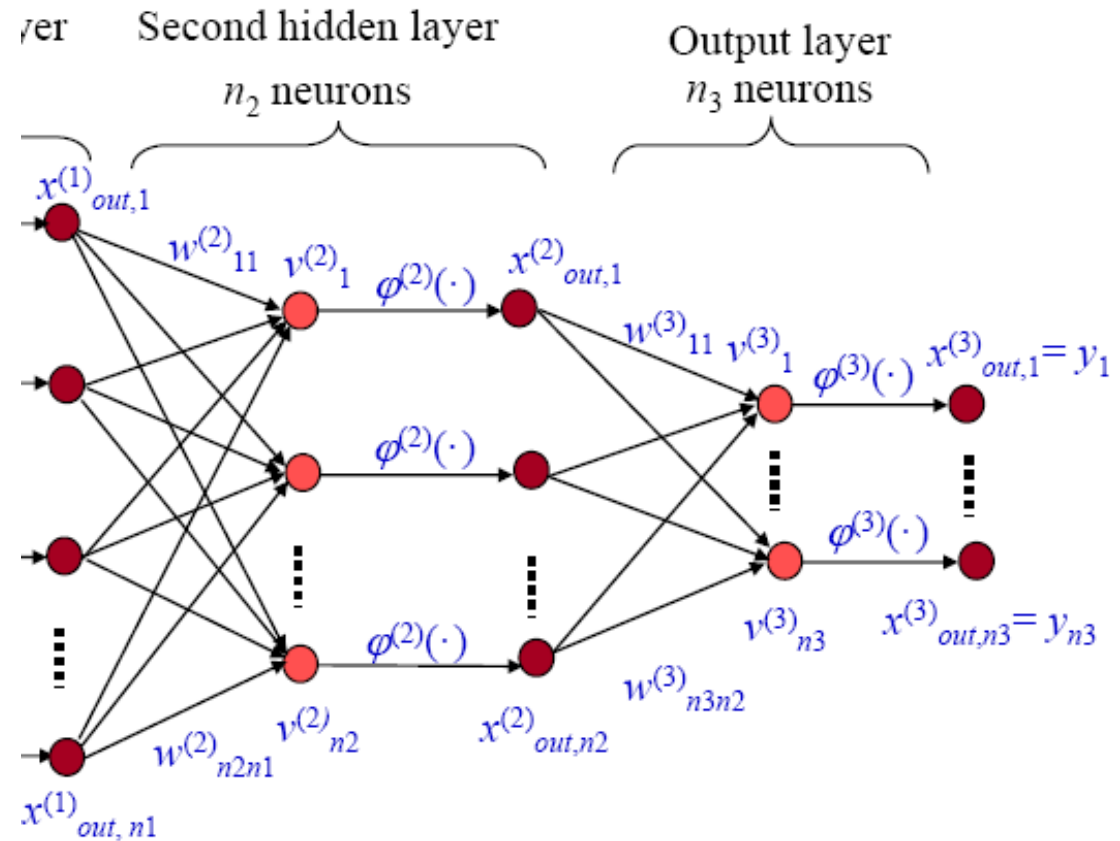
**So we should consider all the outputs of the network instead of only one!**



## Derivatives of cost function with respect to the weights in the second hidden layer:

$$E(n) = \frac{1}{2} \sum_{j=1}^{n_3} (d_j(n) - x_{out,j}^{(3)}(n))^2$$

Now let's apply the chain rule:



$$\frac{\partial E(n)}{\partial w_{ji}^{(2)}(n)} = \sum_{k=1}^{n_3} \left( \frac{\partial E(n)}{\partial x_{out,k}^{(3)}(n)} \frac{\partial x_{out,k}^{(3)}(n)}{\partial v_k^{(3)}(n)} \frac{\partial v_k^{(3)}(n)}{\partial x_{out,j}^{(2)}(n)} \frac{\partial x_{out,j}^{(2)}(n)}{\partial v_j^{(2)}(n)} \frac{\partial v_j^{(2)}(n)}{\partial w_{ji}^{(2)}(n)} \right)$$

Why don't we use the summation for the derivatives with respect to the weights in the [output layer](#)?

The weight in the output layer only affects its connected output neuron!



## Derivatives of cost function with respect to the weights in the second hidden layer:

The cost function: 
$$E(n) = \frac{1}{2} \sum_{j=1}^{n_3} (d_j(n) - x_{out,j}^{(3)}(n))^2$$

The derivatives for all the five levels:

$$\frac{\partial E(n)}{\partial x_{out,k}^{(3)}(n)} = -(d_k(n) - x_{out,k}^{(3)}(n)) = -e_k(n) \qquad \frac{\partial x_{out,k}^{(3)}(n)}{\partial v_k^{(3)}(n)} = \phi^{(3)'}(v_k^{(3)}(n))$$

$$\frac{\partial v_k^{(3)}(n)}{\partial x_{out,j}^{(2)}(n)} = w_{kj}^{(3)}(n) \qquad \frac{\partial x_{out,j}^{(2)}(n)}{\partial v_j^{(2)}(n)} = \phi^{(2)'}(v_j^{(2)}(n)) \qquad \frac{\partial v_j^{(2)}(n)}{\partial w_{ji}^{(2)}(n)} = x_{out,i}^{(1)}(n)$$

The chain rule:

$$\frac{\partial E(n)}{\partial w_{ji}^{(2)}(n)} = \sum_{k=1}^{n_3} \left( \frac{\partial E(n)}{\partial x_{out,k}^{(3)}(n)} \frac{\partial x_{out,k}^{(3)}(n)}{\partial v_k^{(3)}(n)} \frac{\partial v_k^{(3)}(n)}{\partial x_{out,j}^{(2)}(n)} \frac{\partial x_{out,j}^{(2)}(n)}{\partial v_j^{(2)}(n)} \frac{\partial v_j^{(2)}(n)}{\partial w_{ji}^{(2)}(n)} \right)$$

Plug in all the derivatives:

$$\frac{\partial E(n)}{\partial w_{ji}^{(2)}(n)} = \sum_{k=1}^{n_3} (-e_k(n) \phi^{(3)'}(v_k^{(3)}(n)) w_{kj}^{(3)}(n) \phi^{(2)'}(v_j^{(2)}(n)) x_{out,i}^{(1)}(n))$$

## Derivatives of cost function with respect to the weights in the second hidden layer:

$$\frac{\partial E(n)}{\partial w_{ji}^{(2)}(n)} = \sum_{k=1}^{n_3} ( - e_k(n) \phi^{(3)'}(v_k^{(3)}(n)) w_{kj}^{(3)}(n) \phi^{(2)'}(v_j^{(2)}(n)) x_{out,i}^{(1)}(n) )$$

It looks complicated, let's try to simplify it.

Notice that we already define

$$\delta_j^{(3)}(n) = e_j(n) \phi^{(3)'}(v_j^{(3)}(n))$$

So we have

$$\frac{\partial E(n)}{\partial w_{ji}^{(2)}(n)} = - \sum_{k=1}^{n_3} ( \delta_k^{(3)}(n) w_{kj}^{(3)}(n) \phi^{(2)'}(v_j^{(2)}(n)) x_{out,i}^{(1)}(n) )$$

Also notice that the last two terms do not depend upon the index k, so we can take them out from the summation and obtain

$$\frac{\partial E(n)}{\partial w_{ji}^{(2)}(n)} = - \phi^{(2)'}(v_j^{(2)}(n)) x_{out,i}^{(1)}(n) \sum_{k=1}^{n_3} ( \delta_k^{(3)}(n) w_{kj}^{(3)}(n) )$$

## Derivatives of cost function with respect to the weights in the second hidden layer:

$$\frac{\partial E(n)}{\partial w_{ji}^{(2)}(n)} = -\phi^{(2)'}(v_j^{(2)}(n)) x_{out,i}^{(1)}(n) \sum_{k=1}^{n_3} (\delta_k^{(3)}(n) w_{kj}^{(3)}(n))$$

Now let's define **the output error for the hidden neuron**:

$$\delta_j^{(2)}(n) = \left( \sum_{k=1}^{n_3} w_{kj}^{(3)}(n) \delta_k^{(3)}(n) \right) \phi^{(2)'}(v_j^{(2)}(n))$$

So finally we have a very simple formula to compute the derivatives of the cost function with respect to the weights in the second hidden layer:

$$\frac{\partial E(n)}{\partial w_{ji}^{(2)}(n)} = -\delta_j^{(2)}(n) x_{out,i}^{(1)}(n)$$

Thus, by the rule of gradient descent, we have

$$\Delta w_{ji}^{(2)}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}^{(2)}(n)} = \eta \delta_j^{(2)}(n) x_{out,i}^{(1)}(n)$$

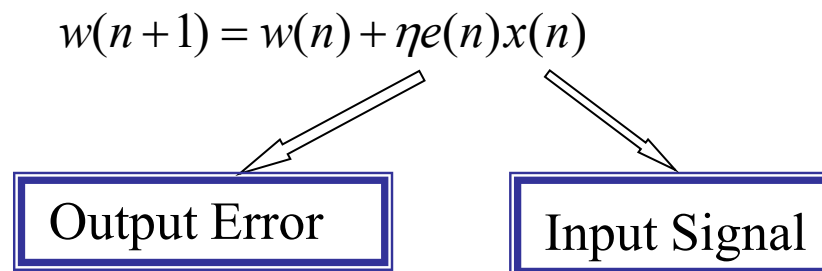
or  $w_{ji}^{(2)}(n+1) = w_{ji}^{(2)}(n) + \eta \delta_j^{(2)}(n) x_{out,i}^{(1)}(n)$

Let's compare it to the updating rule for the output layer:

$$w_{ji}^{(3)}(n+1) = w_{ji}^{(3)}(n) + \eta \delta_j^{(3)}(n) x_{out,i}^{(2)}(n)$$

**Do they have the same form?**

They have the same form as that for the LMS algorithm:

$$w(n+1) = w(n) + \eta e(n)x(n)$$


The diagram illustrates the LMS algorithm equation  $w(n+1) = w(n) + \eta e(n)x(n)$ . Two arrows point from the terms  $e(n)$  and  $x(n)$  in the equation to two boxes below. The left box is labeled "Output Error" and the right box is labeled "Input Signal".

Difference lies in how we compute the output error.

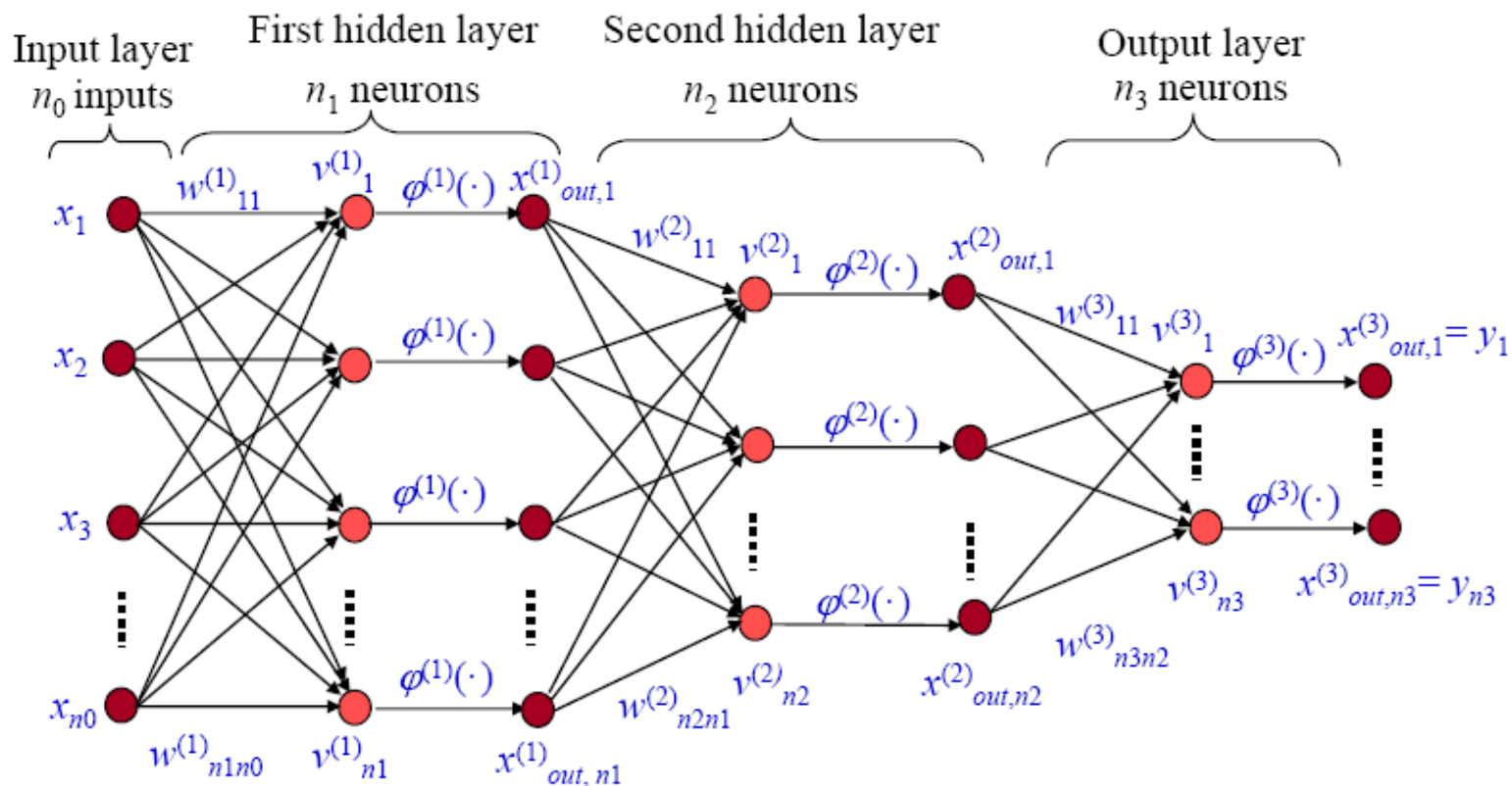
How should we calculate the output error at the output layer,  $\delta_j^{(3)}$  ?

$$\delta_j^{(3)}(n) = e_j(n) \phi^{(3)'}(v_j^{(3)}(n))$$

Network Error

Gradient of the activation function

For the output layer, the output error is proportional to the network error!



How should we calculate the output error at the hidden layer,  $\delta_j^{(2)}$  ?

$$\delta_j^{(2)}(n) = \left( \sum_{k=1}^{n_3} w_{kj}^{(3)}(n) \delta_k^{(3)}(n) \right) \phi^{(2)'}(v_j^{(2)}(n))$$

Output Error

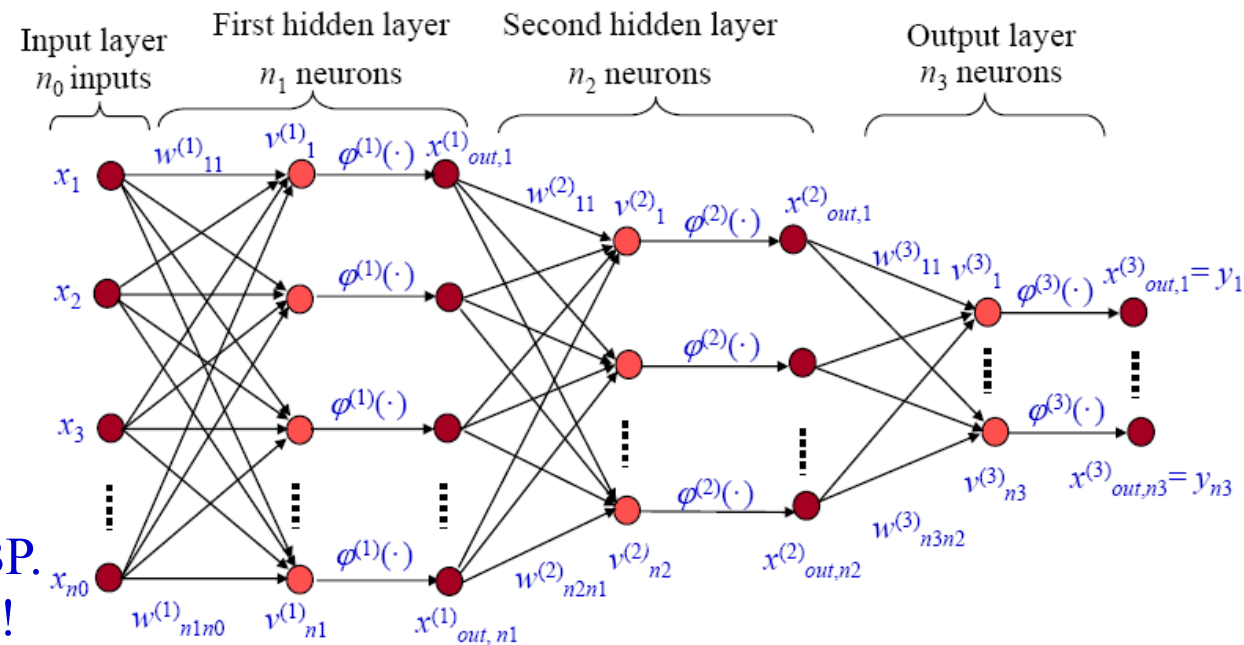
Gradient of the activation function

**For the hidden neurons, the output error is the linear combination of the errors in the higher layer!**

$$\sum_{k=1}^{n_3} w_{kj}^{(3)}(n) \delta_k^{(3)}(n)$$

It can be viewed as the output of the neuron of another network where all the connections are “backwards” now!

This is the most beautiful property for BP.  
That is why it is called backpropagation!



Summary,

$$w_{ji}^{(s)}(n+1) = w_{ji}^{(s)}(n) + \eta \delta_j^{(s)}(n) x_{out,i}^{(s-1)}(n)$$

where

$$\delta_j^{(s)}(n) = (d(n) - x_{out,j}^{(s)}(n)) \varphi^{(s)'}(v_j^{(s)}(n)) \quad \text{for output layer}$$

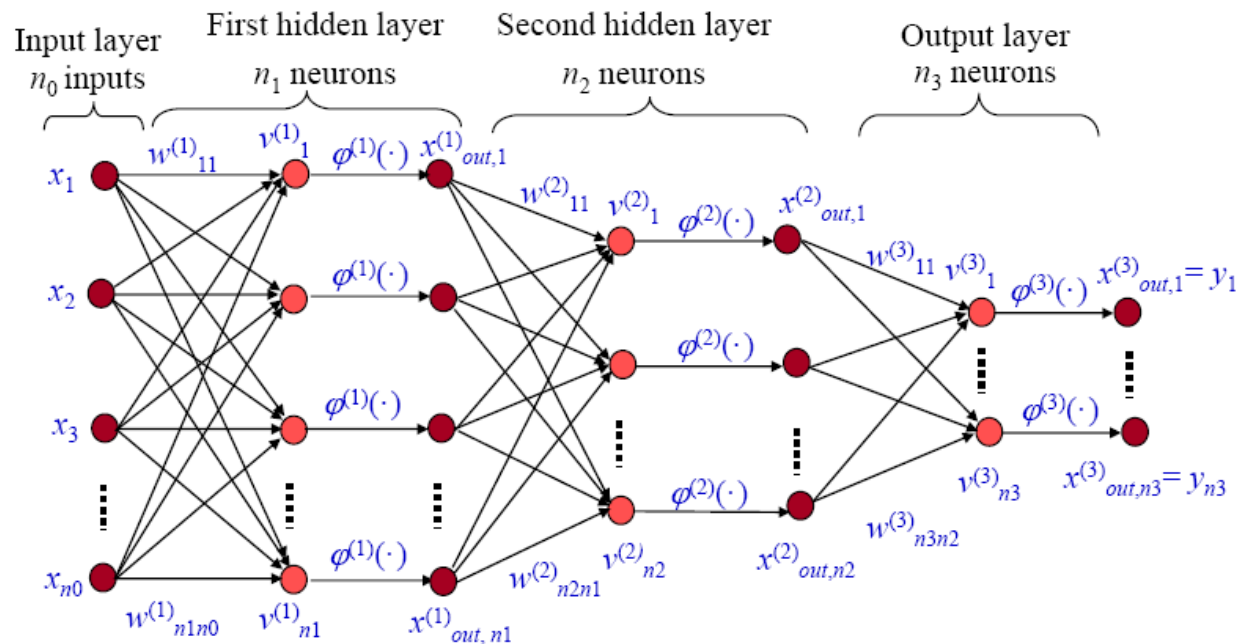
or

$$\delta_j^{(s)}(n) = \left( \sum_{k=1}^{n_{s+1}} \delta_k^{(s+1)}(n) w_{kj}^{(s+1)}(n) \right) \varphi^{(s)'}(v_j^{(s)}(n)) \quad \text{for hidden layer}$$

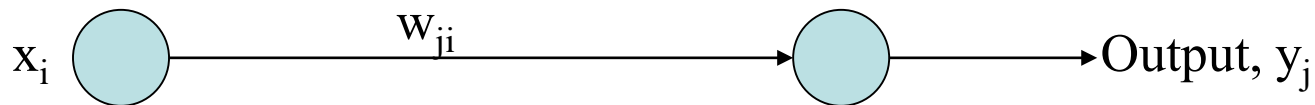
Back-propagation algorithm - 2 passes of computation:

1. Forward pass: Computation of function signals for each neuron.
2. Backward pass: Starts at the output layer, *backwardly compute*  $\delta$  for each neuron from output layer towards the first hidden layer. At each layer, the synaptic weights are changed accordingly to the above delta rule.

# The learning of the Multi-layer perceptron



$$w_{ji}^{(s)}(n+1) = w_{ji}^{(s)}(n) + \eta \delta_j^{(s)}(n) x_{out,i}^{(s-1)}(n)$$



The adjustment of the synaptic weight only depends upon the information of the input neuron and the output neuron, and nothing else.

The output of the neuron depends upon all the connected neurons!

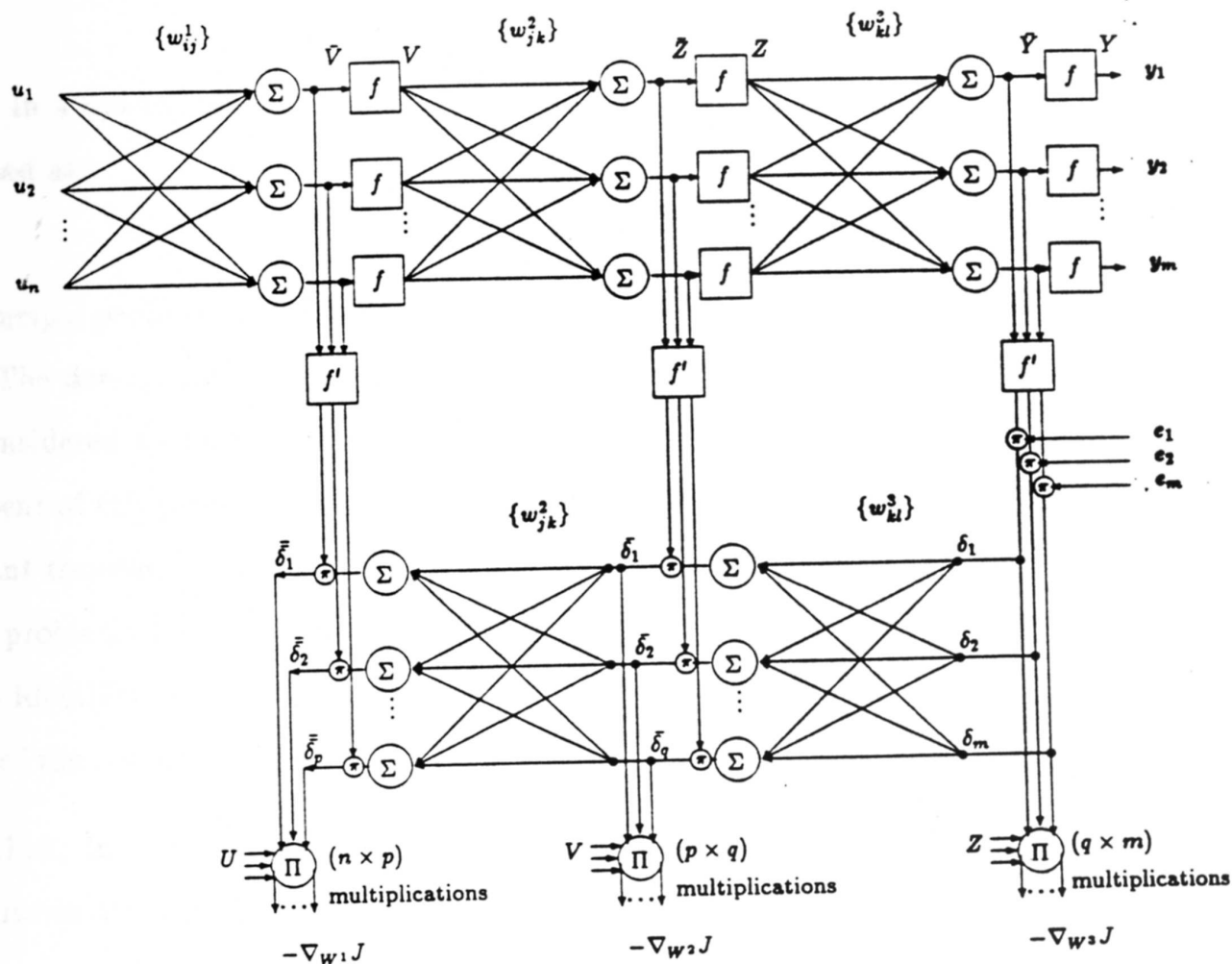
The adjustment of the synaptic weight is proportional to both the input signal and the output error.



*Break*

- Visions of The Future 3

## Signal-flow graphic representation of BP



Is BP a special case of steepest descent method?

Sure!

Why is it called back-propagation?

The errors are propagated back from the output layer to the input layer!

## Back-Propagation Algorithm - Rate of learning

Back-propagation algorithm is based upon the method of steepest descent.

What is the requirement for the learning rate for steepest descent method?

⇒ Small learning-rate parameter  $\eta$  is desirable to avoid “jumping”.

Small learning rate parameter corresponds to “SLOW Learning”!

To increase the learning speed,

⇒ Modification of delta rule to give the generalized delta rule

$$w_{ji}^{(s)}(k+1) = w_{ji}^{(s)}(k) + \underbrace{\eta^{(s)} \delta_j^{(s)} x_{out,i}^{(s-1)}}_{\Delta w_{ji}^{(s)}}$$

$$\Rightarrow \Delta w_{ji}^{(s)}(k) = \alpha \Delta w_{ji}^{(s)}(k-1) + \eta^{(s)} \delta_j^{(s)}(k) x_{out,i}^{(s-1)}(k)$$

where  $\alpha$  is usually a positive number called the *momentum constant* or *forgetting factor*;  $\alpha \Delta w_{ji}^{(s)}(k-1)$  is called the *momentum term*.

The adjustments depend upon not only the derivatives at present, but also the history!

$$\Delta w_{ji}^{(s)}(k) = \alpha \Delta w_{ji}^{(s)}(k-1) + \eta^{(s)} \delta_j^{(s)}(k) x_{out,i}^{(s-1)}(k)$$

Let's write down the solutions step by step,

At the initial step,  $k=0$

$$\Delta w_{ji}^{(s)}(0) = \eta^{(s)} \delta_j^{(s)}(0) x_{out,i}^{(s-1)}(0)$$

At the first step,  $k=1$

$$\Delta w_{ji}^{(s)}(1) = \alpha \Delta w_{ji}^{(s)}(0) + \eta^{(s)} \delta_j^{(s)}(1) x_{out,i}^{(s-1)}(1)$$

$$\Delta w_{ji}^{(s)}(1) = \alpha \eta^{(s)} \delta_j^{(s)}(0) x_{out,i}^{(s-1)}(0) + \eta^{(s)} \delta_j^{(s)}(1) x_{out,i}^{(s-1)}(1)$$

At the second step,  $k=2$

$$\Delta w_{ji}^{(s)}(2) = \alpha \Delta w_{ji}^{(s)}(1) + \eta^{(s)} \delta_j^{(s)}(2) x_{out,i}^{(s-1)}(2)$$

$$\Delta w_{ji}^{(s)}(2) = \alpha^2 \eta^{(s)} \delta_j^{(s)}(0) x_{out,i}^{(s-1)}(0) + \alpha \eta^{(s)} \delta_j^{(s)}(1) x_{out,i}^{(s-1)}(1) + \eta^{(s)} \delta_j^{(s)}(2) x_{out,i}^{(s-1)}(2)$$

In general, at step  $k$ ,

$$\Delta w_{ji}^{(s)}(k) = \alpha \Delta w_{ji}^{(s)}(k-1) + \eta^{(s)} \delta_j^{(s)}(k) x_{out,i}^{(s-1)}(k)$$

$$\Delta w_{ji}^{(s)}(k) = \alpha^k \eta^{(s)} \delta_j^{(s)}(0) x_{out,i}^{(s-1)}(0) + \alpha^{(k-1)} \eta^{(s)} \delta_j^{(s)}(1) x_{out,i}^{(s-1)}(1) + \alpha^{(k-2)} \eta^{(s)} \delta_j^{(s)}(2) x_{out,i}^{(s-1)}(2) + \dots + \eta^{(s)} \delta_j^{(s)}(k) x_{out,i}^{(s-1)}(k)$$

$$\Delta w_{ji}^{(s)}(k) = \sum_{t=0}^k \alpha^{k-t} \eta^{(s)} \delta_j^{(s)}(t) x_{out,i}^{(s-1)}(t) = \eta^{(s)} \sum_{t=0}^k \alpha^{k-t} \delta_j^{(s)}(t) x_{out,i}^{(s-1)}(t)$$

$$\Delta w_{ji}^{(s)}(k) = \alpha \Delta w_{ji}^{(s)}(k-1) + \eta^{(s)} \delta_j^{(s)}(k) x_{out,i}^{(s-1)}(k)$$

$$\Delta w_{ji}^{(s)}(k) = \eta^{(s)} \sum_{t=0}^k \alpha^{k-t} \delta_j^{(s)}(t) x_{out,i}^{(s-1)}(t)$$

From previous analysis,  $\frac{\partial E}{\partial w_{ji}^{(s)}} = -x_{out,i}^{(s-1)} \delta_j^{(s)}$ , so

$$\Delta w_{ji}^{(s)}(k) = -\eta^{(s)} \sum_{t=0}^k \alpha^{k-t} \frac{\partial E(t)}{\partial w_{ji}^{(s)}(t)}$$

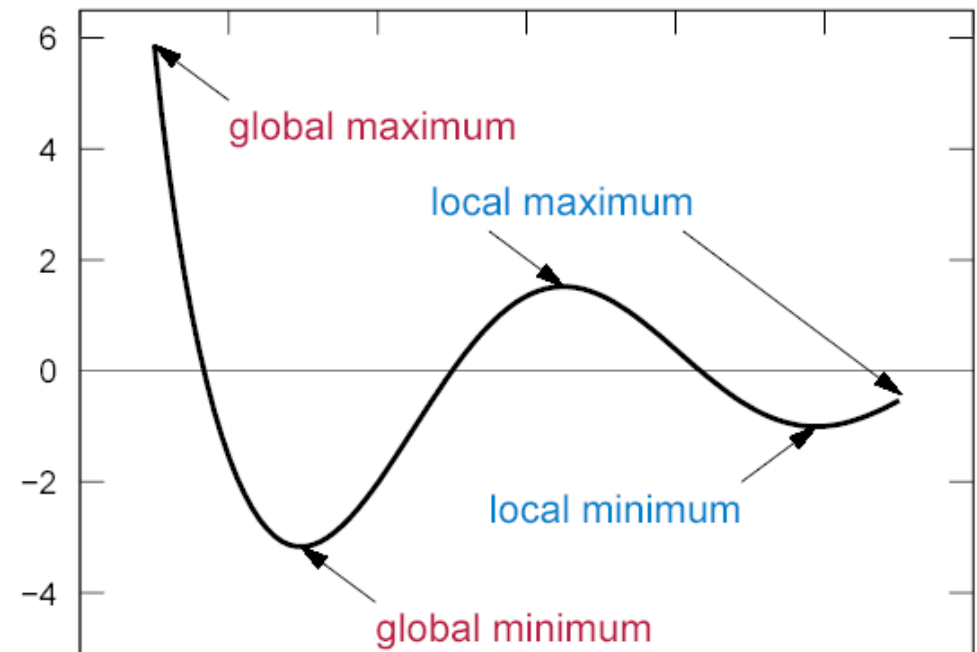
The adjustments depend upon the weighted sum of present and the past derivatives!

Note:

1. For stability,  $0 \leq |\alpha| < 1$

2. Normally, the momentum constant is positive to assure good performance. Why?

$$\Delta w_{ji}^{(s)}(k) = -\eta^{(s)} \sum_{t=0}^k \alpha^{k-t} \frac{\partial E(t)}{\partial w_{ji}^{(s)}(t)}$$



When  $\frac{\partial E(t)}{\partial w_{ji}^{(s)}(t)}$  has the *same* algebraic sign on consecutive iterations, then  $\Delta w_{ji}^{(s)}(k)$  grows in magnitude and  $w_{ji}^{(s)}(k)$  is adjusted by a *large* amount. The inclusion of momentum in the back-propagation algorithm tends to accelerate descent in steady downhill directions.

When  $\frac{\partial E(t)}{\partial w_{ji}^{(s)}(t)}$  has the *opposite* signs on consecutive iterations, the  $\Delta w_{ji}^{(s)}(k)$  shrinks in magnitude, the  $w_{ji}^{(s)}(k)$  is adjusted by a *small* amount. The inclusion of momentum in BP algorithm has a stabilizing effect in directions along which  $\frac{\partial E(t)}{\partial w_{ji}^{(s)}(t)}$  oscillates in sign.

## Back-Propagation Algorithm - Stopping Criteria

**Epoch:** A complete presentation of the entire training set during the learning process.

For single layer perceptron, does the LMS converge?

Similarly, BP would not converge.

Learning process maintains on an epoch-by-epoch basis until certain stopping criterion is met.

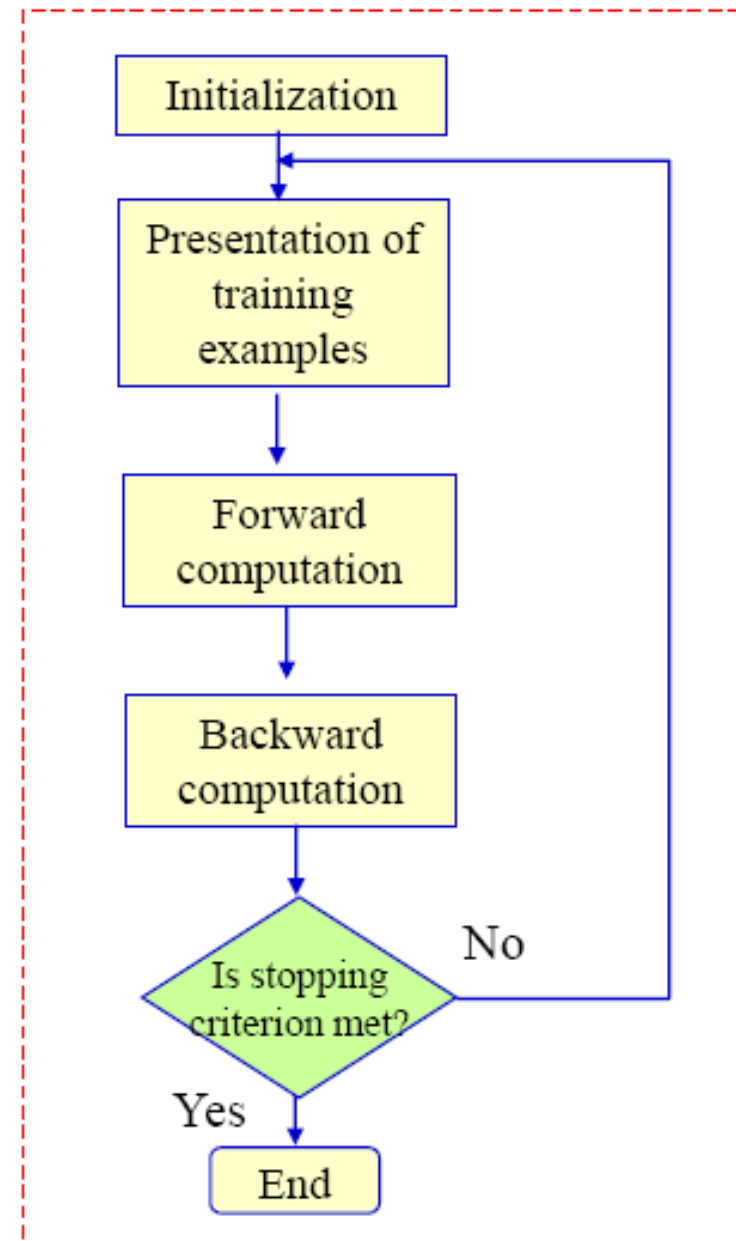
When to stop training? Any suggestions?

Possible criteria:

- ♦ Mean squared error over the entire training set is less than some threshold value.
- ♦ The total number of epochs reaches a threshold.
- ♦ Absolute rate of change in mean squared error per epoch is sufficiently small.
- ♦ Synaptic weights and bias level stabilized.

# Back-Propagation Algorithm - Summary

1. Initialization:
  - ♦ weights, biases
2. Presentations of training examples
  - ♦ Present an epoch of training examples
  - ♦ For each example:
    - perform forward and backward computations
3. Forward computation
  - ♦ Compute error signal
4. Backward computation
  - ♦ Compute  $\delta$ s and adjust weights based on generalized delta rule
5. Iteration
  - ♦ If stopping criterion is not met, go through step 2, 3, 4.





## Approximations of Functions

### Question: Can Multi-layer Perceptrons approximate any functions?

This question was answered shortly after 1986 by a number of people including Cybenko, Hecht-Nielsen, Funahashi, Hornik and White.

### Universal Approximation Theorem:

- ♦ Let  $\varphi(\cdot)$  be a non-constant, bounded, and monotone-increasing continuous function. Let  $I_{m_0}$  denote the  $m_0$ -dimensional unit hypercube  $[0, 1]^{m_0}$ . Then, given any continuous function  $f$  on  $I_{m_0}$  and  $\varepsilon > 0$ , there exist an integer  $m_1$  and sets of real constant  $\alpha_i, b_i$  and  $w_{ij}$ , where  $i = 1, \dots, m_1$  and  $j = 1, \dots, m_0$  such that we may define,

$$F(x_1, \dots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi \left( \sum_{j=1}^{m_0} w_{ij} x_j + b_i \right)$$

as an approximate realization of the function  $f(\cdot)$ ; that is,

$$\left| F(x_1, \dots, x_{m_0}) - f(x_1, \dots, x_{m_0}) \right| < \varepsilon$$

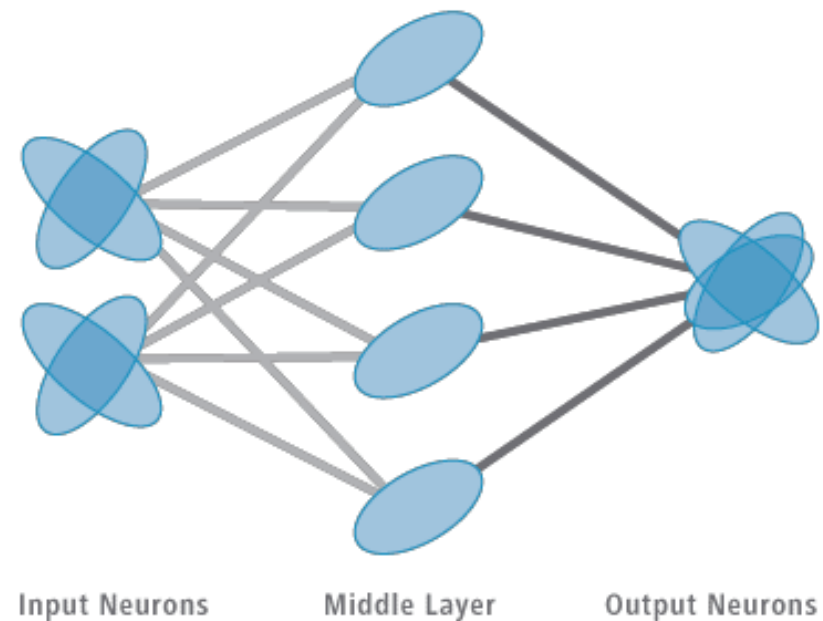
for all  $x_1, x_2, \dots, x_{m_0}$  that lie in the input space.

$$F(x_1, \dots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi \left( \sum_{j=1}^{m_0} w_{ij} x_j + b_i \right)$$

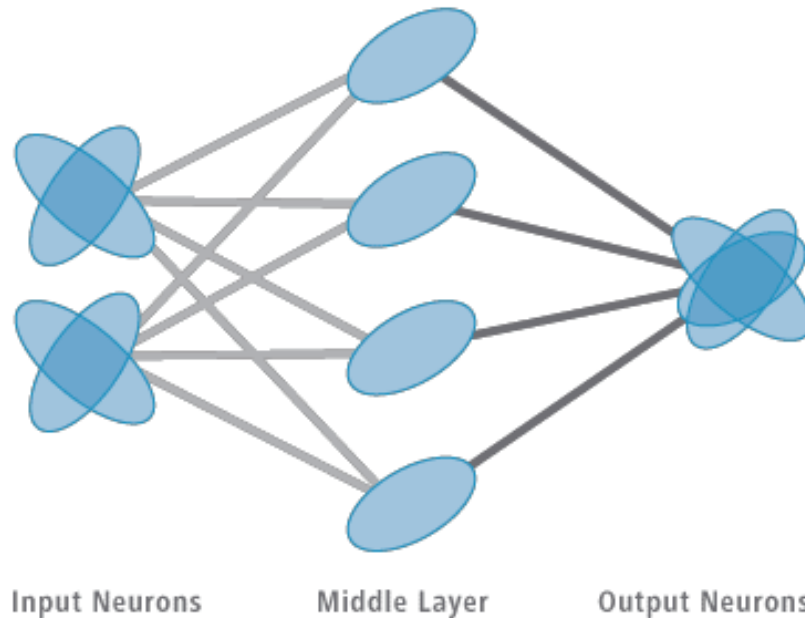
The theorem is directly applicable to multi-layer perceptrons. The logistic function  $1/[1 + \exp(-v)]$  is indeed a non-constant, bounded, and monotone-increasing function, i.e., it satisfies the conditions imposed on the function  $\varphi(\cdot)$ .

How to realize this function using a multilayer perceptron?

- ♦ How many input neurons?  $m_0$
- ♦ How many hidden layers? One
- ♦ How many hidden neurons?  $m_1$
- ♦ How many output neurons? One
- ♦ What are the activation functions?  $\varphi(\cdot)$



$$F(x_1, \dots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi \left( \sum_{j=1}^{m_0} w_{ij} x_j + b_i \right)$$



The theorem merely states that a **single hidden layer** is sufficient for a multilayer perceptron to approximate any bounded continuous function.

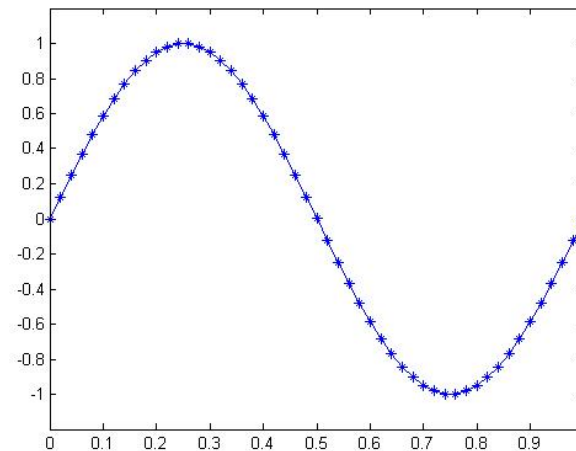
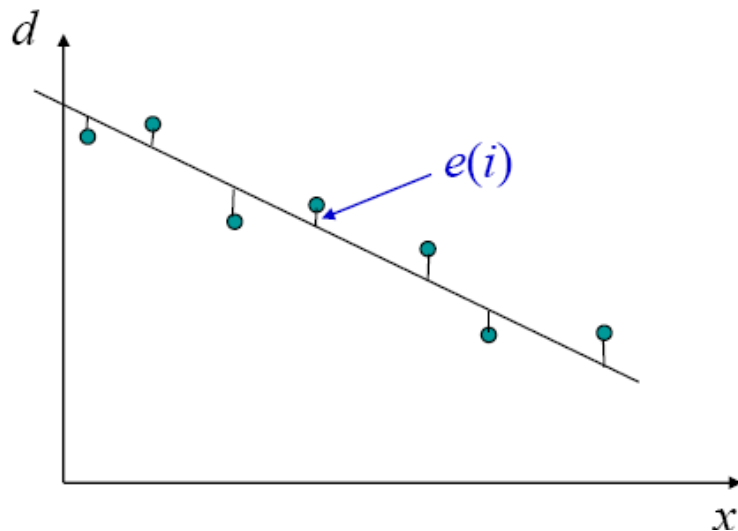
**Does the theorem tell you how to find out the optimal weights?**

**No. This is just an existence result.**

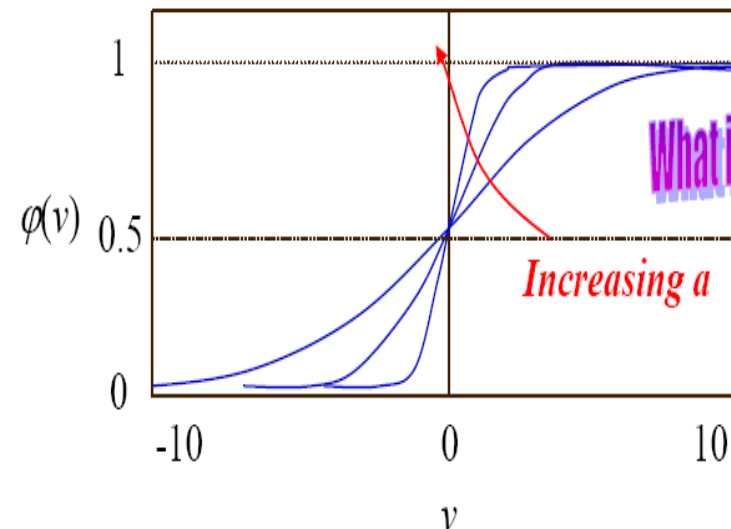
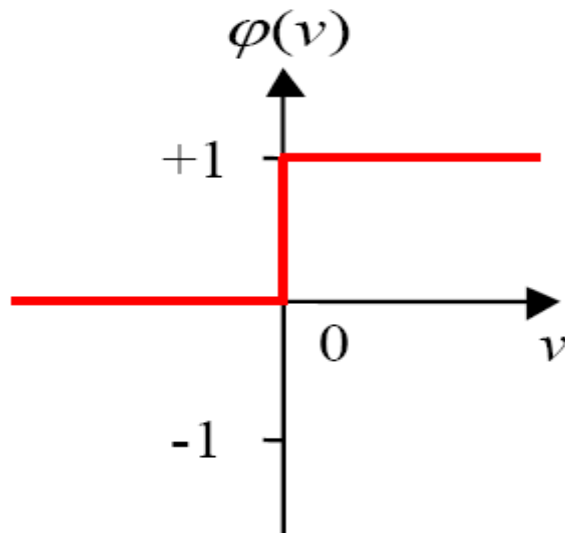
This theorem assures you that the good solution is out there. How to find it?

**That's why BP and other learning algorithms are proposed!**

## Function Approximation Examples:



**How about the threshold function? Is it continuous?**



**Can a discontinuous function be approximated by continuous function?**

**Sure!** 52

## MLP application example: Voice Recognition:

Torsten Reil

torsten.reil@zoo.ox.ac.uk

**Task: Learn to discriminate between two different voices  
saying “Hello”**

### Data

#### Sources

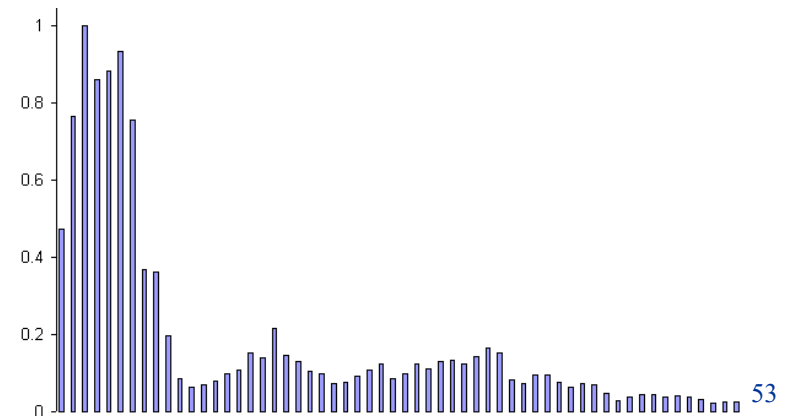
**Steve Simpson**

**David Raubenheimer**



#### Format

**Frequency distribution (60 bins)**



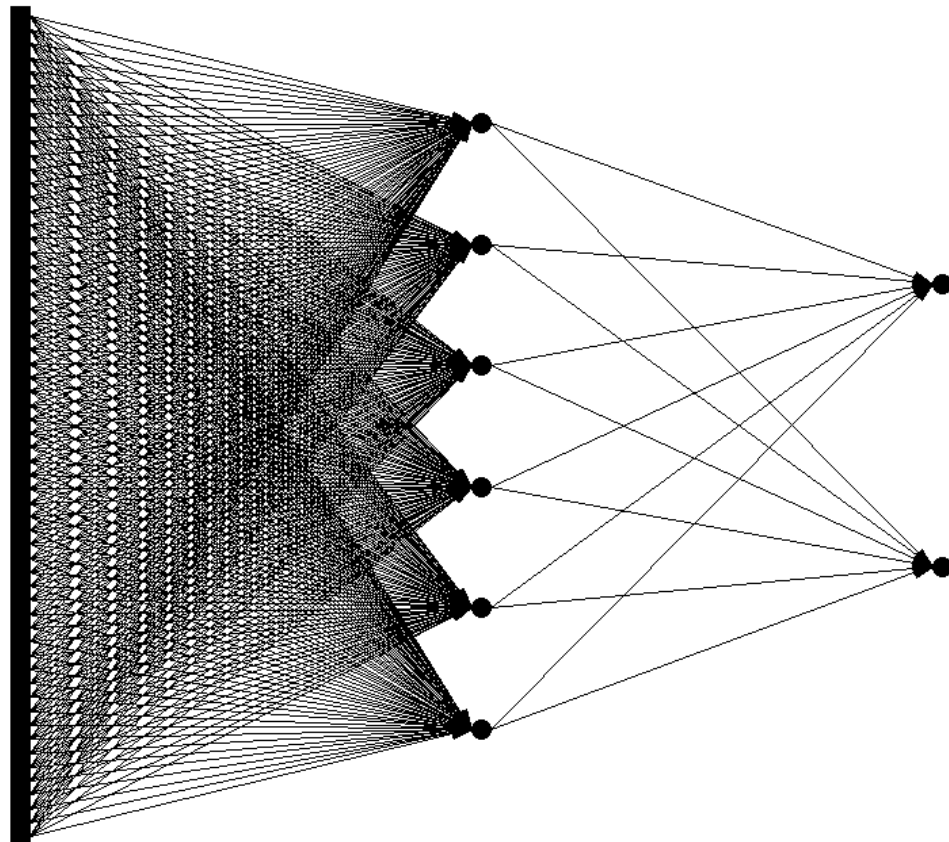
# Network architecture

## Feed forward network

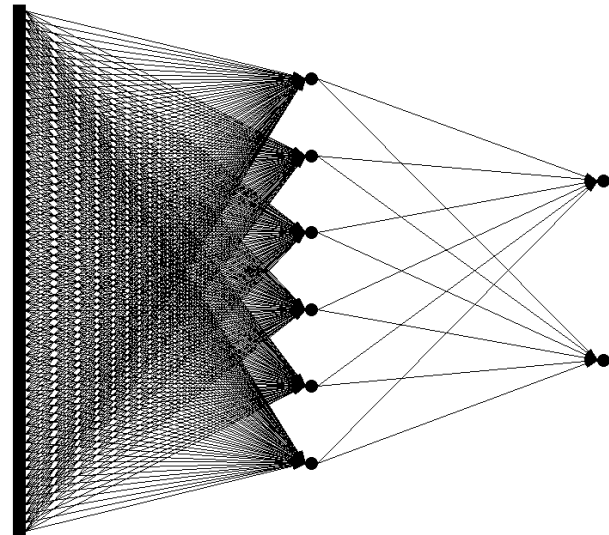
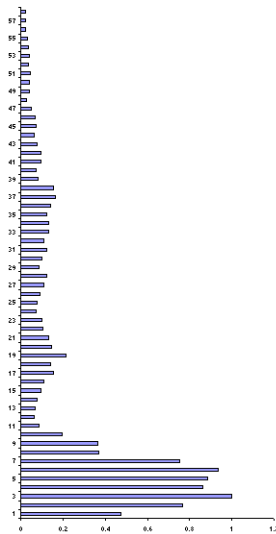
**60 input (one for each frequency bin)**

**6 hidden**

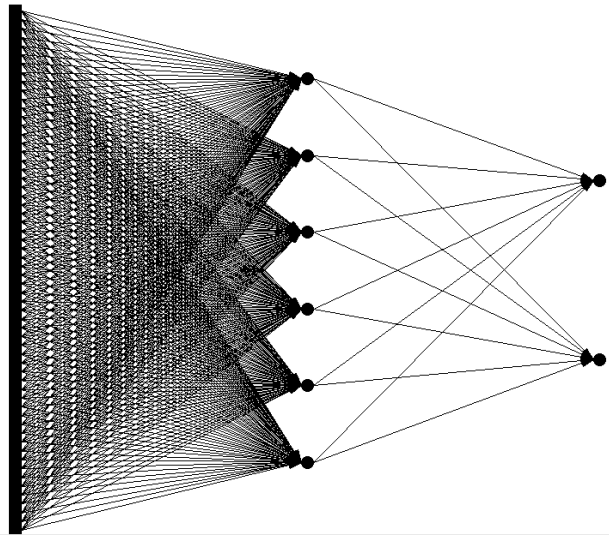
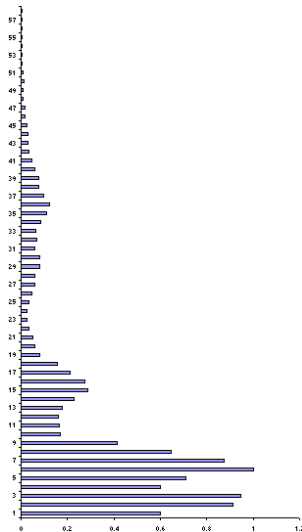
**2 output (0-1 for “Steve”, 1-0 for “David”)**



Steve



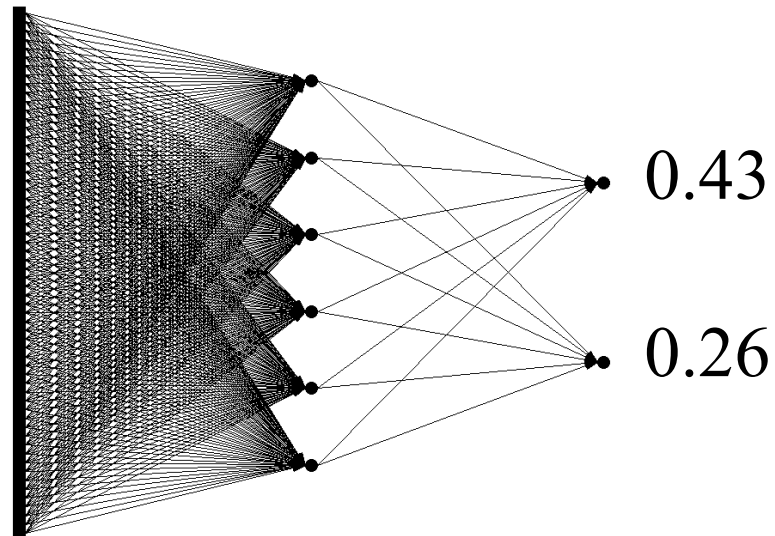
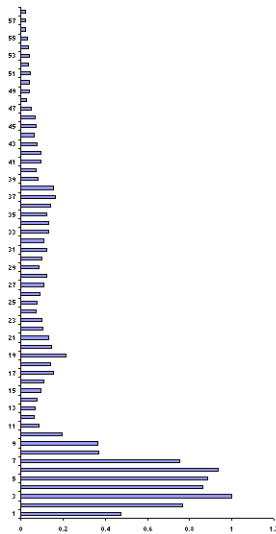
David



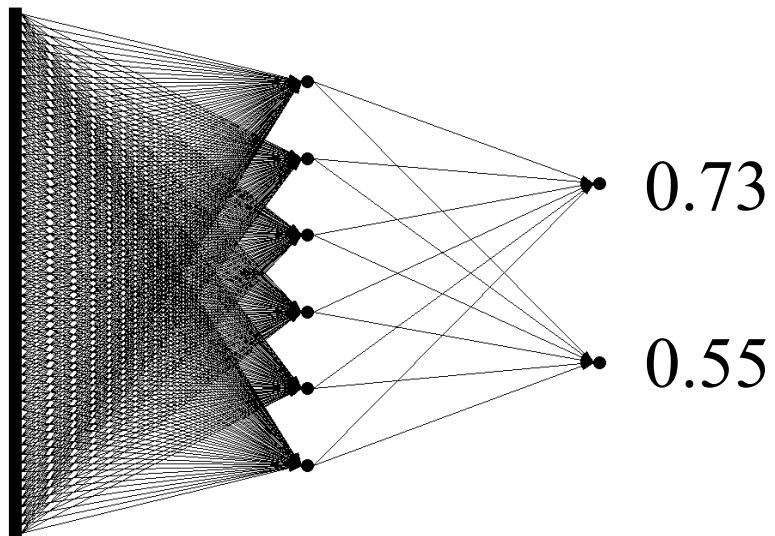
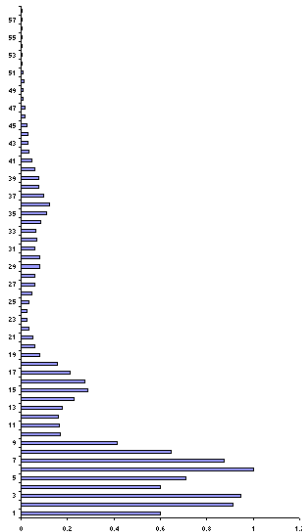


# Presenting the data (untrained network)

Steve



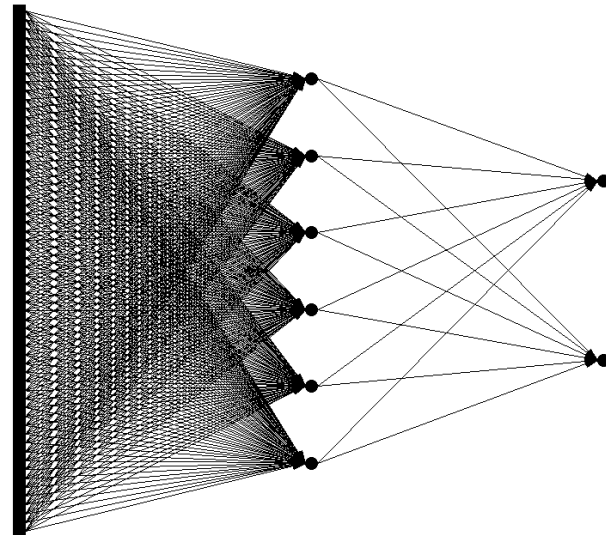
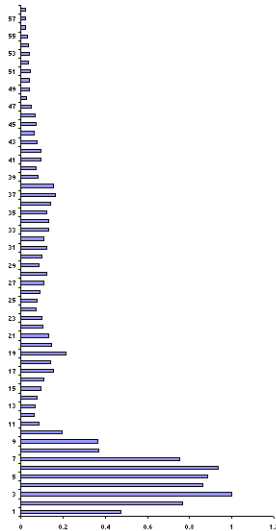
David





# Calculate error

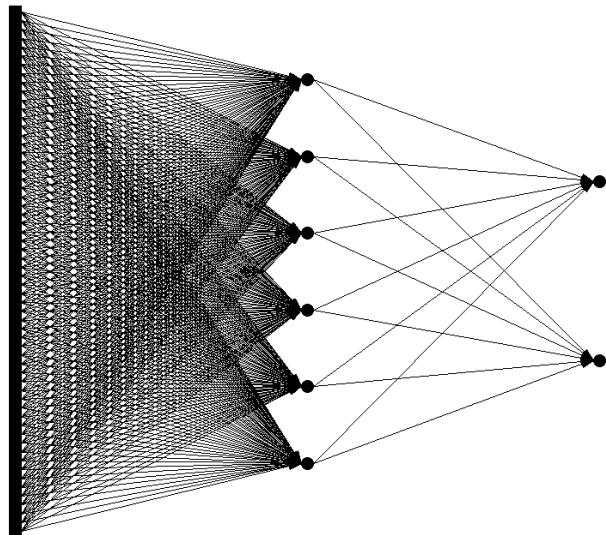
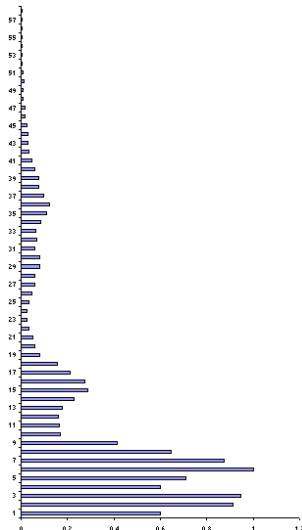
Steve



$$0 - 0.43 = -0.43$$

$$1 - 0.26 = 0.74$$

David

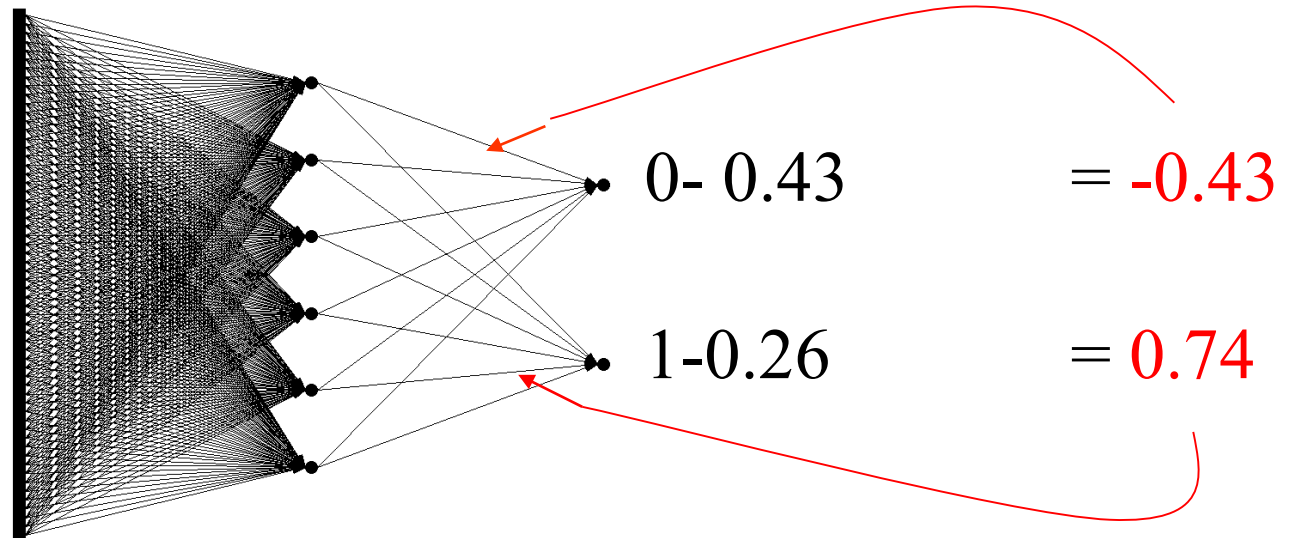
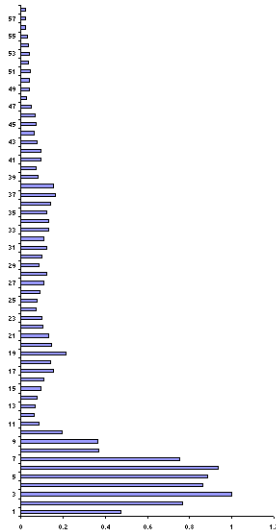


$$1 - 0.73 = 0.27$$

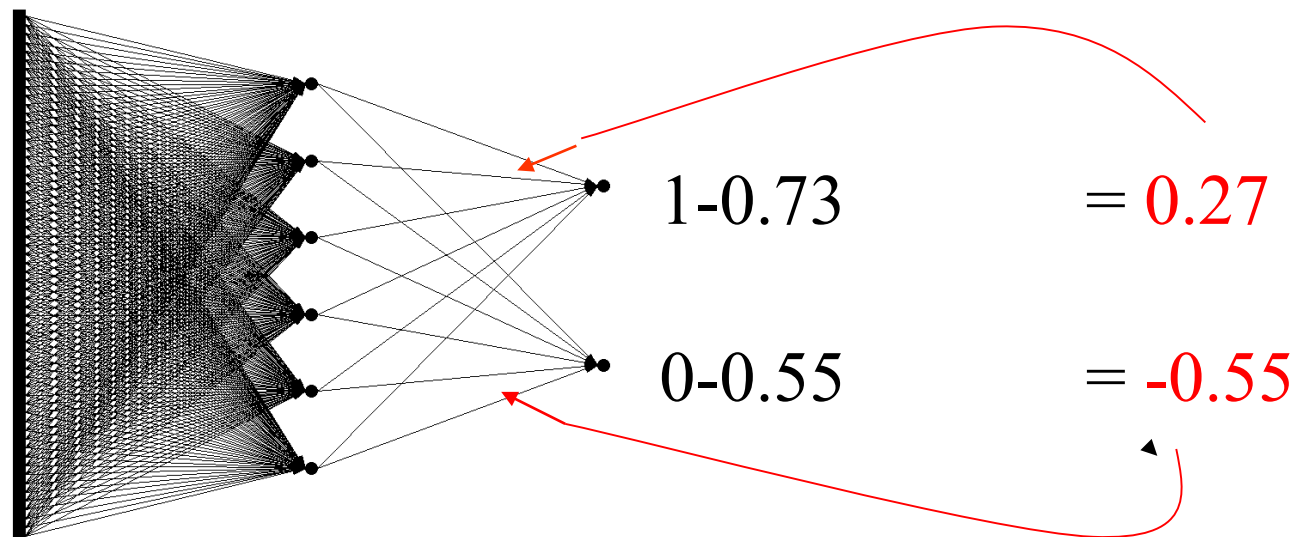
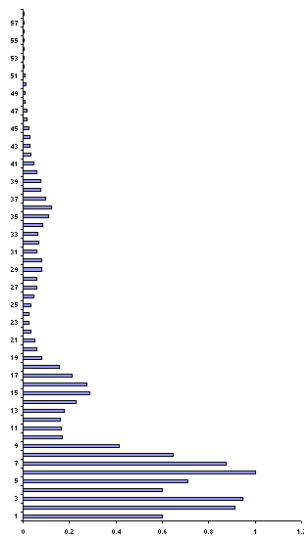
$$0 - 0.55 = -0.55$$

# Backpropagate error and adjust weights

Steve



David



## Repeat process (sweep) for all training pairs

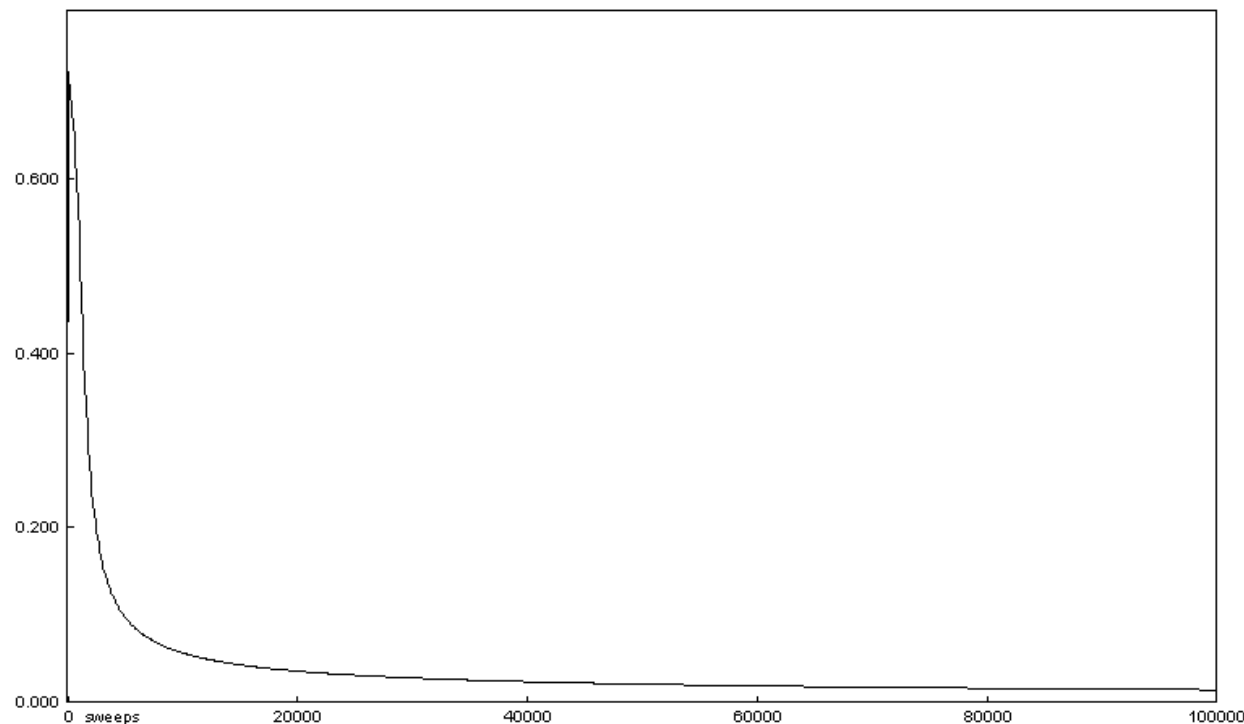
Present data

Calculate error

Backpropagate error

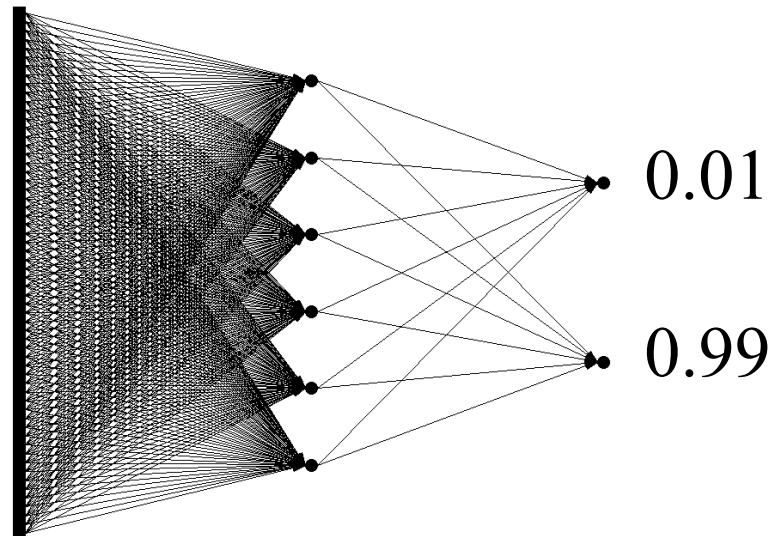
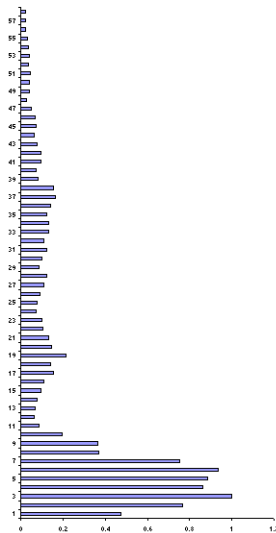
Adjust weights

Repeat process multiple times

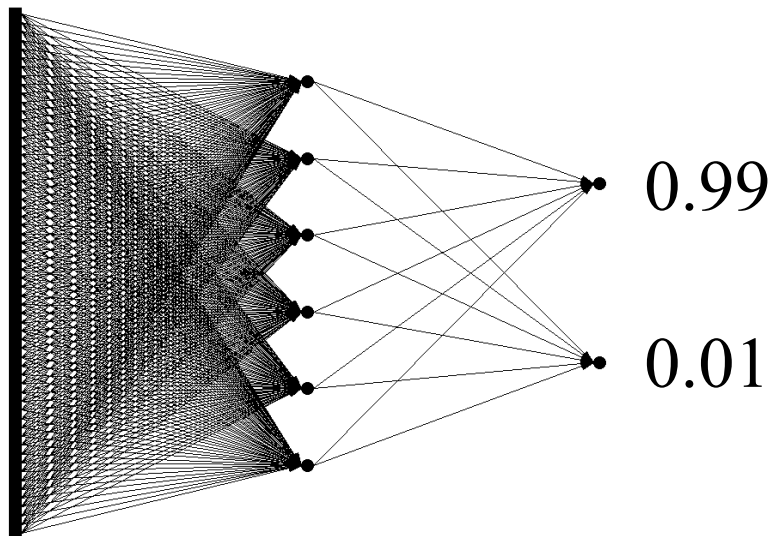
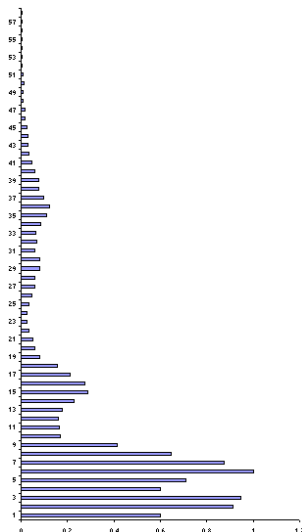


# Presenting the data (trained network)

Steve



David



## Results – Voice Recognition

Performance of trained network

**Discrimination accuracy between known “Hello”s**

100%

**Discrimination accuracy between new “Hello”s**

100%

Network has learnt to generalise from original data.

Networks with different weight settings can have same functionality. (The final solution is not unique!)

Trained networks ‘concentrate’ on lower frequencies.

Network is robust against non-functioning nodes.

Let's look at another more famous example: **NETtalk**

Terrence Sejnowski and Charles Rosenberg in 1987.

NETtalk was created to learn how to correctly pronounce English from written English text.

This is not straightforward since English is an especially difficult language to pronounce because of its irregular spelling.

The network was, of course, not explicitly given any of the rules of English pronunciation.

It had to learn them just from the corrections it received after each of its attempts during the training sessions.

The input was fed into the network, letter by letter, in a special way. The overall output of NETtalk was a string of symbols related to spoken sounds. To make the demonstration more vivid, this output was coupled to a speech synthesizer that produced speech sounds from the output of NETtalk.

It is particularly fascinating when you hear the [audio examples](#) of the neural network as it progresses through training seems to progress from a **baby babbling** to what sounds like a young child **reading** a kindergarten text, making the occasional mistake, but clearly demonstrating learned the major rules of reading.

**How did they do it?**



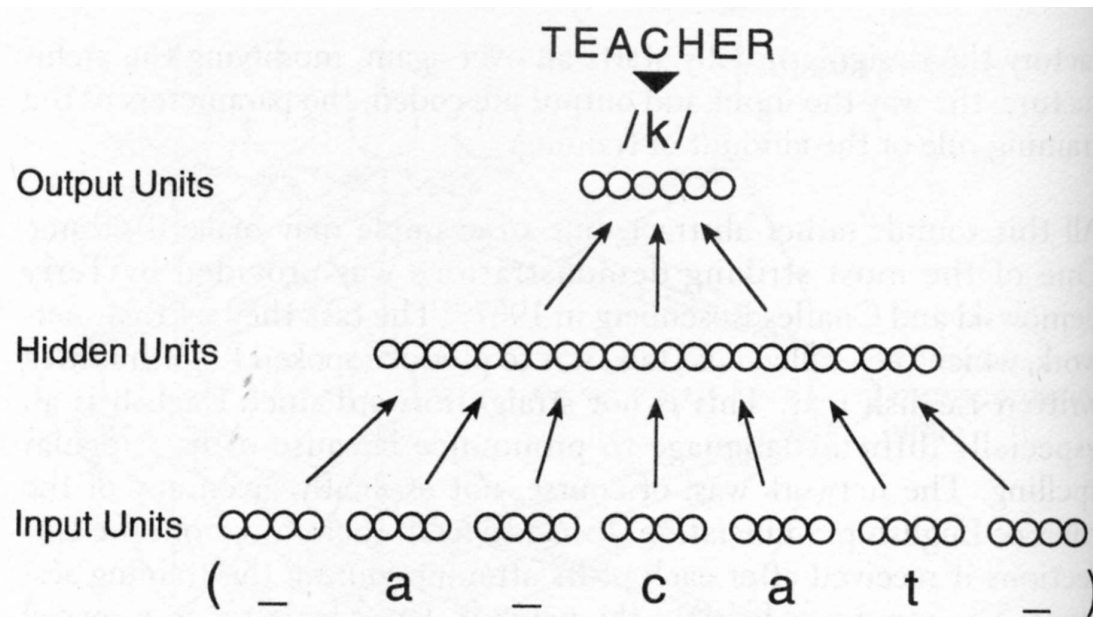


Fig. 56. A schematic drawing of the NETtalk network architecture, a specific example of the general scheme in Figure 5.5. A window of seven “letters” in an English text (“a cat” here) is fed to an array of 203 input units. Information from these units is transformed by an intermediate layer of 80 “hidden” units to produce patterns of activity in 26 output units.

## What are the inputs to the NETtalk?

Since the pronunciation of English depends on what letters lie before and after it, the input layer looks at string of **seven** letters at a time.

## Can you feed letters directly to MLP?

## How do you code the 26 English letters?

One input unit for each of the 26 letters, plus three for punctuation and word boundaries. Thus there are  $29 \times 7 = 203$  inputs

## What are the outputs?

The output layer has a neuron for each of the 21 “articulatory features” of the required phonemes plus 5 units to handle syllable boundaries and stresses. As an example, the consonants p and b are both called “labial stops” because they start with pursed lips.

## How many hidden layers?

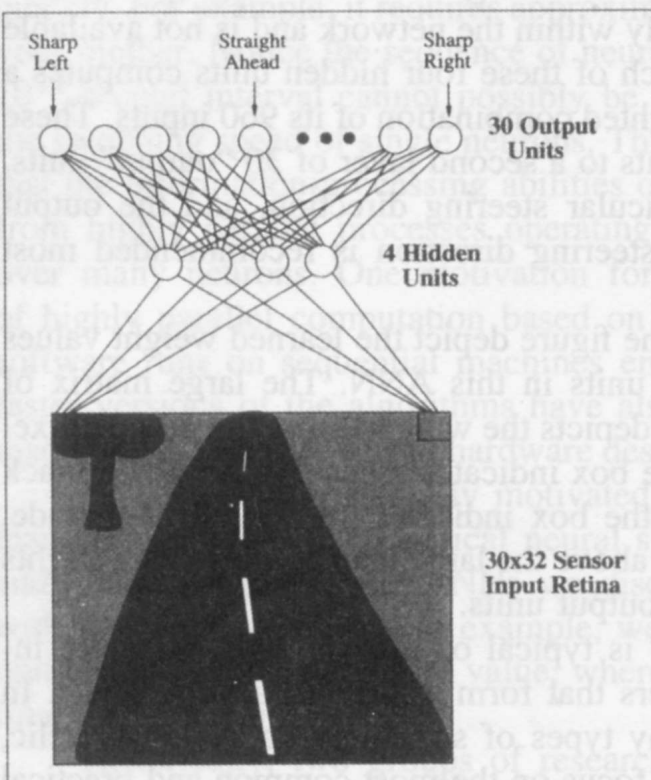
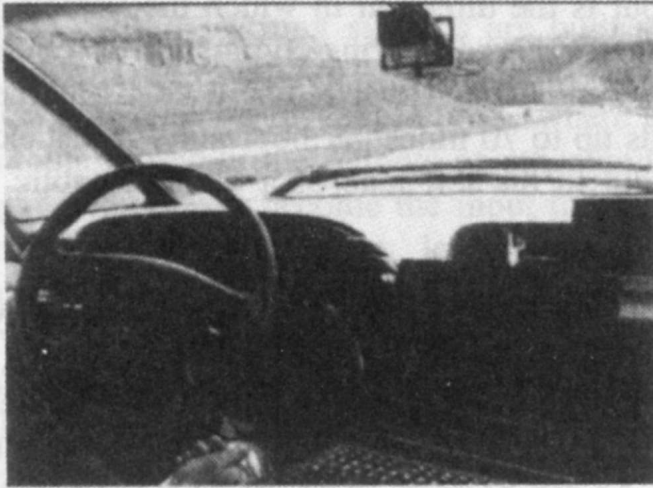
## How many hidden neurons?

Originally they used 80, then later increased to 120.

Structure of MLP: 203-80-26, total number of weights:  $204 \times 80 + 81 \times 26 = 18426$

# "Autonomous Land Vehicle In a Neural Network" ([ALVINN](#))

by Dean Pomerleau and Todd Jochem, 1993, CMU



## What are the inputs?

The image captured by the camera.

## How do you code an image?

30x32 pixels in the gray-level image  
30x32 matrix

## How many input neurons?

$30 \times 32 = 960$

## What are the 30 output units?

Each output neuron corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly

## How many hidden layers?

Only one!

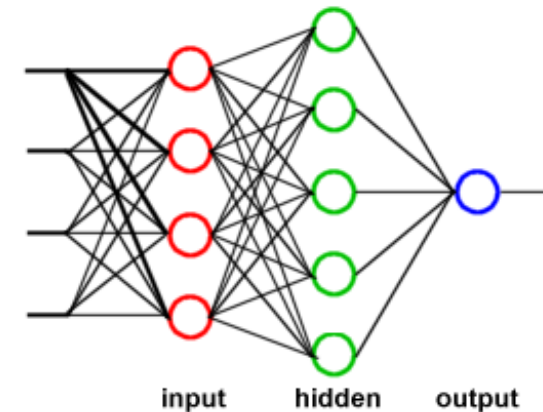
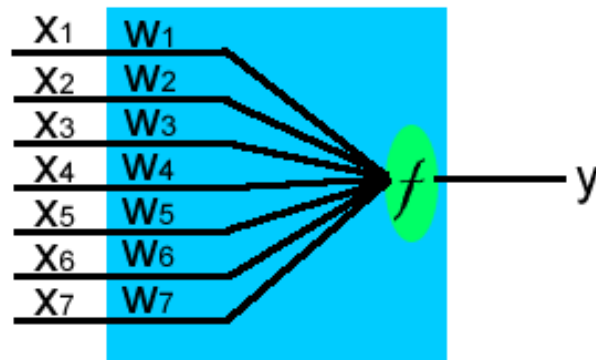
## How many hidden neurons?

Only four.

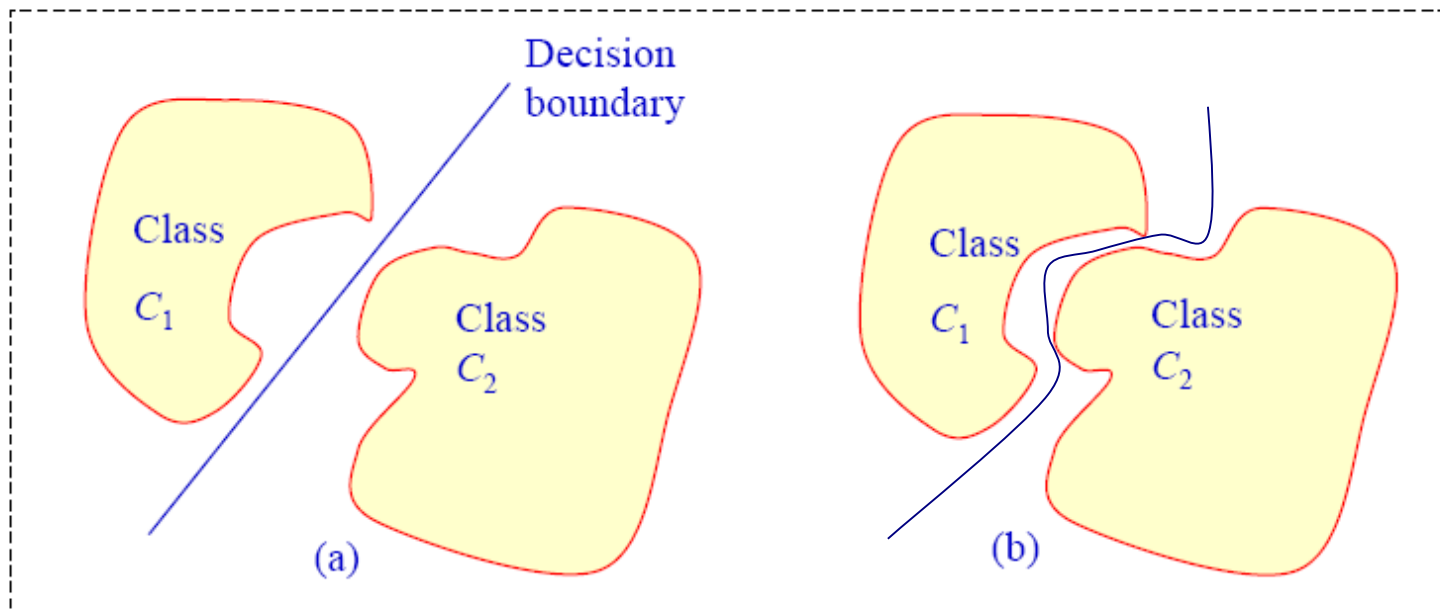
Architecture of MLP: 960-4-30

# of weights:  $961 \times 4 + 5 \times 30 = 3994$





Pattern Recognition Problem:

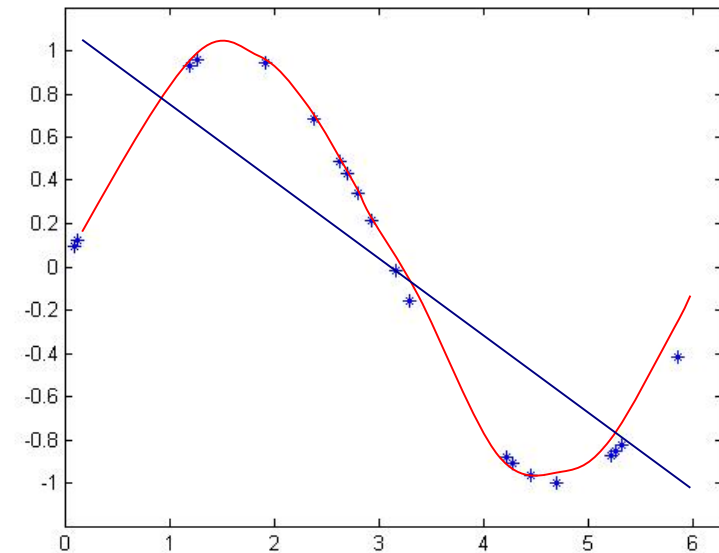
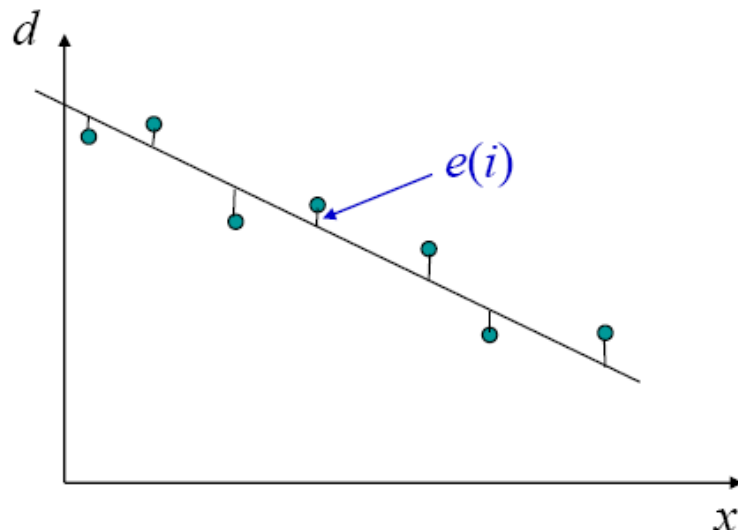


**Why can MLP solve the nonlinearly separable problem?**

The nonlinearly separable problem can be transformed into linearly separable problem  
in the feature space produced by the hidden layer!

# Single Layer Perceptron v.s. Multi-layer Perceptrons

## Regression Problem:



Multi-layer Perceptrons can approximate any bounded continuous functions!

The learning algorithms are based upon the steepest descent method:

$$w(n+1) = w(n) + \eta e(n)x(n)$$

Output Error

Input Signal

$$w_{ji}^{(s)}(k+1) = w_{ji}^{(s)}(k) + \eta^{(s)} \delta_j^{(s)} x_{out,i}^{(s-1)}$$

$\Delta w_{ji}^{(s)}$

Output Error

Input Signal

## Deficiencies of BP Trained NNs

Nobody is perfect. BP trained NNs have many limitations too.

**What is the most serious weakness associated with Neural Networks ?**

The net is essentially a black box!

The most powerful weapon of NN can turn against itself!

The Neural networks can learn almost everything due to its plasticity!

The synaptic weights can adapt to any learning task!

**On the other hand, do you have any idea about how to interpret the weights?**

Almost impossible!

- ♦ It may provide a desired mapping between input and output vectors ( $x, y$ ) but does not have the information of why a particular  $x$  is mapped to a particular  $y$ .
- ♦ It thus cannot provide an intuitive (e.g., causal) explanation for the computed result.
- ♦ This is because the hidden units and the learned weights do not have a semantics. What can be learned are operational parameters, not general, abstract knowledge of a domain.

## Deficiencies of BP Trained NNs

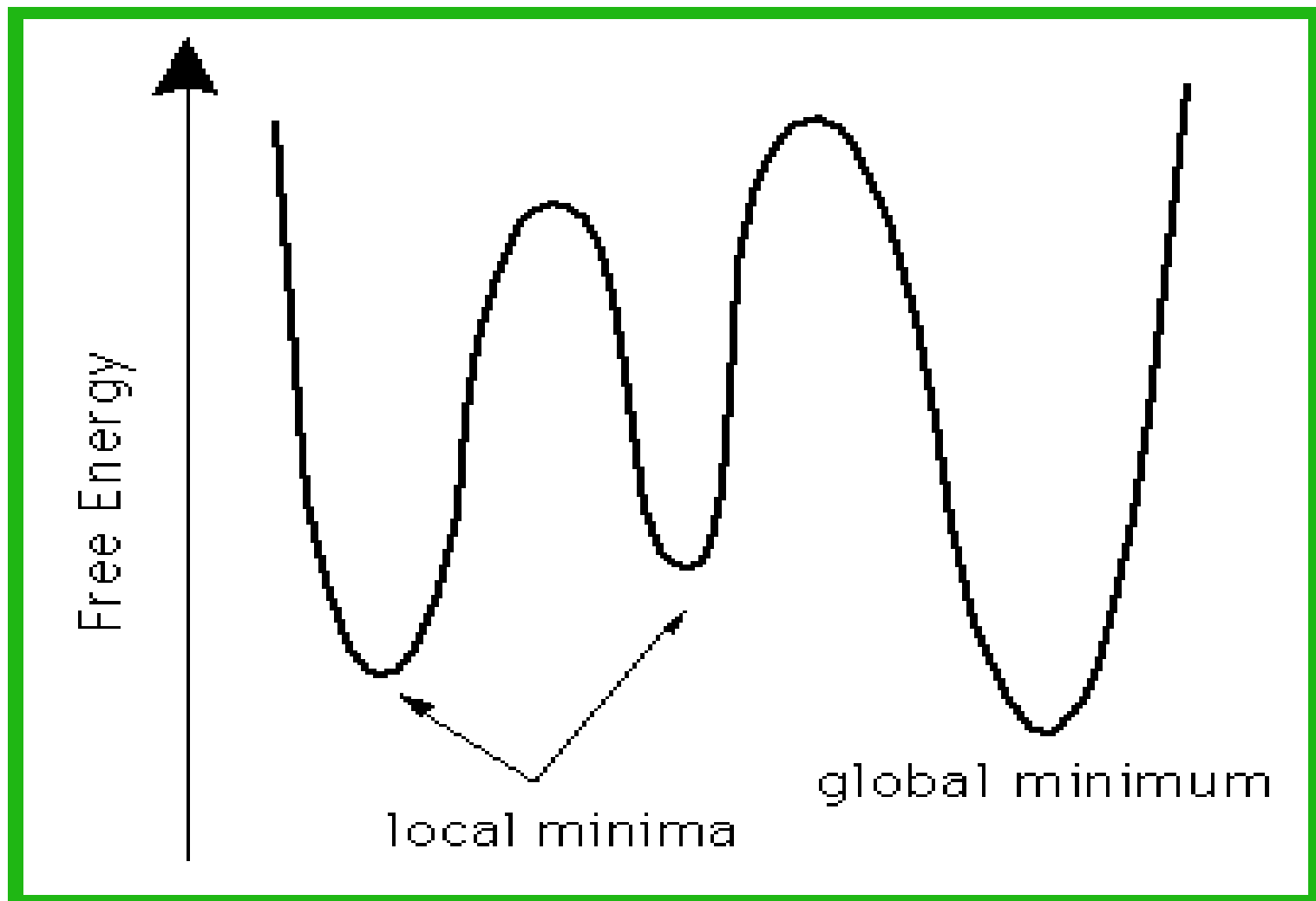
There are other minor limitations too:

1. Learning often takes a *long time* to converge:

◆Complex functions often need hundreds or thousands of epochs.

2. Gradient descent approach only guarantees to reduce the total error to a *local minimum*.

All gradient based methods like the gradient descent and its variations share the same weakness: **cannot escape from local minimum**



♦ Possible remedies for local minima problem:

- Try nets with different # of hidden layers and hidden units (they may lead to different error surfaces, some might be better than others).
- Try different initial weights (different starting points on the surface).
  - Forced escape from local minima by random perturbation (e.g., *simulated annealing*).

The learning (accuracy, speed, and generalization) is highly dependent on a set of learning parameters:

- ♦ Initial weights, learning rate, # of hidden layers and # of units...
- ♦ Many of them can only be determined empirically (via experiments).

We will discuss all these design issues next week.

Q & A...

**THANK YOU!**