

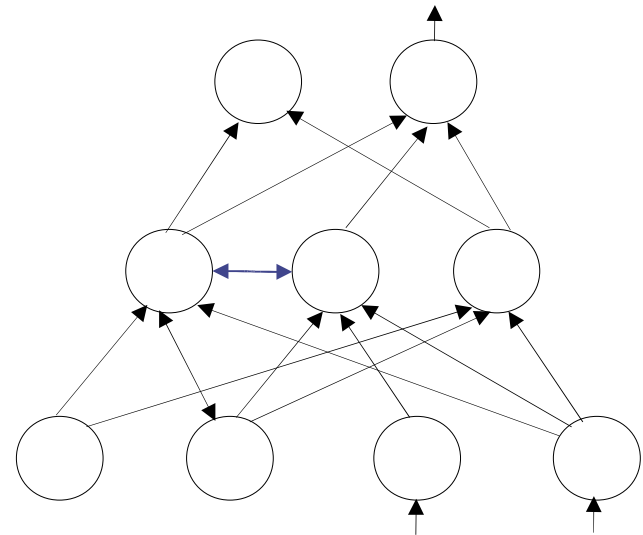
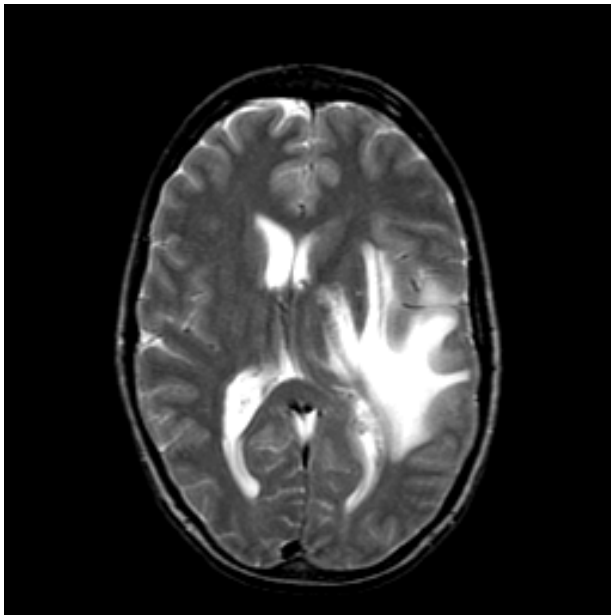
# What is a Neural Network?

A massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:

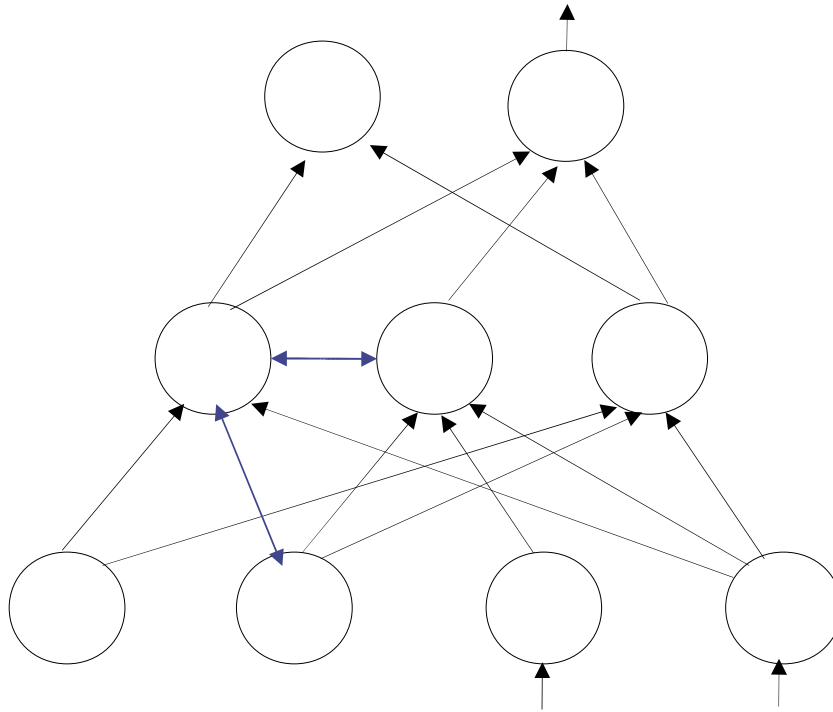
- ⊕ Knowledge is acquired from the environment through a learning process
- ⊕ Inter-neuron connection strength, known as synaptic weights, are used to store the acquired knowledge.

# Human Brain

- ⊕ Robust, Fault Tolerant, Massively Parallel, Capable of Learning from Examples...

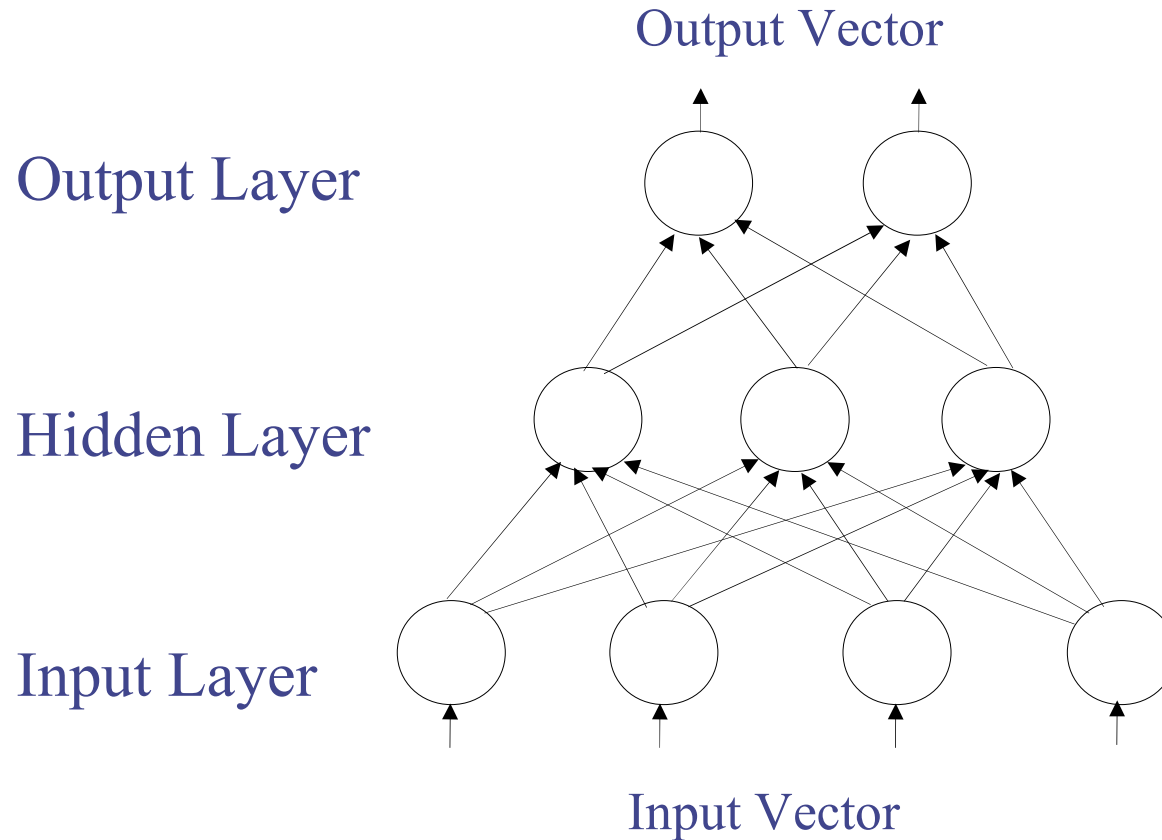


# Artificial Neural Networks



The most interesting properties of neural networks do not arise from the functionality of each neuron, but from the effect of the interconnection of neurons.

# Feed-forward Networks



The network computes a function  $f : \mathcal{R}^r \rightarrow \mathcal{R}^s$

# Learning from Examples in Neural Networks

The network computes a function

$$f : \mathcal{R}^r \rightarrow \mathcal{R}^s$$

A set of examples (input vectors and their respective *target output vectors*) defines a new function

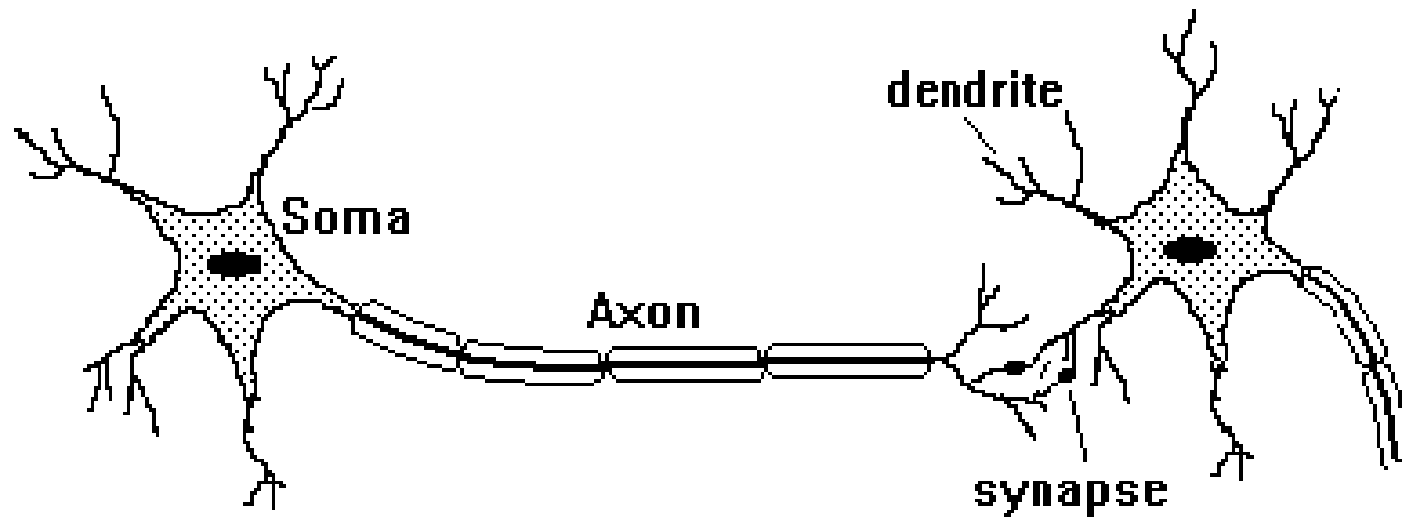
$$g : \mathcal{R}^r \rightarrow \mathcal{R}^s$$

We want to change the function  $f$  computed by the network - by changing its weights ( $W_{ij}$ ) - according to the set of examples, in order to approximate  $g$ .

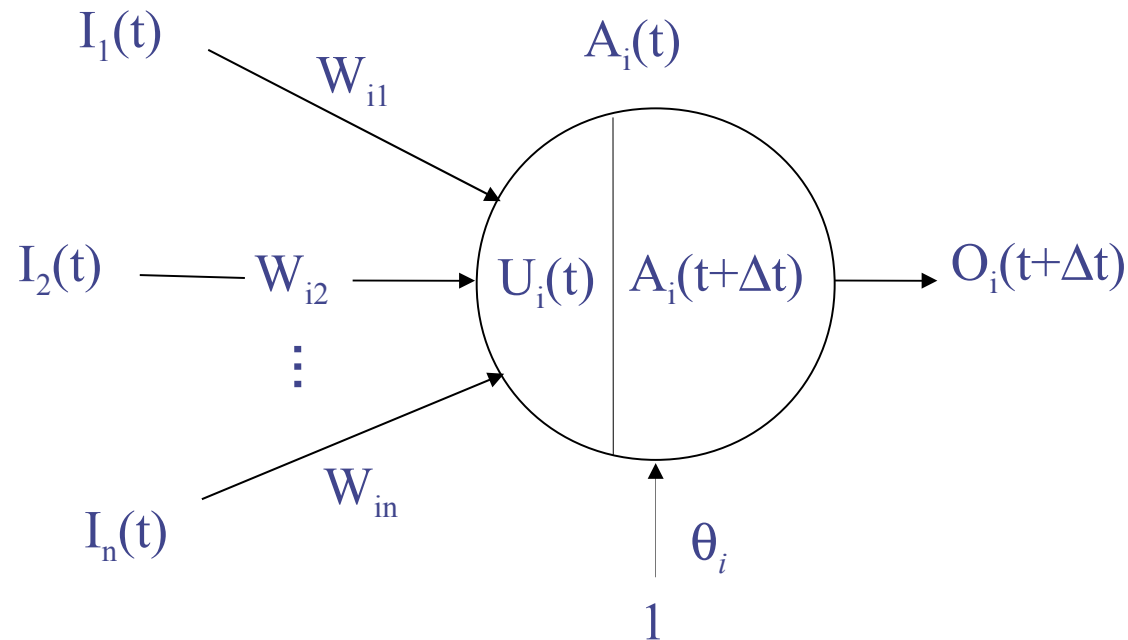
# Applications

- Pattern Recognition (e.g.: face recognition)
- Classification (e.g.: fault diagnosis, DNA)
- Associative Memory (e.g.: image compression)
- Clustering (e.g.: credit analysis, fraud analysis)
- with applications in Aerospace, Finance, Medical, Security, Transports, etc.

# The Neuron



# Model of a Neuron



$I_i(t)$ : Input Vector

$W_{ij}$ : Weight Vector;  $\theta_i$ : Bias

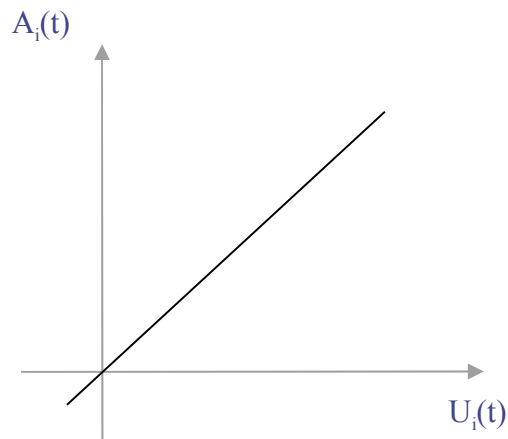
$U_i(t)$ : Input Potential;  $U_i(t) = g_i(I_i(t), W_{ij}, \theta_i)$

$A_i(t)$ : Activation State;  $A_i(t+\Delta t) = h_i(A_i(t), U_i(t))$

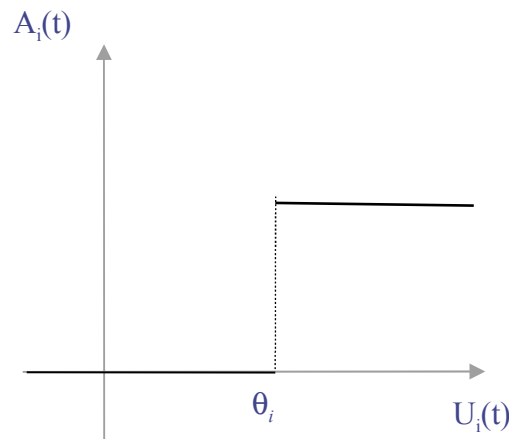
$O_i(t)$ : Output;  $O_i(t) = f_i(A_i(t))$

# Types of Activation Function $h(x)$

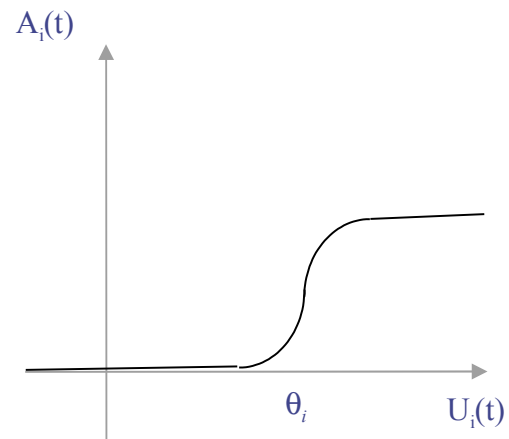
Linear, Non-Linear (step function),  
Semi-Linear (sigmoid function), etc.



Linear

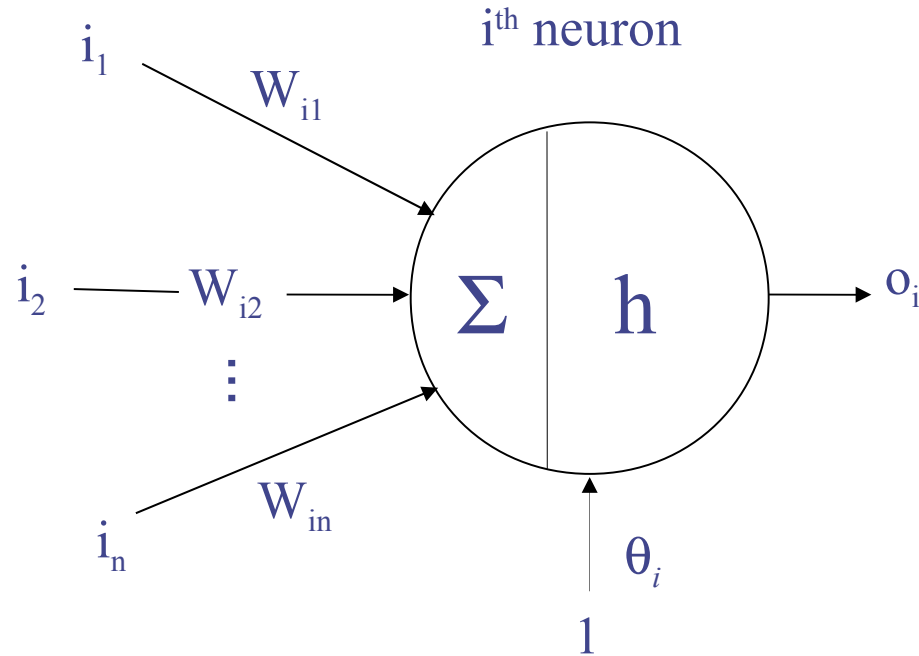


Nonlinear



Semi-linear

# McCulloch-Pitts Neuron



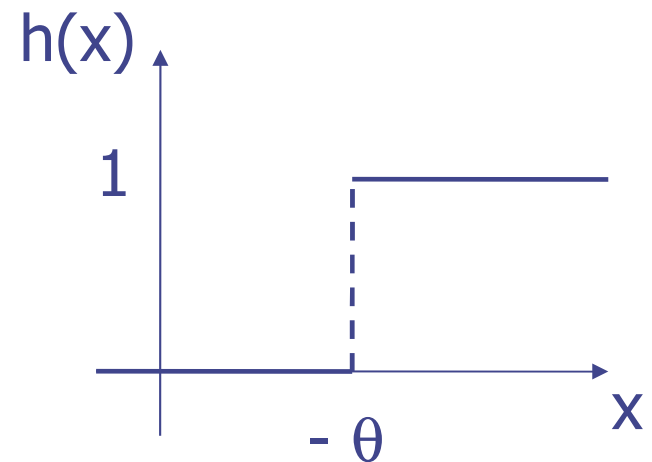
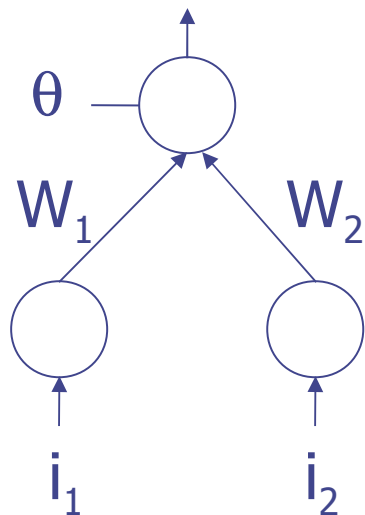
$$U_i = \sum_j (W_{ij} \cdot i_j) + \theta_i$$

$o_i = h(U_i)$ , where  $h(x) = 1$  if  $x > 0$  and 0 otherwise.

# The Perceptron

- Uses McCulloch-Pitts Neurons
- Contains  $n$  input neurons, no hidden neuron, and 1 output neuron

$$o = h(W_1 i_1 + W_2 i_2 + \theta)$$

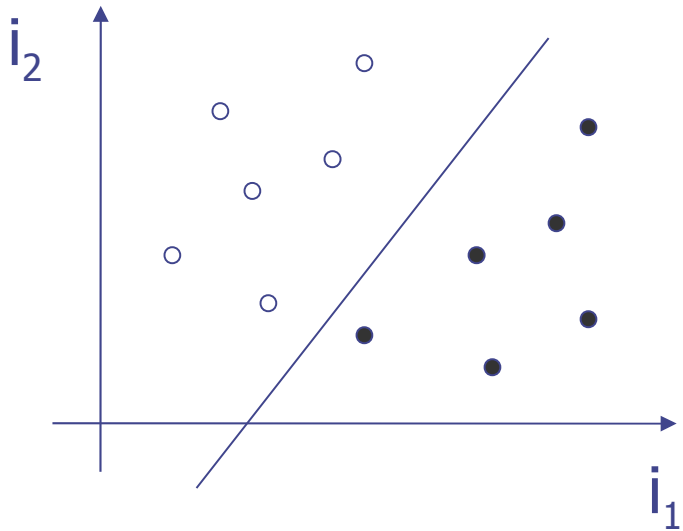


Bias = - Threshold

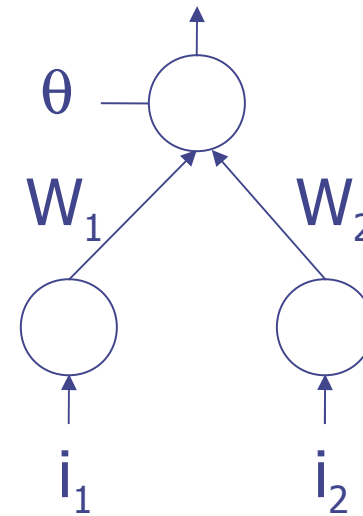
Note: Input neurons have *identity* as activation function!

# The Perceptron (Cont.)

Distinguishes Two Classes of Data (in the n-dimensional space, by applying a n-dimensional hyper-plane)



$$o = h(W_1 i_1 + W_2 i_2 + \theta)$$



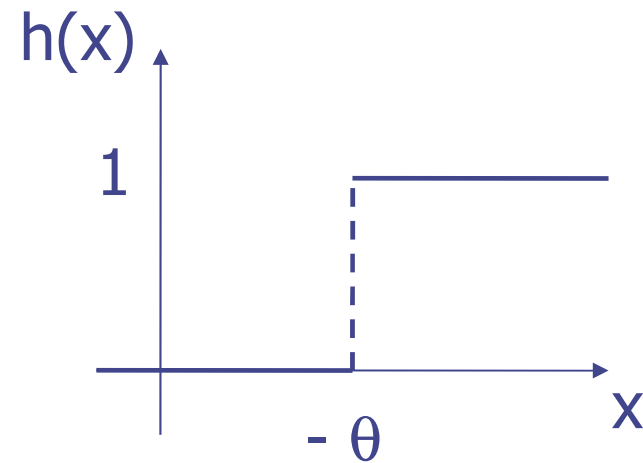
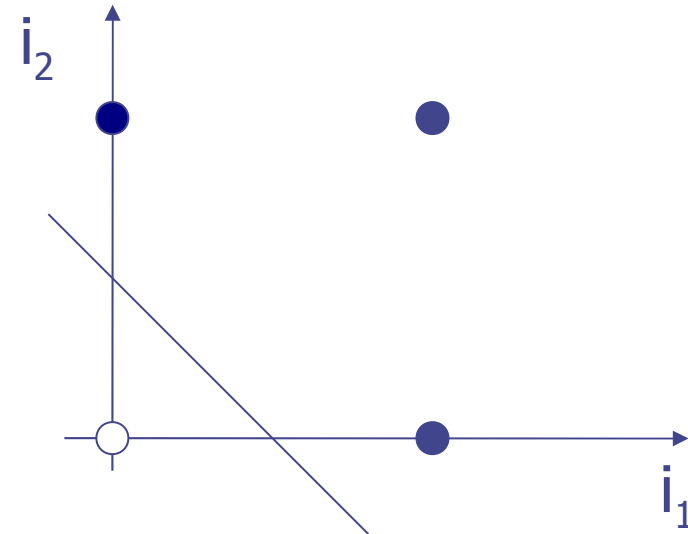
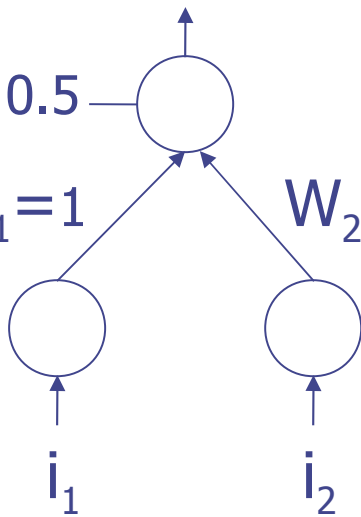
# Example: OR

$$o = h(W_1 i_1 + W_2 i_2 + \theta)$$

$$\theta = -0.5$$

$$W_1 = 1$$

$$W_2 = 1$$



# Learning Algorithm (Perceptron)

1. Initialise the weights randomly;
2. For each example  $(\mathbf{i}, t)$  do:

$$\Delta \mathbf{W} = \eta (t - o(i)) \mathbf{i}$$

Until  $t = o(i)$  for all examples.

$\mathbf{i}$  = input vector

$o$  = network's output

$t$  = target output ( $t \in \{0, 1\}$ )

$\eta \in \mathfrak{R}^+$  is called the Learning Rate

Note: The algorithm always terminates when the set of examples is linearly separable.

# Learning $i_1$ OR $i_2$

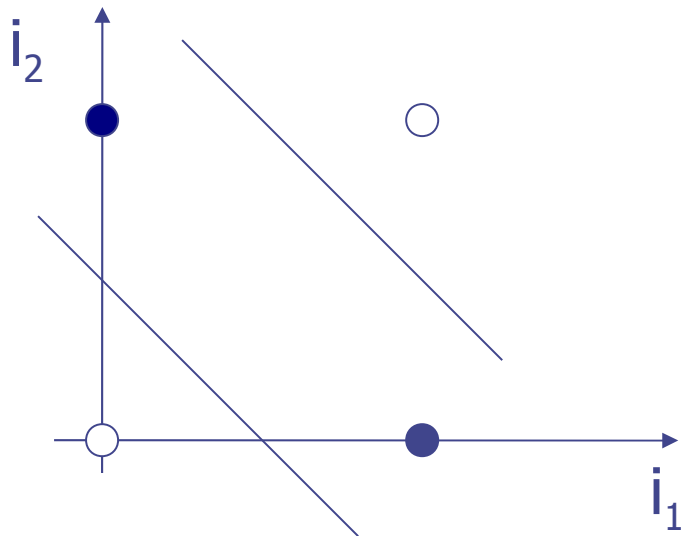
$$i = (1, i_1, i_2)$$

$$W = (\theta, W_1, W_2)$$

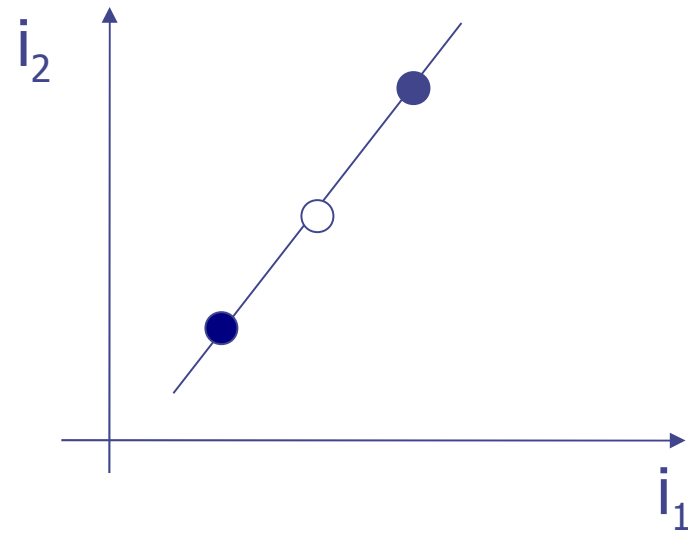
W	i	t	o	$\Delta W$
(-2,-2,0)	(1,0,0)	<b>0</b>	0	
(-2,-2,0)	(1,0,1)	<b>1</b>	0	(1,0,1)
(-1,-2,1)	(1,1,0)	<b>1</b>	0	(1,1,0)
(0,-1,1)	(1,1,1)	<b>1</b>	0	(1,1,1)
(1,0,2)			1	(-1,0,0)
(0,0,2)			1	
(0,0,2)			0	(1,1,0)
(1,1,2)			1	
(1,1,2)			1	(-1,0,0)
(0,1,2)			1	
(0,1,2)			1	
(0,1,2)			1	
(0,1,2)			<b>0</b>	
(0,1,2)			<b>1</b>	
(0,1,2)			<b>1</b>	
(0,1,2)			<b>1</b>	

$$\eta = 1$$

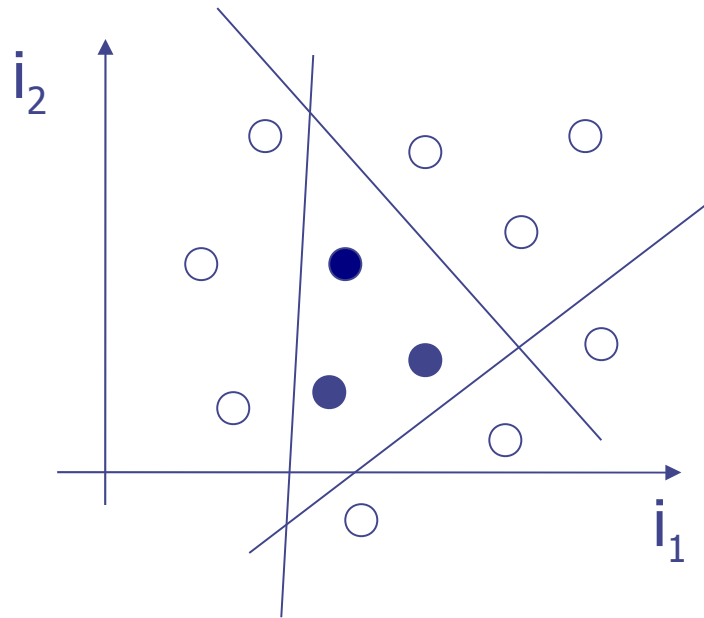
# Perceptron's Linear Separability



XOR

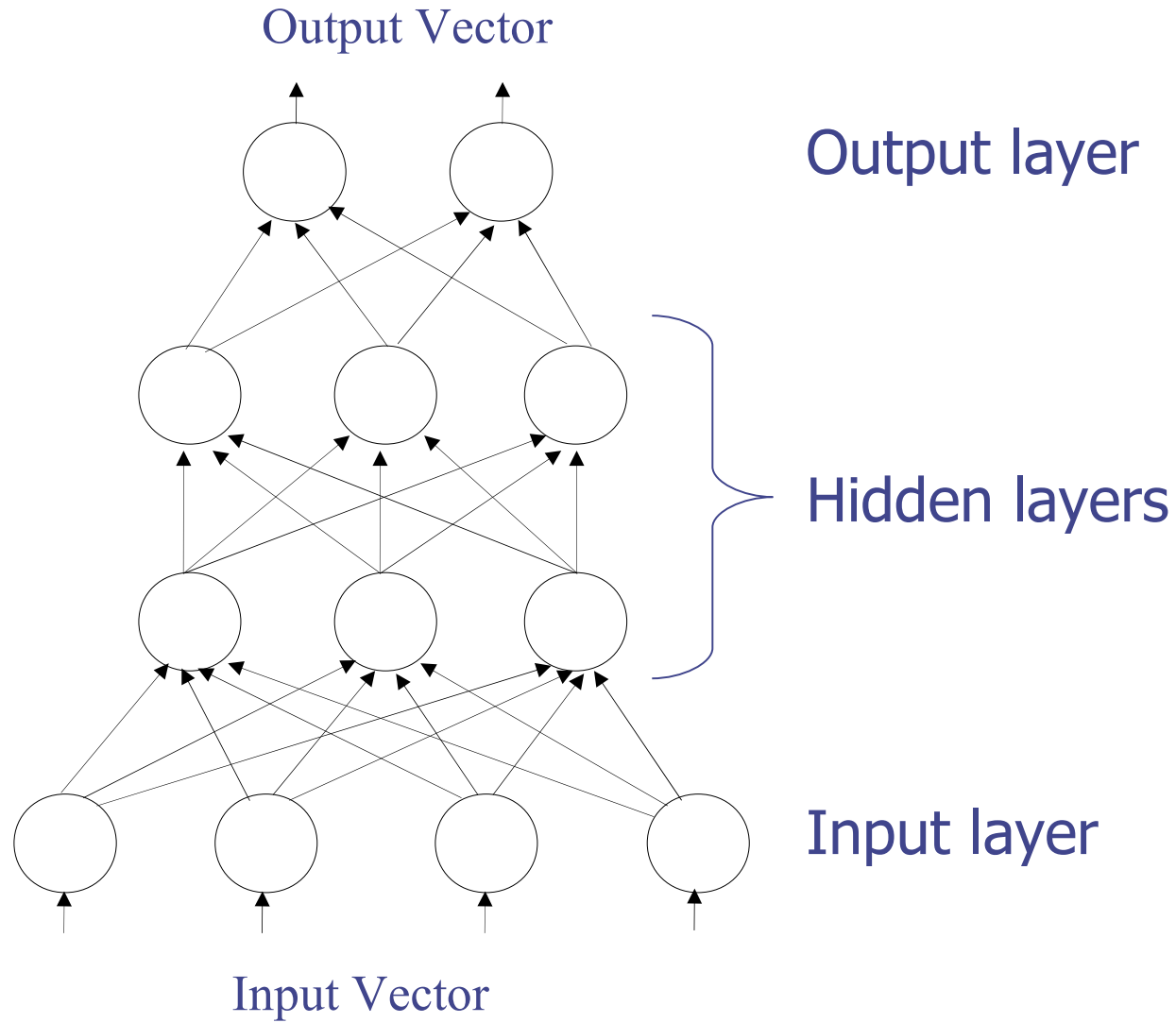


# Perceptron's Linear Separability



Requires More Inputs or Hidden Neurons!

# Multilayer Perceptron



# Multilayer Perceptrons are Universal Approximators

Theorem [Cybenko, 1989]: Let  $h:\mathfrak{R}\rightarrow\mathfrak{R}$  be a continuous and sigmoid function, i.e.  $\lim_{x\rightarrow-\infty} h(x) = 0$  and  $\lim_{x\rightarrow\infty} h(x) = 1$ . Then, finite sums of the form:

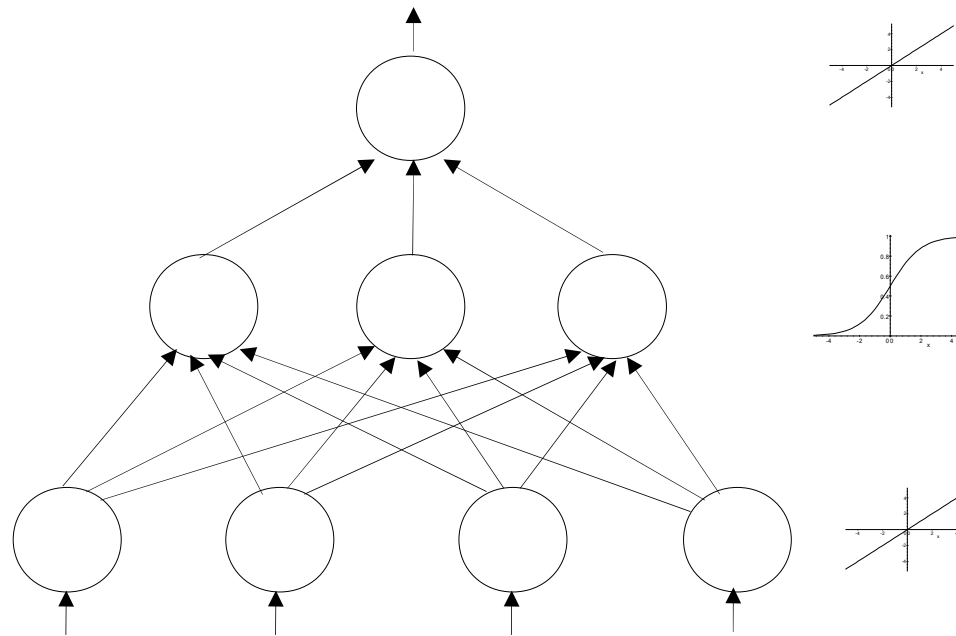
$$g(x) = \sum_j \alpha_j h(W_j x + \theta_j), \quad x \in I^n$$

with parameters  $\alpha_j, \theta_j \in \mathfrak{R}$  and  $W_j \in \mathfrak{R}^n$ , are such that  $|g(x) - f(x)| < \varepsilon$  for any continuous function  $f(x)$ .

Neural Networks with as few as a single hidden layer with sigmoid activation function can approximate virtually any function of interest.

# Learning in Multilayer Perceptrons

We need to find  $\alpha_j$ ,  $\theta_j$  and  $W_j$  such that  $g(x)$  approximates  $f(x)$

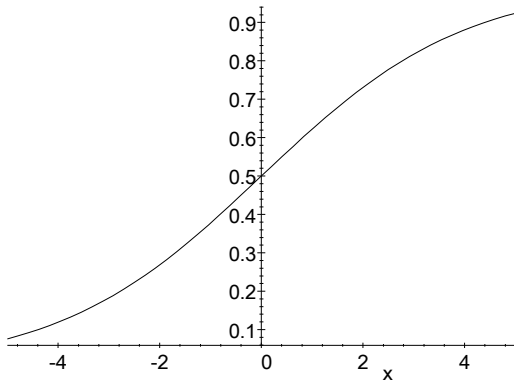


$$g(x) = \sum_j \alpha_j h(W_j x + \theta_j)$$

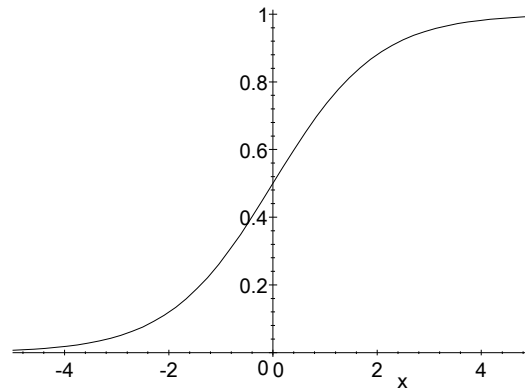
# Semi-Linear Activation Function

$$h(x) = 1 / ( 1 + e^{-\beta x} )$$

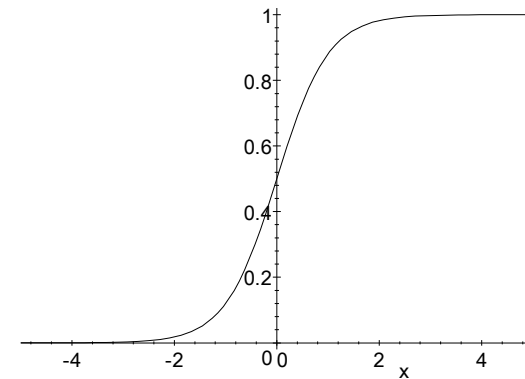
$$h'(x) = \beta e^{-\beta x} / ( 1 + e^{-\beta x} )^2$$



$$\beta = 1/2$$



$$\beta = 1$$



$$\beta = 2$$

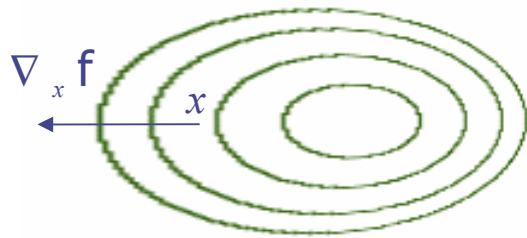
# Backpropagation

- ⊕ A computationally efficient method for training multilayer perceptrons.
- ⊕ The neural learning algorithm most successfully applied in industry.
- ⊕ It computes an estimate of the gradient of an error function ( $E_{\mathbf{w}}$ ) w.r.t. a set of weights  $\mathbf{W}$

$$\nabla E_{\mathbf{w}} = \frac{\partial E_{\mathbf{w}}}{\partial W_{11}}, \frac{\partial E_{\mathbf{w}}}{\partial W_{12}}, \dots, \frac{\partial E_{\mathbf{w}}}{\partial W_{ij}}$$

# Gradient Descent

The gradient of  $E_w$  is the vector pointing in the direction of the fastest growth of  $E_w$ , perpendicular to contour lines.

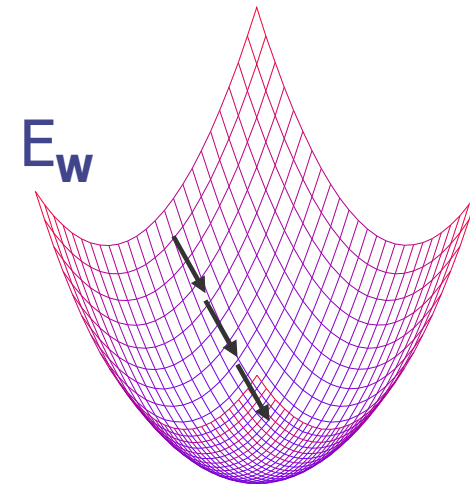


The iterative application of changes

$$\Delta W = -\eta \cdot \nabla E_w$$

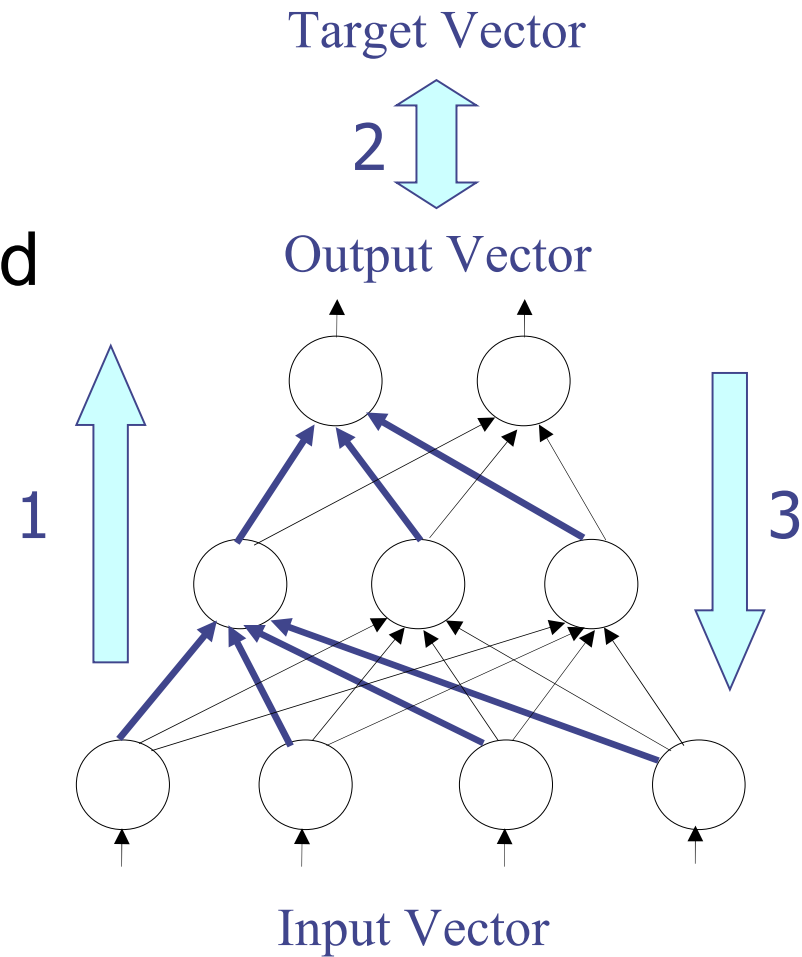
to  $W$  tries to minimise

$$E_w = \frac{1}{2} \sum_i (o_i - t_i)^2$$

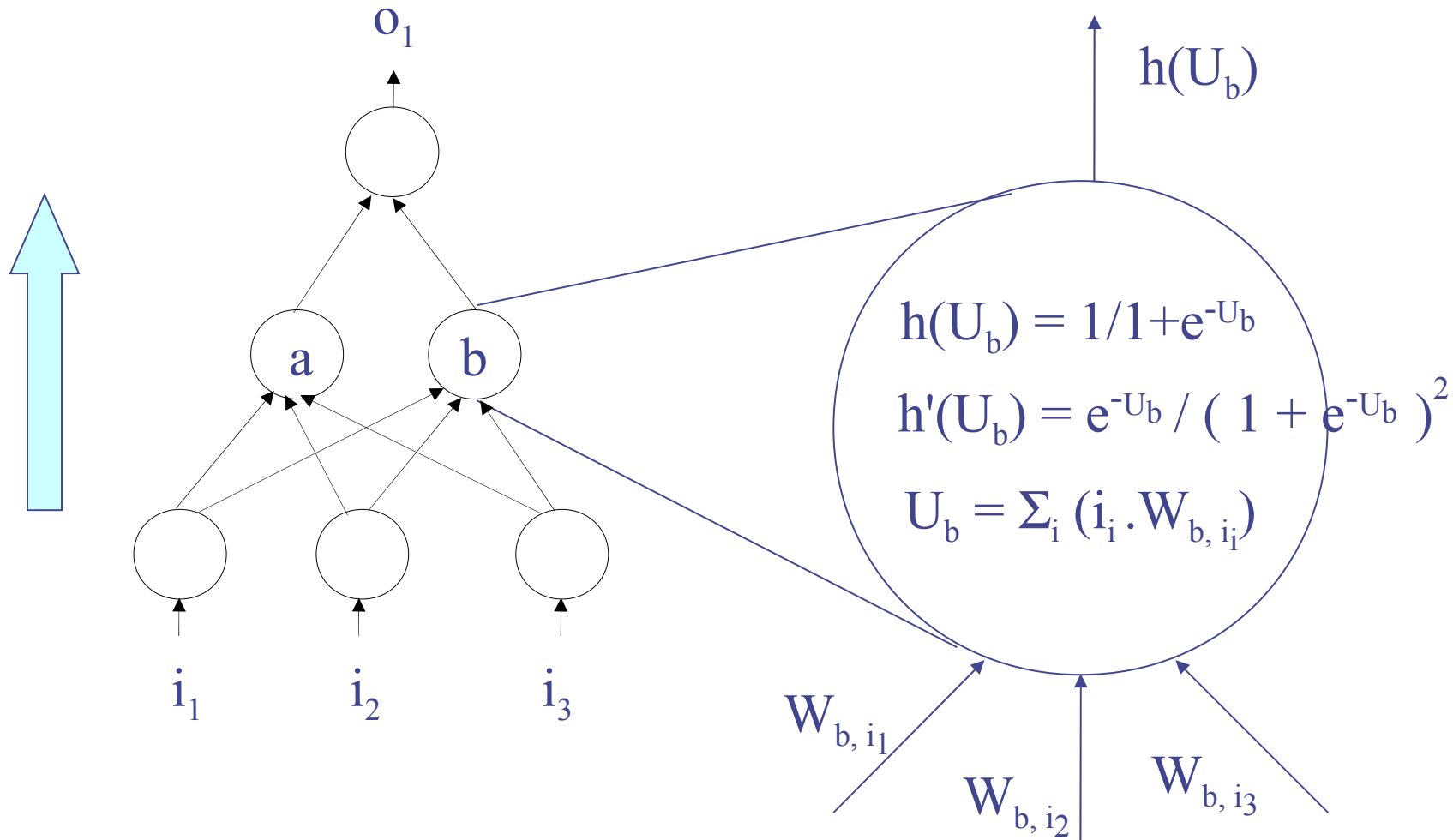


# Backpropagation Learning Algorithm

1. Propagate
2. Compute Error
3. Back-propagate Error and Change Weights

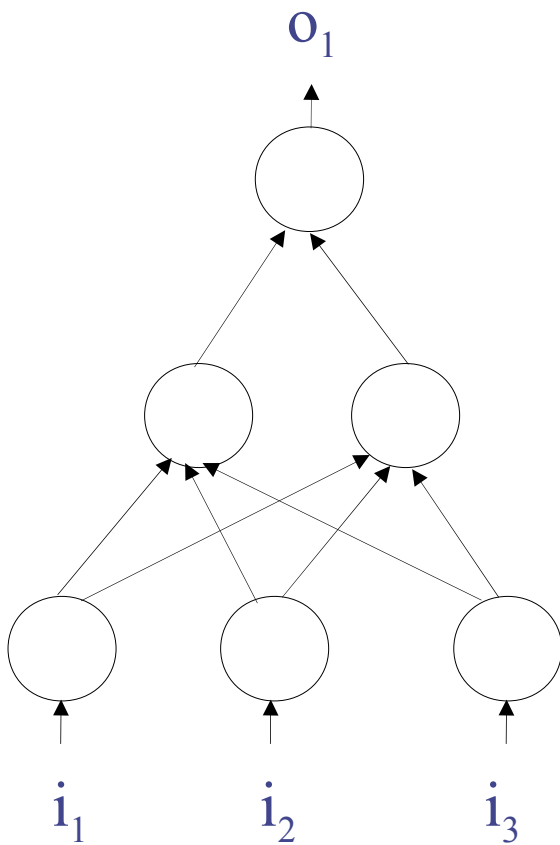


# Propagate



# Compute Error

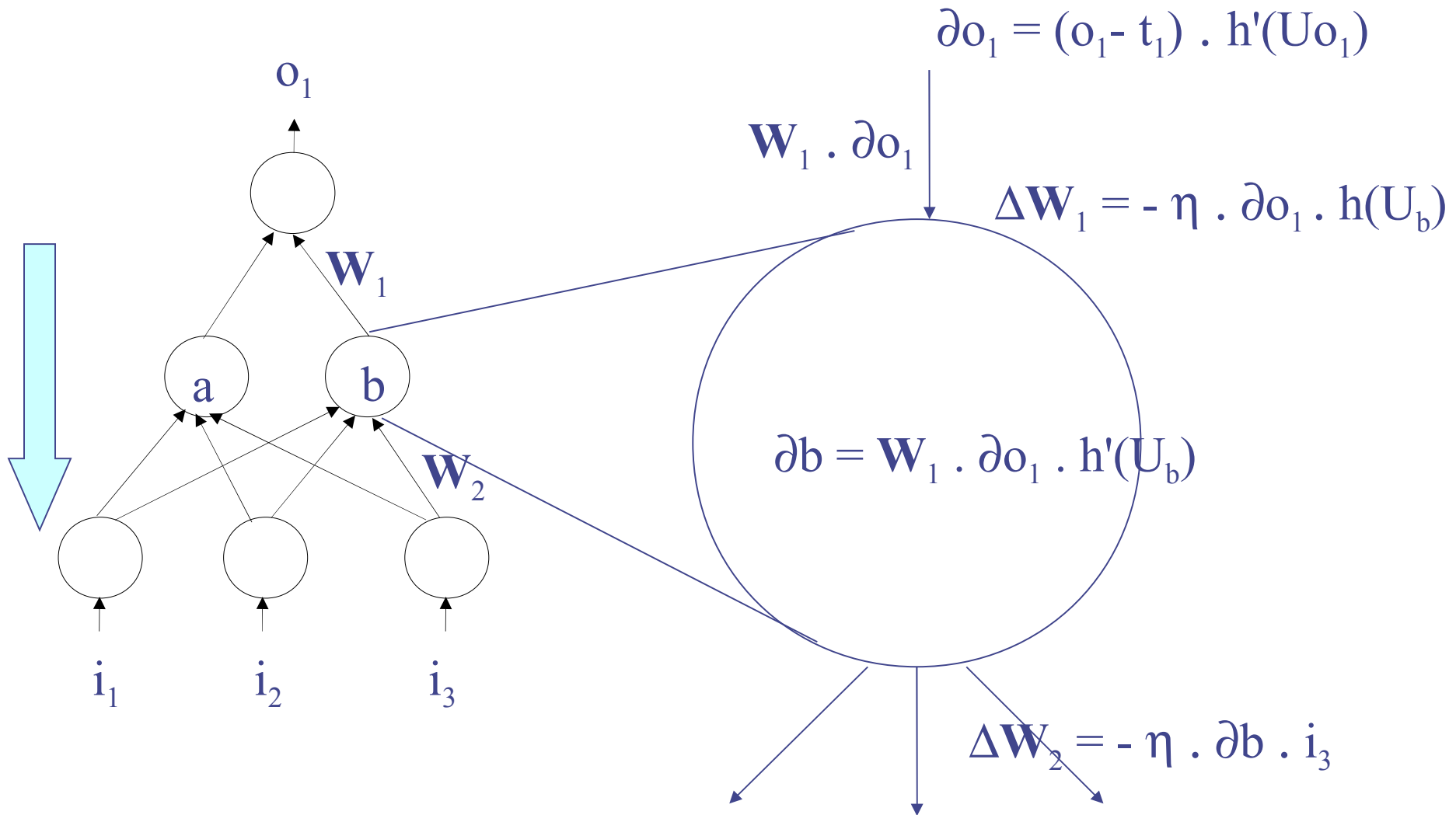
$e_1 = \text{Obtained output } (o_1) - \text{Target output } (t_1)$



## Note:

- A Training Example is a pair (input vector, target output)
- $e_1$  is the (local) error of output  $o_1$ .
- $E_w = \frac{1}{2} \sum_i (o_i - t_i)^2$  is the network's global error (which we want to minimise)

# Backpropagate Error and Change Weights



$$o_j = h(U_j), \text{ where } h(x) = 1/(1+e^{-x})$$

$$h'(U_j) = e^{-U_j} / (1 + e^{-U_j})^2$$

$$U_k = \sum_j (W_{kj} \cdot o_j) + \theta_k$$

$$o_k = h(U_k)$$

$$h'(U_k) = e^{-U_k} / (1 + e^{-U_k})^2$$

$$e_k = o_k - t_k$$

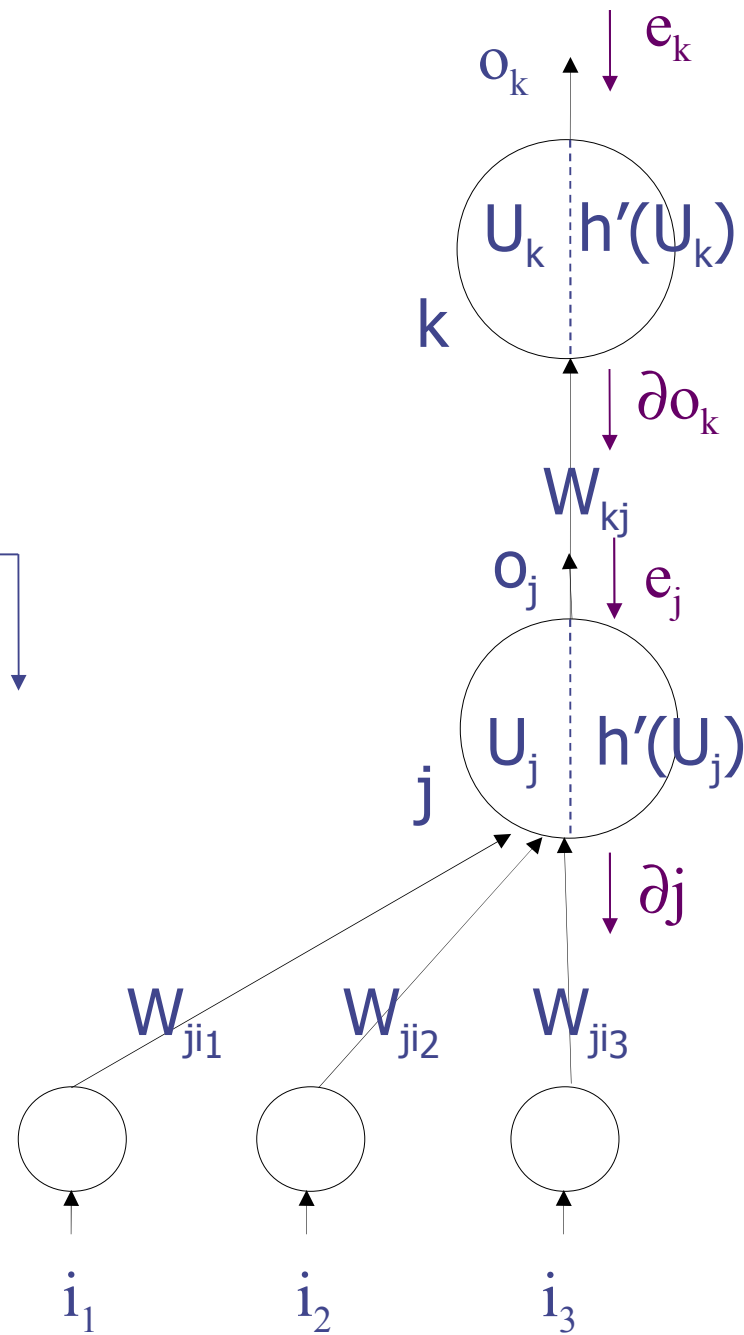
$$\partial o_k = e_k \cdot h'(U_k)$$

$$e_j = W_{kj} \cdot \partial o_k$$

$$\partial j = e_j \cdot h'(U_j)$$

$$\Delta W_{kj} = -\eta \cdot \partial o_k \cdot o_j$$

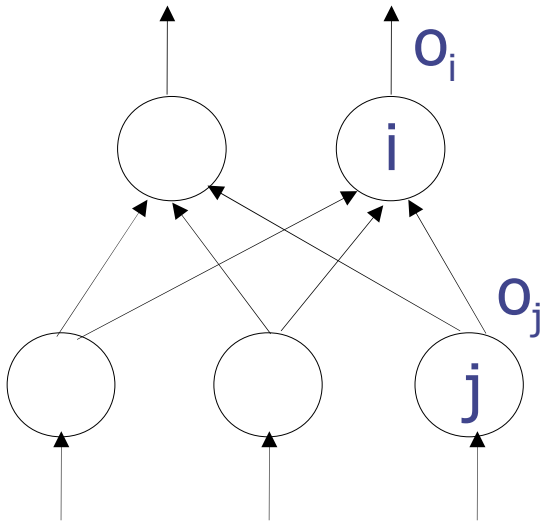
$$\Delta W_{ji} = -\eta \cdot \partial j \cdot i_i$$



# Backprop. computes $\nabla E_{\mathbf{w}} = \partial E_{\mathbf{w}} / \partial \mathbf{W}$ efficiently

$$\frac{\partial E_{\mathbf{w}}}{\partial W_{ij}} = \left( \frac{\partial E_{\mathbf{w}}}{\partial e_i} \right) \left( \frac{\partial e_i}{\partial o_i} \right) \left( \frac{\partial o_i}{\partial U_i} \right) \left( \frac{\partial U_i}{\partial W_{ij}} \right)$$

$\downarrow$   $e_i$        $\downarrow$   $1$        $\downarrow$   $h'(U_i)$        $\downarrow$   $o_j = h(U_j)$



Note:

$$E_{\mathbf{w}} = \frac{1}{2} \sum_i (o_i - t_i)^2 \quad e_i = o_i - t_i$$

$$o_i = h(U_i) \quad U_i = \sum_j (W_{ij} \cdot o_j)$$

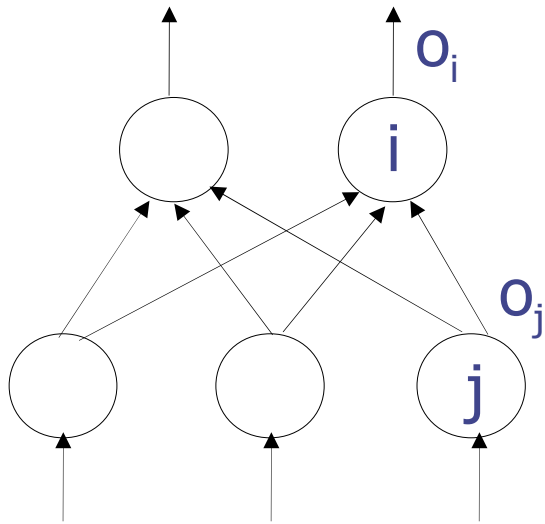
$$\partial o_i = e_i \cdot h'(U_i) \quad \Delta \mathbf{W} = -\eta \cdot \partial o_i \cdot h(U_j)$$

( $\partial o_i$  is the local gradient)

# Backprop. computes $\nabla E_{\mathbf{W}} = \partial E_{\mathbf{W}} / \partial \mathbf{W}$ efficiently

When  $i$  is an output neuron  $\Delta W = -\eta \cdot e_i \cdot h'(U_i) \cdot o_j$

What if  $i$  is a hidden neuron ( $h_i$ )?



$$\frac{\partial E_{\mathbf{W}}}{\partial W_{ij}} = \left( \frac{\partial E_{\mathbf{W}}}{\partial o_i} \right) \left( \frac{\partial o_i}{\partial U_i} \right) \left( \frac{\partial U_i}{\partial W_{ij}} \right)$$

$e_k \cdot h'_k(U_k) \cdot W_{ki}$        $h'(U_i)$        $o_j$

$$\Delta W = -\eta \cdot \partial h_i \cdot o_j$$

$$\partial h_i = h'(U_i) \sum_k (W_{ki} \cdot \partial o_k) \quad (\partial h_i, \partial o_j \text{ are local gradients})$$

# Backpropagation Algorithm

For each example ( $\mathbf{i} = i_1, \dots, i_p; \mathbf{t} = t_1, \dots, t_q$ ) in the training set, do:

For each neuron  $n$  in the network in *ascending topological order*, do:

Compute  $o_n = h(U(\mathbf{i}))$  and  $d_n = h'(U(\mathbf{i}))$

For each neuron  $n$  in the network in *descending topological order*, do:

If  $n$  is an output neuron  $o_k$  then:

$$\partial o_k = e_k \cdot d_k, \text{ where } e_k = o_k - t_k$$

$$\Delta W_{ki} = -\eta \cdot \partial o_k \cdot o_i$$

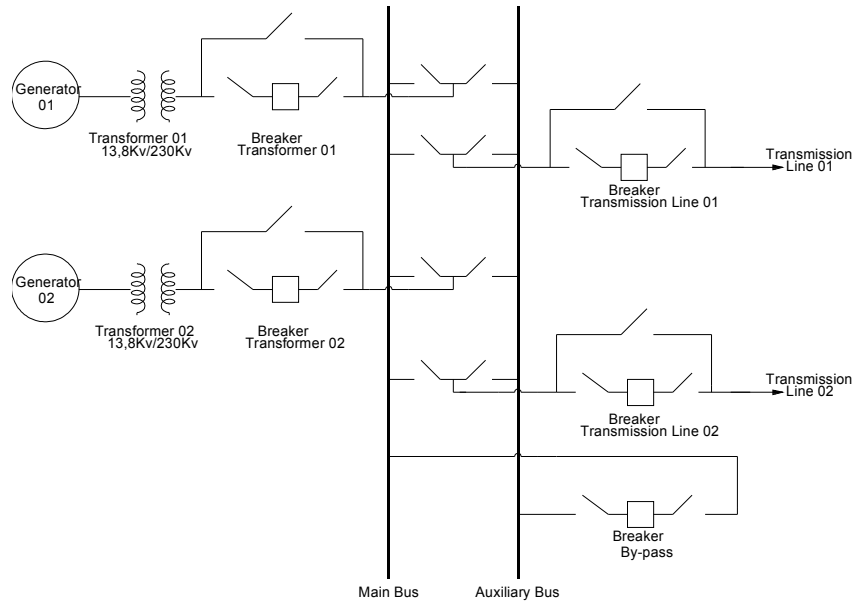
If  $n$  is a hidden neuron  $h_i$  then:

$$\partial h_i = d_i \cdot \sum_k (W_{ki} \partial o_k)$$

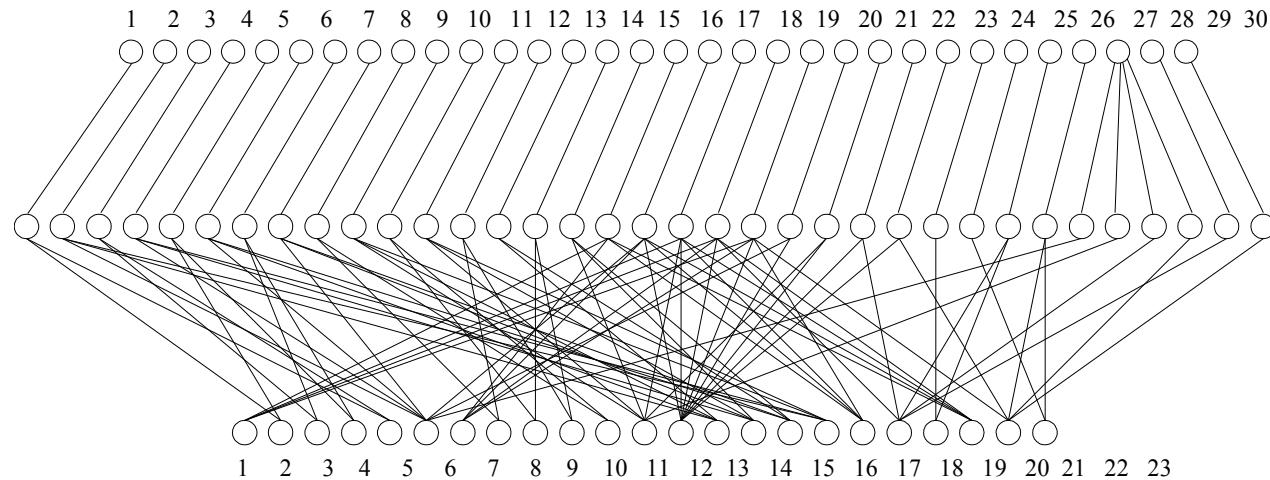
$$\Delta W_{ij} = -\eta \cdot \partial h_i \cdot o_j$$

Notes: Each pass through the training set is called an epoch.  
Typically, Backprop. takes several epochs to converge.  
The algorithm could be easily extended to networks with more than a single hidden layer.

# Application: Fault Diagnosis

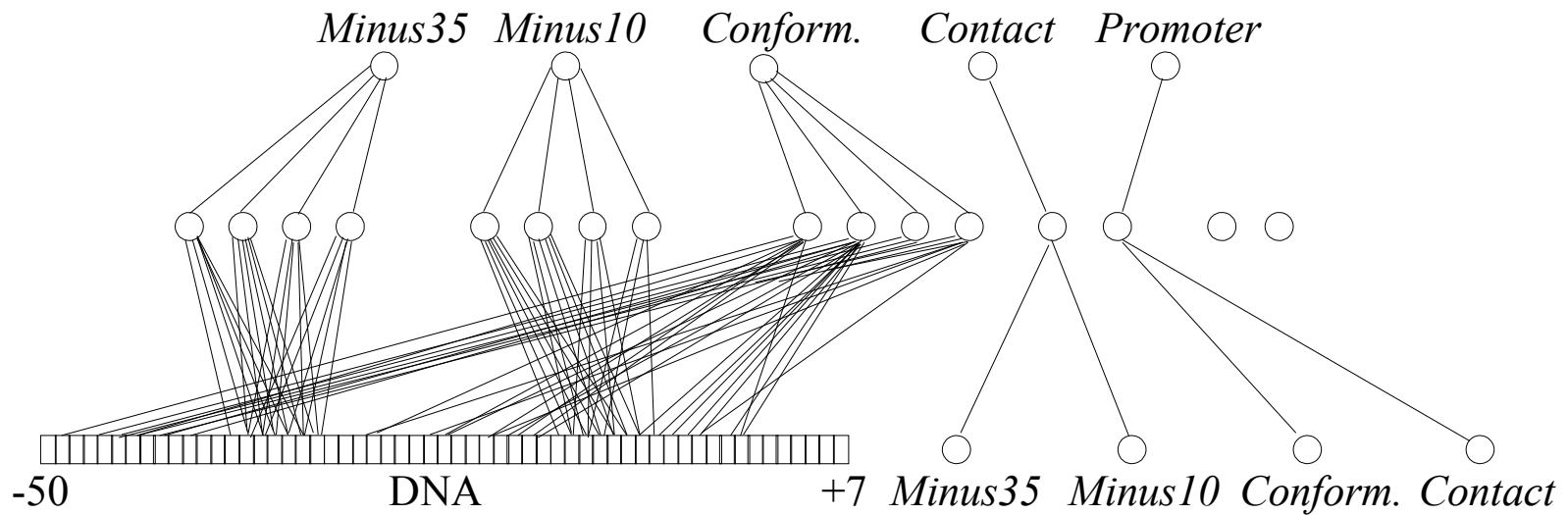


Alarms  $\Rightarrow$  Faults



# Application: DNA analysis

Promoter = small DNA sequence at beginning of genes



# Limitations of Backprop.

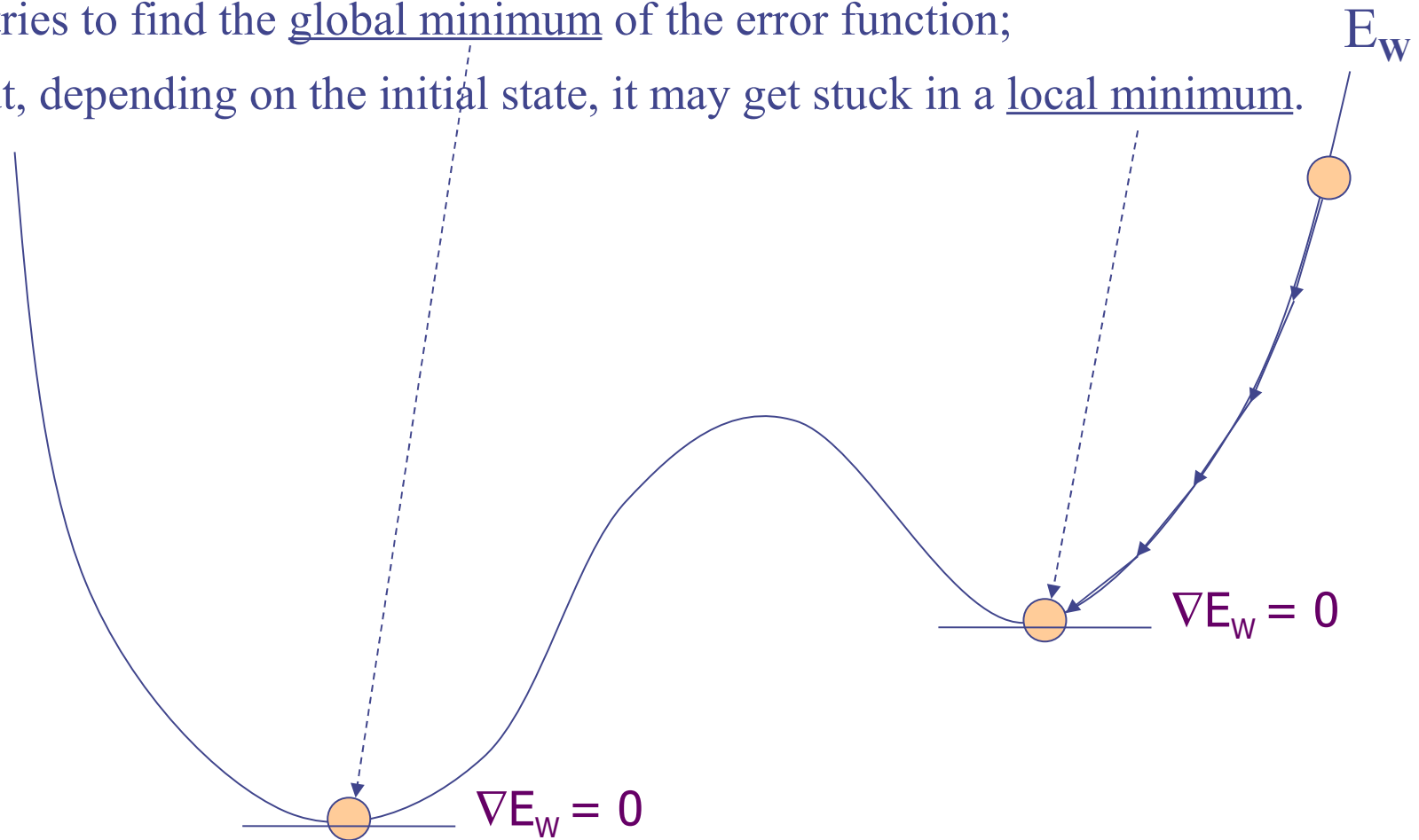
Backprop. is about trying to minimise a training set error...

Optimisation: The problem of local minima

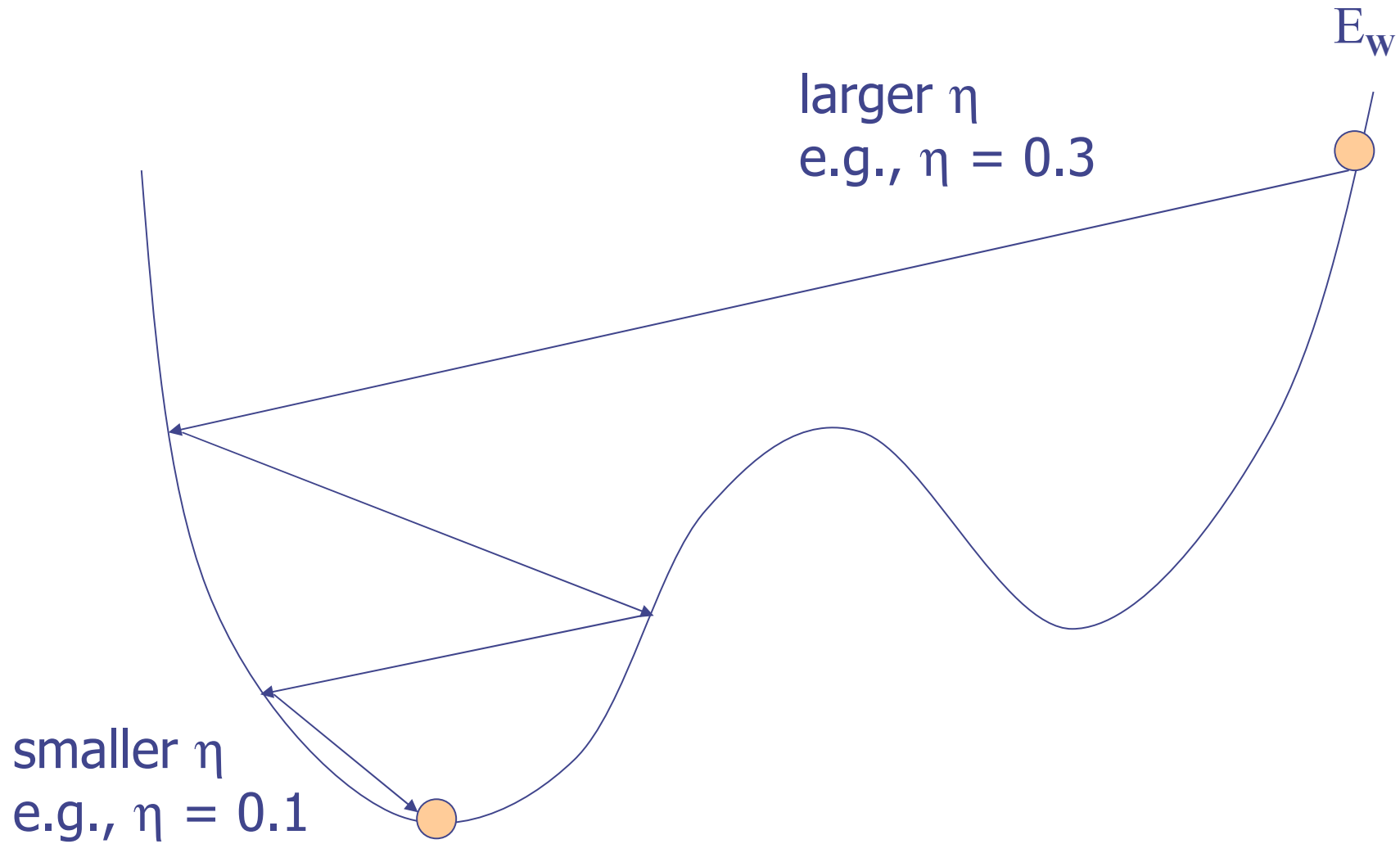
Lack of biological plausibility

# The Problem of Local Minima

- Backprop. performs gradient descent on an error surface;
- It tries to find the global minimum of the error function;
- But, depending on the initial state, it may get stuck in a local minimum.

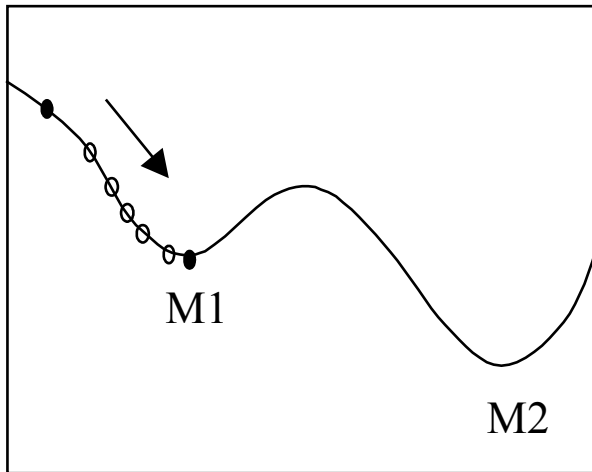


# Changing the Learning Rate

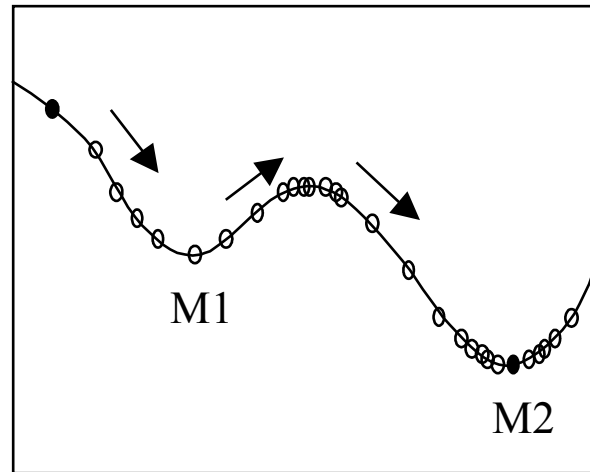


# Adding Momentum

$$\Delta W_t = -\eta \cdot \nabla E_w + \underbrace{\mu \Delta W_{t-1}}_{\text{Term of Momentum}}, \text{ where } 0 \leq \mu \leq 1$$



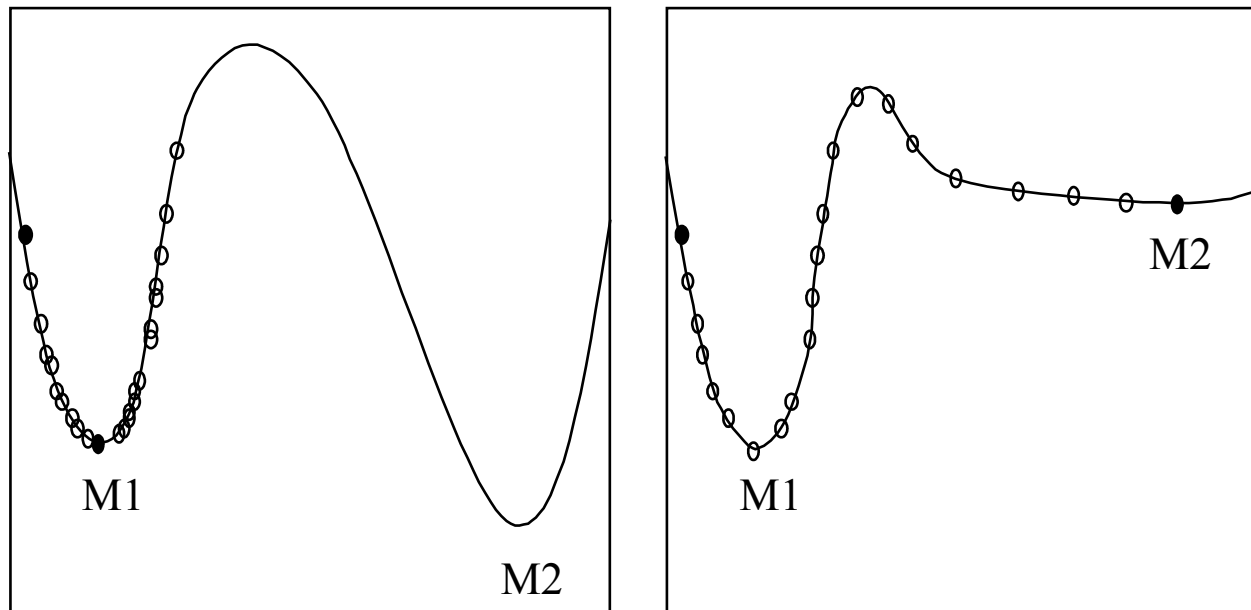
$\mu = 0.0, \eta$  small



$\mu = 0.95, \eta$  small

# Global Minimisation

Momentum does not guarantee global minimisation of  $E_w$



Note:  $E_w$  is a function in the  $n$ -dimensional space, where  $n$  is the number of output neurons

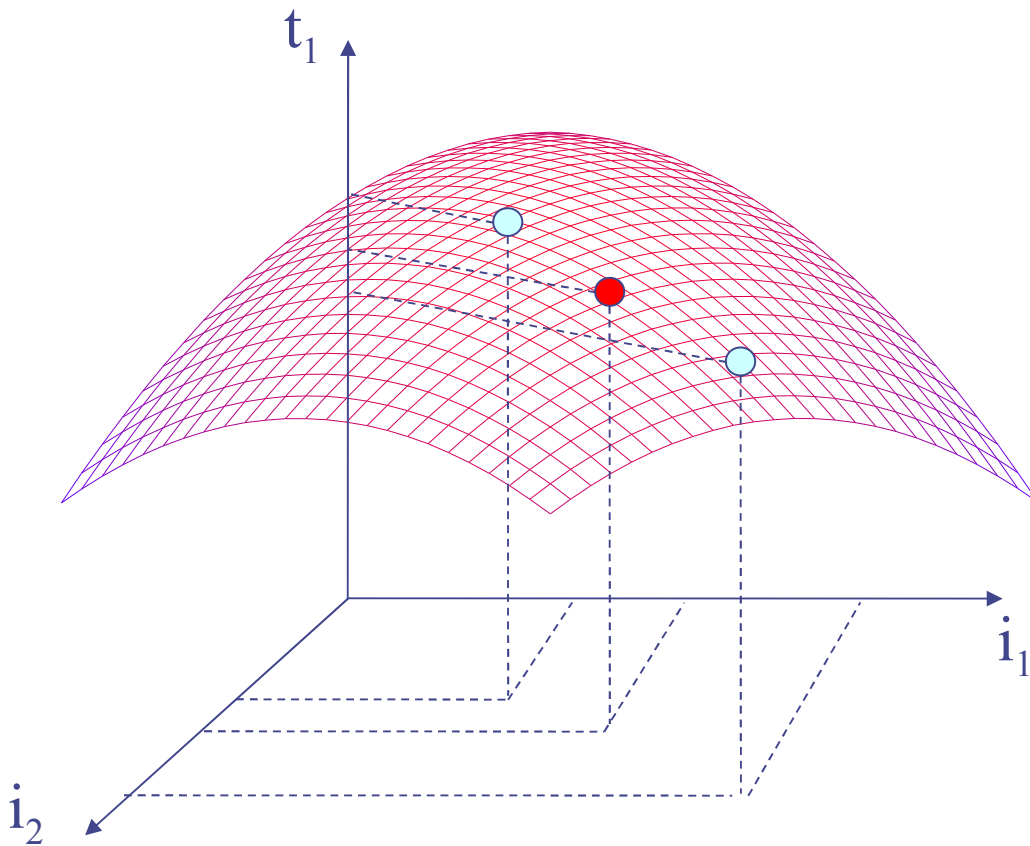
# Generalisation

- ⊕ We expect  $g$  (our approximation of  $f$ ) to produce reasonable results for examples not seen during training, i.e. we expect the learning algorithm to *generalise* to unseen cases.
- ⊕ The most likely hypothesis ( $g$ ) is the *simplest* one that is consistent with all observations (*Ockham's razor*).

# Generalisation

We try to minimise a training set error

We would like to minimise a generalisation error



○ training examples

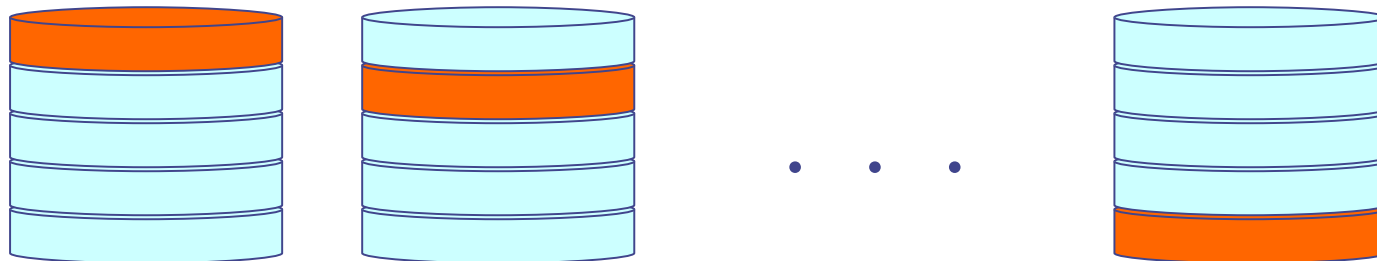
● new (unseen) case

# Estimating Generalisation Error

- Partition the set of examples into a training set and a test set.
- The test set is never seen by the network during training.
- The test set error is an estimate of the generalisation error.

# n-fold Cross-Validation

- ⊕ Divide the set of examples  $E$  into  $n$  subsets  $E_1, E_2, \dots, E_n$
- ⊕ For each  $E_i$  ( $1 \leq i \leq n$ ), train a network with  $E - E_i$  and test it with  $E_i$
- ⊕ Calculate the averaged test set error

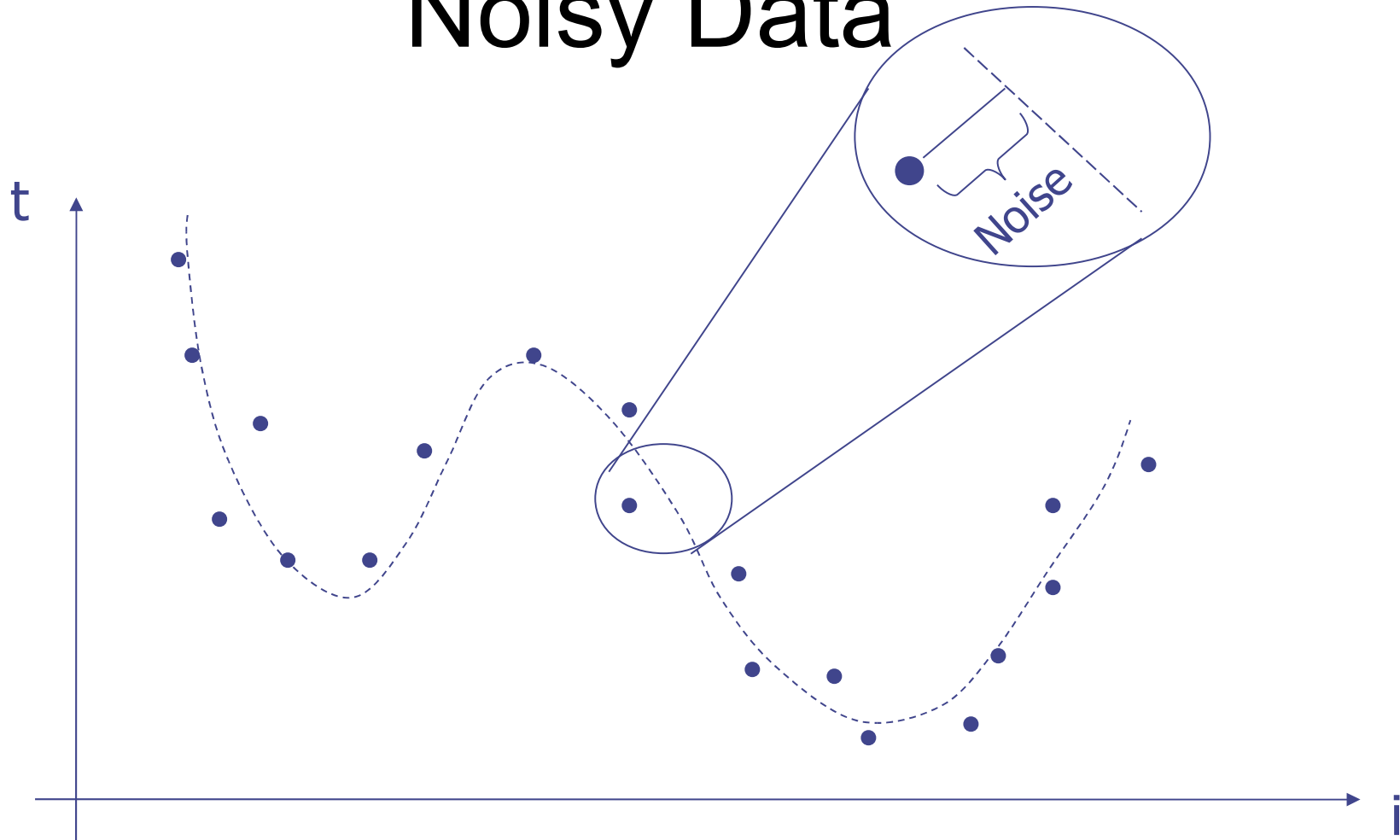


Note: Leaving  $m$  out =  $(|E| / m)$ -fold cross validation.

# Bootstrapping

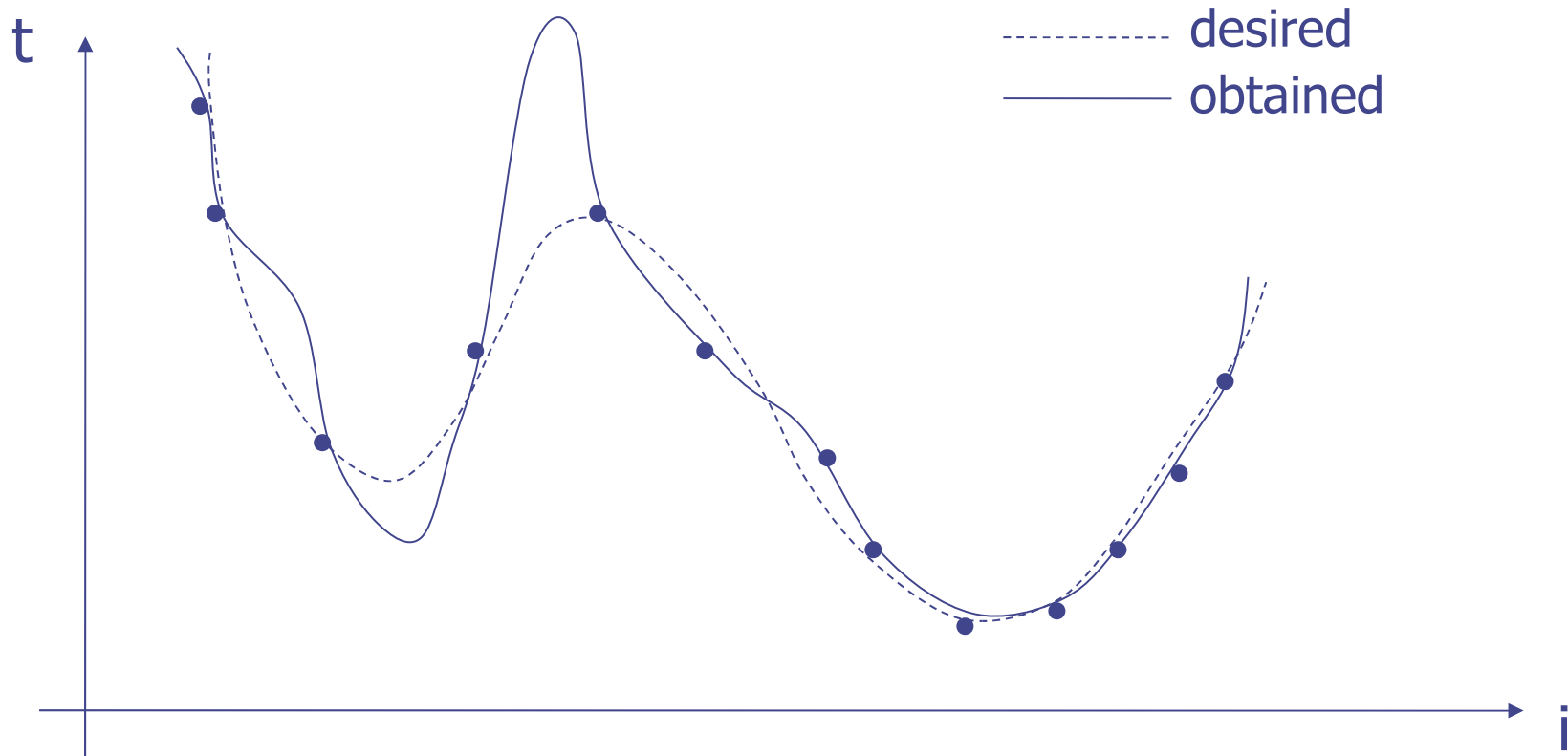
- ⊕ Create  $k$  (pseudo) sets of examples  $E_1, E_2, \dots, E_k$  by randomly selecting  $|E|$  elements (with replacement) from  $E$
- ⊕ For each  $E_i$  ( $1 \leq i \leq k$ ), train a network with  $E - E_i$  and test it with  $E_i$
- ⊕ Calculate the averaged test set error

# Noisy Data



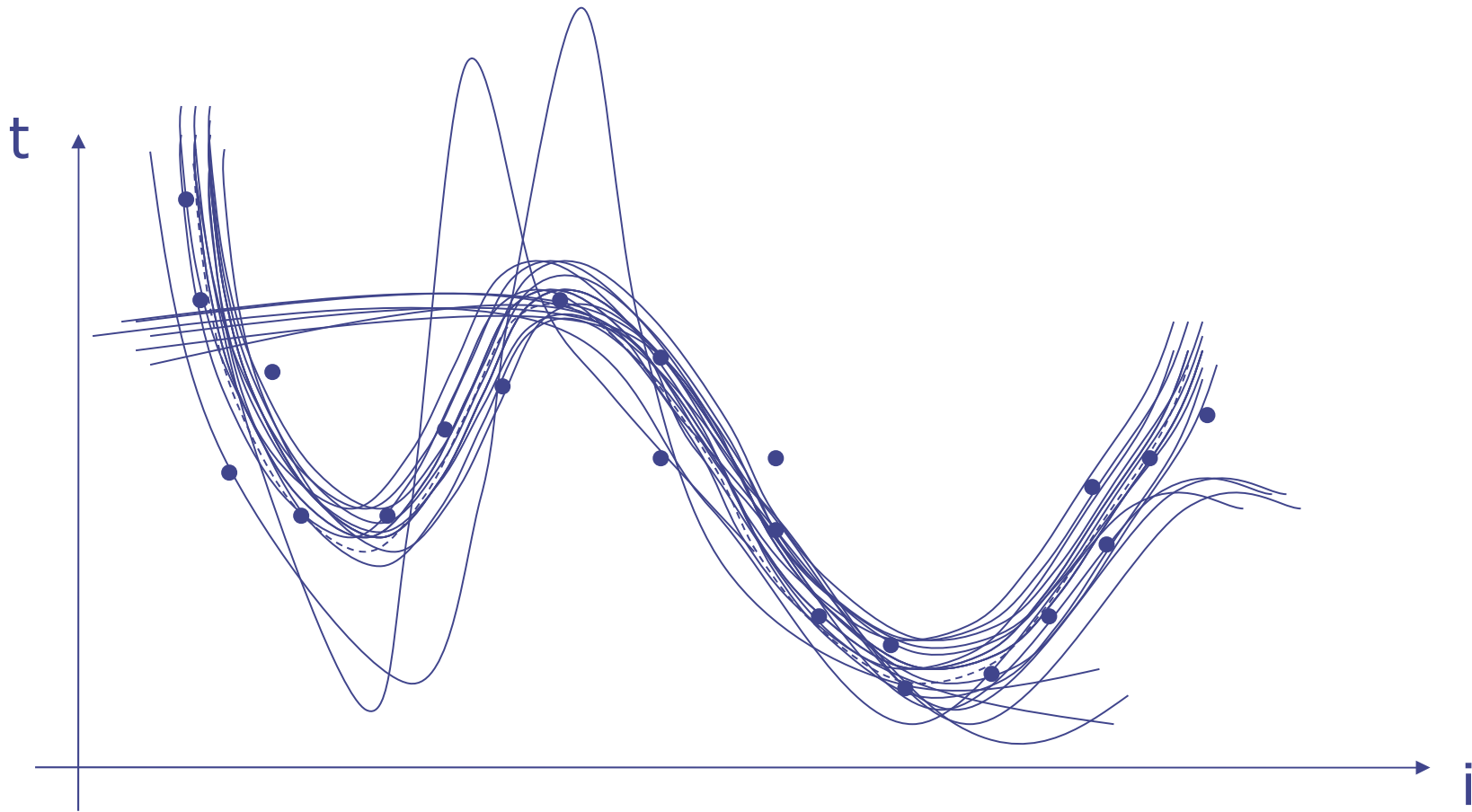
Note: If the set of training examples contains noise, the training set error must not be zero!

# Overfitting



Note: Typically caused by too many hidden neurons or not enough training examples, overfitting results in good training set performance and poor test set performance.

# Testing the Network is Key!



# Learning and Generalisation

Backprop. is about trying to minimise a training set error...

Problem of local minima: add momentum, vary learning rate

...but learning is about trying to minimise a generalisation error

Estimate using test set error, cross-validation, bootstrapping  
(Note: generalisation error approaches training set error when the number of training examples grows)

In addition, in the presence of noisy training examples or to check for overfitting, testing the network is key!