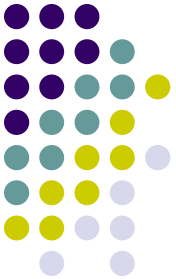
Birkbeck
UNIVERSITY OF LONDON



(Concepts of) Machine Learning

Lecture 4: Neural networks and deep learning

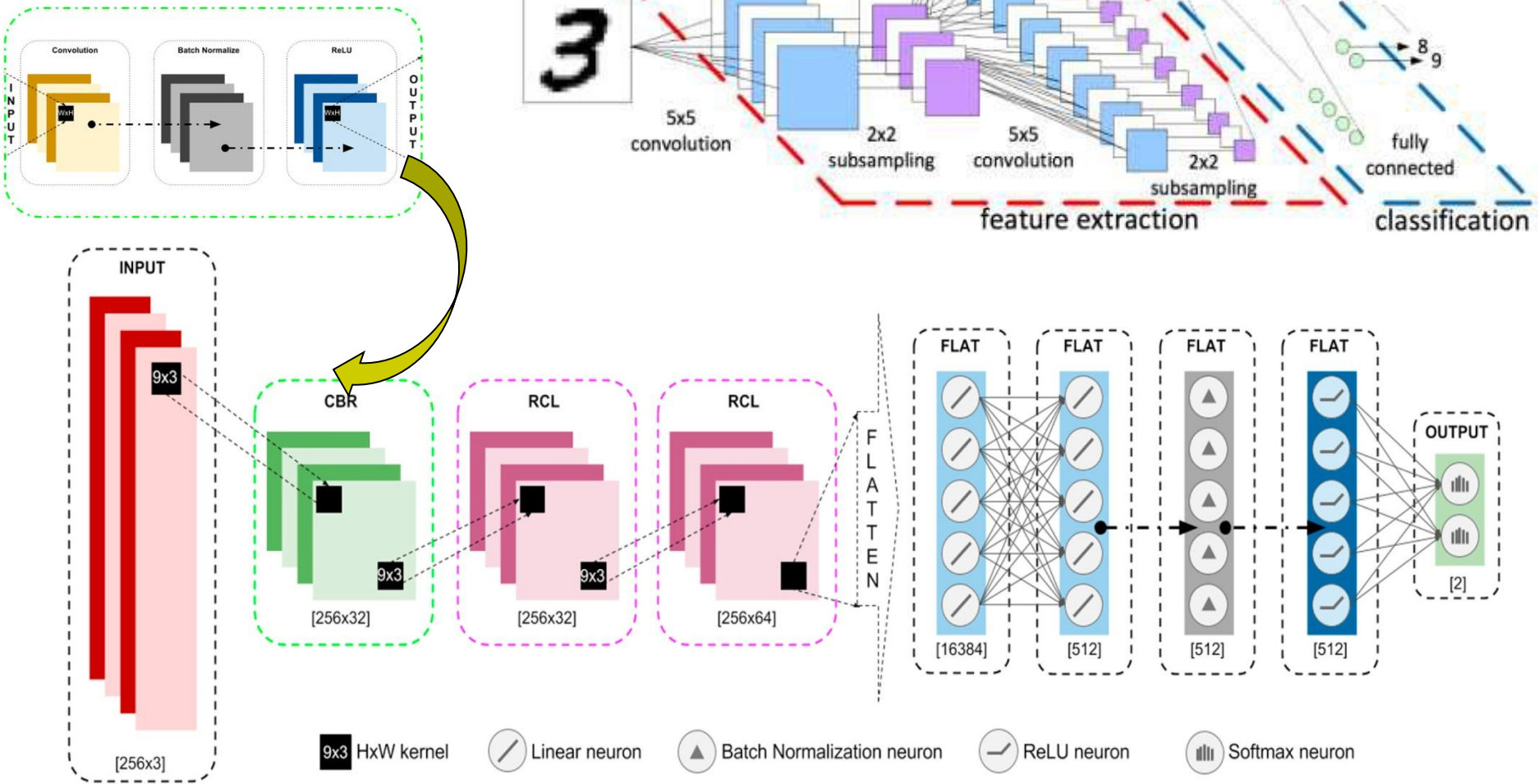
George Magoulas
gmagoulas@dcs.bbk.ac.uk

Contents

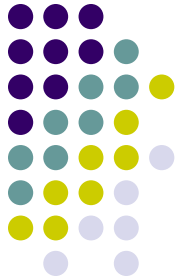
- Neural networks and deep learning
- Multilayer Networks and back-propagation
- Backpropagation variants for training
- Recurrent Networks
- Backpropagation through time

What is a neural network?

Neural networks and deep learning



Neural networks and deep learning



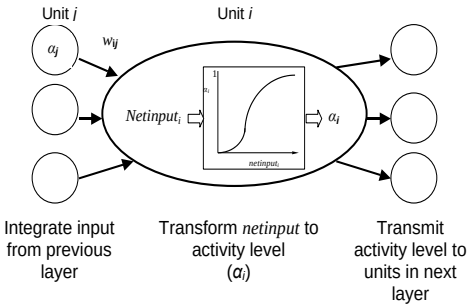
1980s-1990s: Artificial neural networks (ANNs) with 2 or 3 hidden layers (or more) and with many units in the hidden layers. Researchers couldn't train larger and deeper networks on the serial, single-core computers used at the time. Now these are called "shallow" neural networks.

Geoff Hinton moves from CMU to Uni Toronto. Uses the term **"deep network"** in the papers
Reducing the Dimensionality of Data with Neural Networks, 28 JULY 2006, Vol. 313 SCIENCE
Learning multiple layers of representation, TRENDS in Cognitive Sciences Vol.11 No.10, 2007

Collaboration of Ng's group at Stanford with Google. Use the term **"deep learning"** in the paper
Building High-level Features Using Large Scale Unsupervised Learning, Proceedings of the 29 h International Conference on Machine Learning, Edinburgh, Scotland, UK, 2012

Neural networks and deep

Learning by weight change: If the response of an output unit is incorrect then the network can be changed so that it is more likely to produce the correct response the next time that the stimulus is presented. This is achieved by changing the connection weights.



$$w_{ij}^{new} = w_{ij}^{old} + \Delta w_{ij}$$

$$\Delta w_{ij} = -\eta \frac{dE}{dw_{ij}} = -\eta \frac{d\left[a_i^{desired} - a_i^{obtained}\right]^2}{dw_{ij}}$$

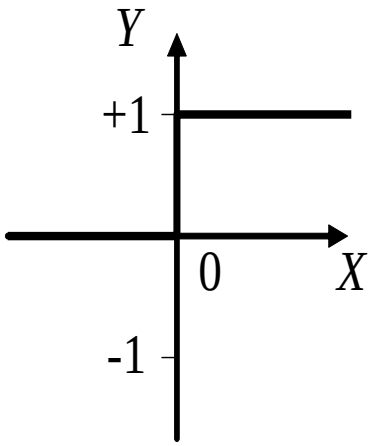
Δw_{ij} is the change in the connection weight w_{ij} from unit *j* to unit *i*

Bias: There is one special input unit, which is called bias unit. The bias unit receives no input itself, and its activity is always set at +1. The weight from the bias unit to the unit of interest can be positive or negative and changes just like any other weight during learning.

Neural networks and deep learning

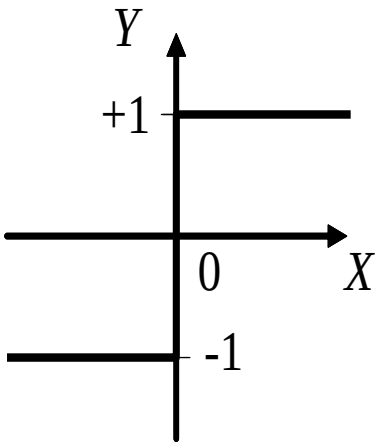
Activation functions of a neuron

Step function



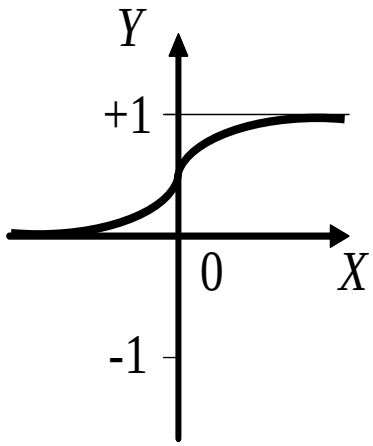
$$Y^{step} = \begin{cases} 1, \text{ if } X \geq 0 \\ 0, \text{ if } X < 0 \end{cases}$$

Sign function



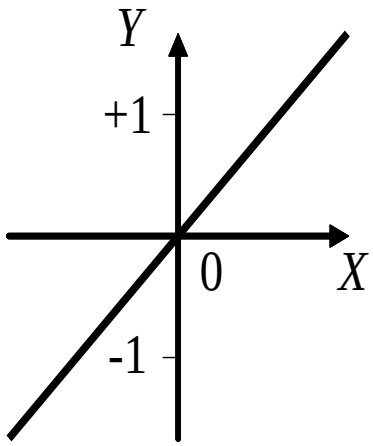
$$Y^{sign} = \begin{cases} +1, \text{if } X \geq 0 \\ -1, \text{if } X < 0 \end{cases}$$

Sigmoid function



$$Y^{sigmoid} = \frac{1}{1+e^{-X}}$$

Linear function

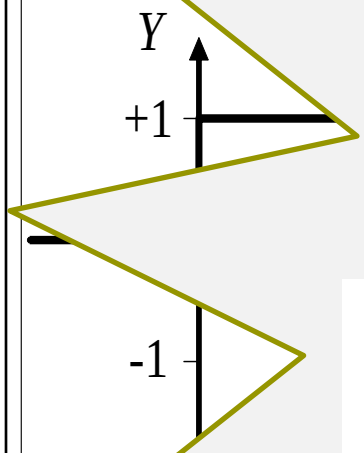


$$Y^{linear} = X$$

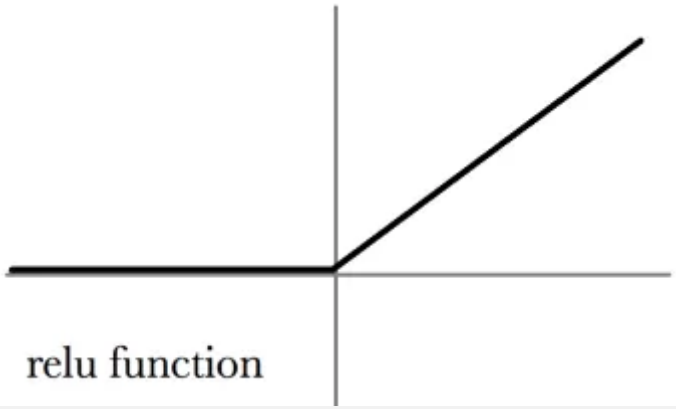
Neural networks and deep learning

Activation fun s of n

Step function



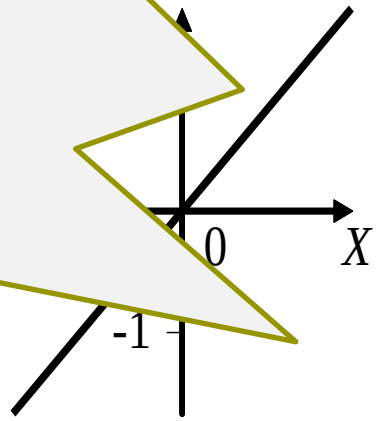
$$Y^{step} = \begin{cases} 1, \text{ if } X \geq 0 \\ 0, \text{ if } X < 0 \end{cases}$$



$$RELU(x) = \begin{cases} 0 \; if \; x < 0 \\ x \; if \; x >= 0 \end{cases}$$

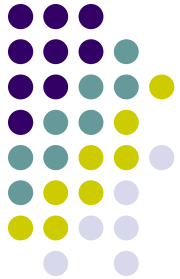
$$= \begin{cases} +1, \text{ if } X \geq 0 \\ -1, \text{ if } X < 0 \end{cases}$$

$$Y^{sigmoid} = \frac{1}{1+e^{-X}}$$



$$Y^{linear} = X$$

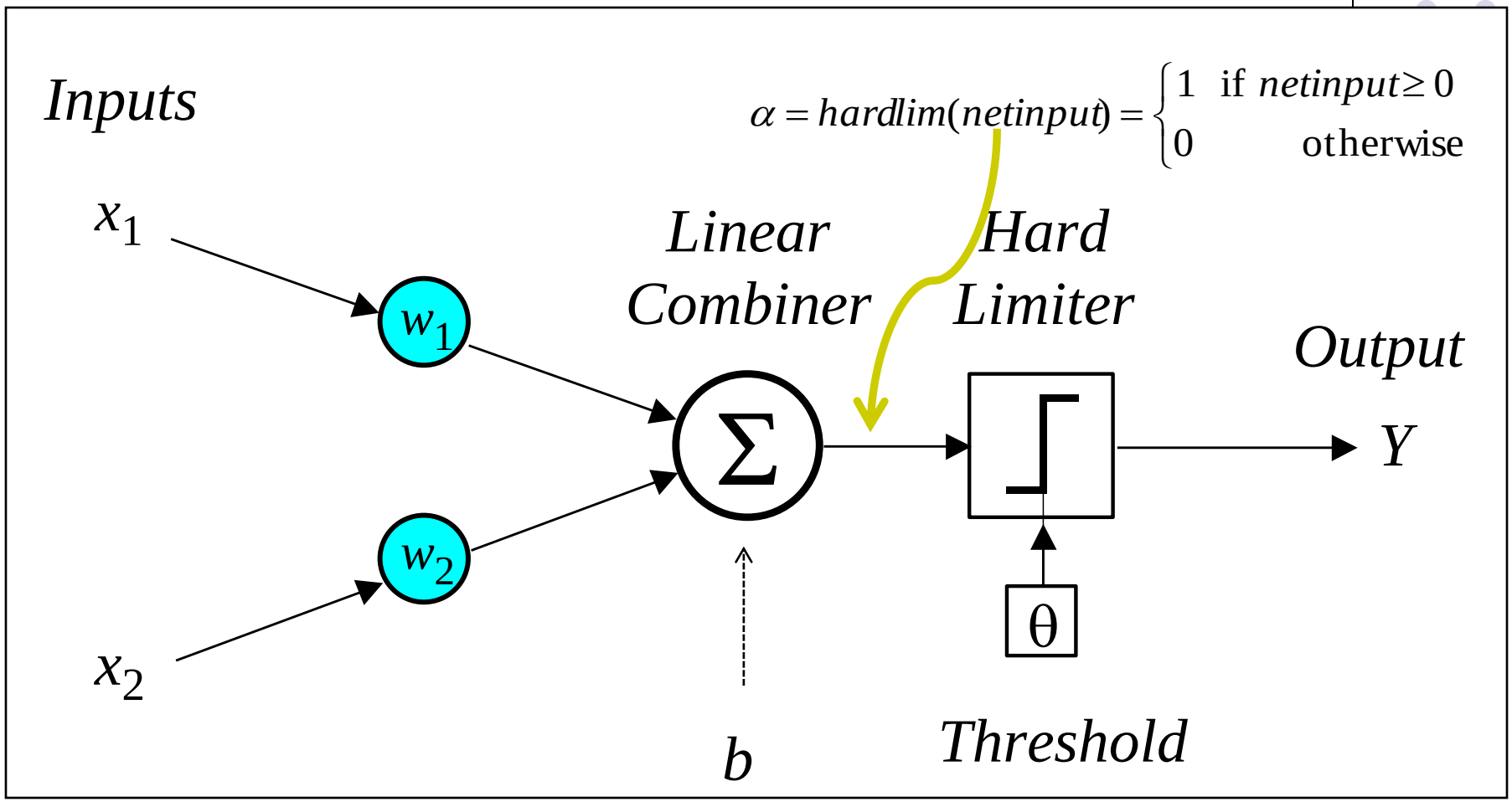
Multilayer Networks and back-propagation



The perceptron and the perceptron rule: can a single neuron learn a task?

- In 1958, **Frank Rosenblatt** introduced a training algorithm that provided the first procedure for training a simple ANN: a **perceptron**.
- The perceptron is the simplest form of a neural network. It consists of a single neuron with *adjustable* synaptic weights and a *hard limiter*.

Single-layer two-input perceptron



The node divides the input space into two regions because it can only be in one of two states (i.e. 1 or 0)

(1) Assume a node with only two inputs : $w_{11}x_1 + w_{12}x_2 = 0$

(2) Assume the node has a bias term *b:* $w_{11}x_1 + w_{12}x_2 + b = 0$

(3) Assume w_{11}=1; w_{12}=1; b=-1

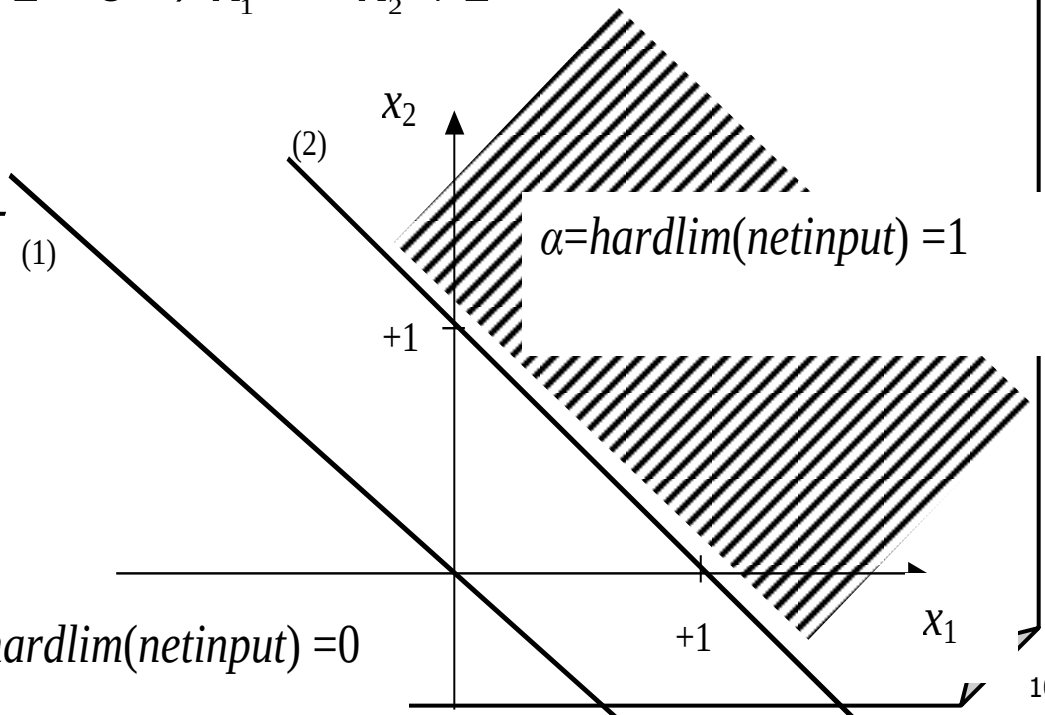
Then:

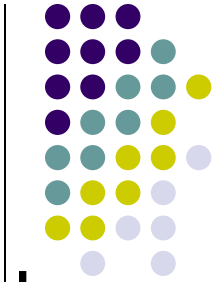
These are line equations→

$$x_1 + x_2 = 0 \Rightarrow x_1 = -x_2$$ (1) No bias

$$x_1 + x_2 - 1 = 0 \Rightarrow x_1 = -x_2 + 1$$ (2) With a bias

The line defines the boundary between regions where the input pattern produces a positive response (output) and regions where the response will be negative or zero. A line of this kind is also called *decision boundary*

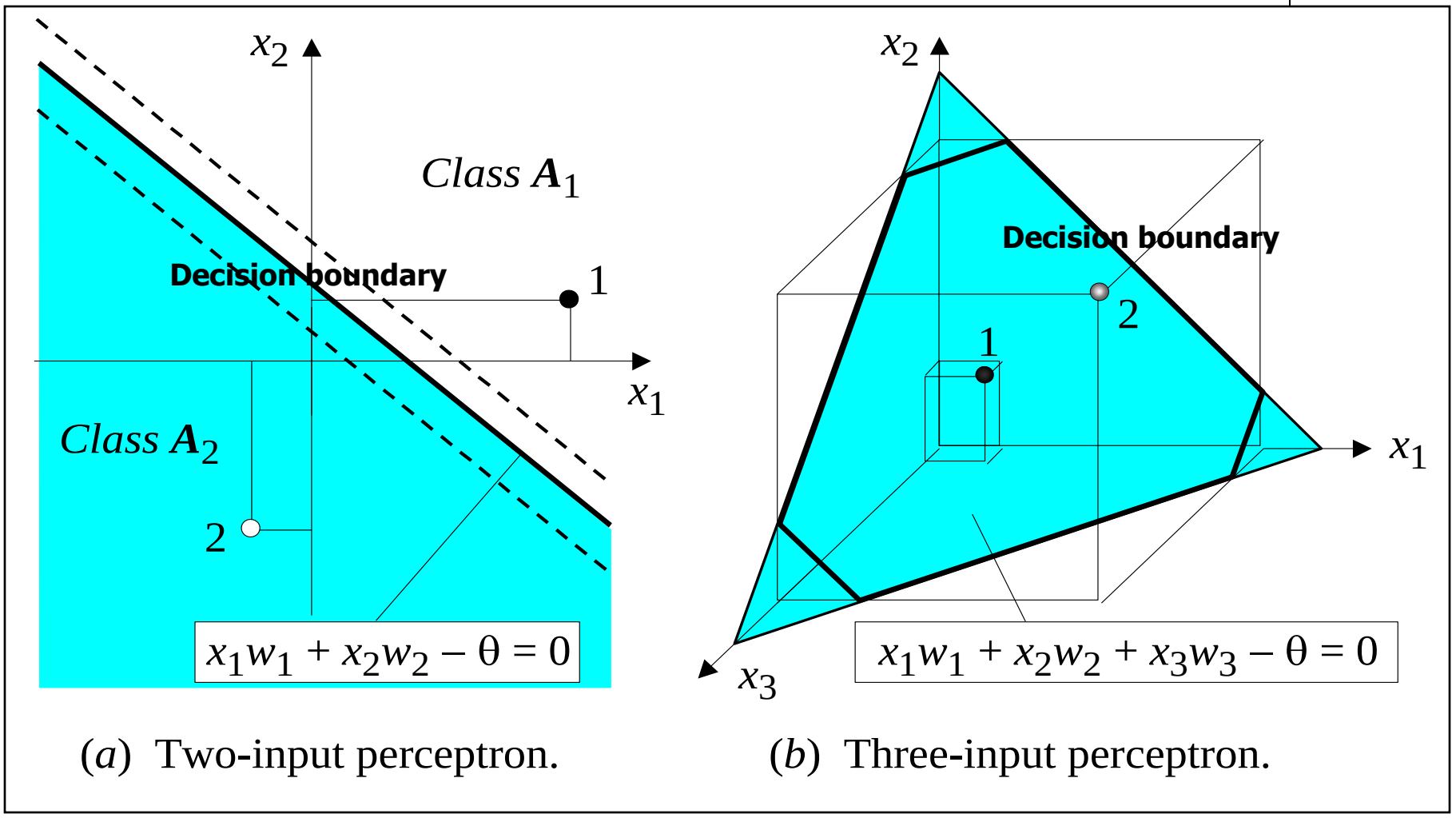




- The aim of the perceptron is to classify inputs, $x_1, x_2, \ldots, x_n$, into one of two classes, say $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$.
- For an elementary perceptron, the n-dimensional space is divided by a *hyperplane* into two regions en.wikipedia.org/wiki/Hyperplane)
- The hyperplane is defined by the ***linearly separable* function**:

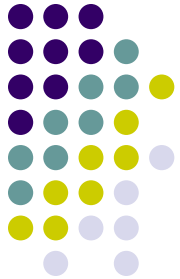
$$\sum_{i=1}^{n} x_i w_i - \theta = 0$$

Linear separability in the perceptrons



(*a*) Two-input perceptron. (*b*) Three-input perceptron.

How does the perceptron learn its classification tasks?



This is done by making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron. The initial weights are randomly assigned, usually in the range [–0.5, 0.5], and then updated to obtain the output consistent with the training examples.

- If at iteration p, the actual output is $Y(p)$ and the desired output is $Y_d(p)$, then the error is given by:

$$e(p) = Y_d(p) - Y(p)$$ where $p = 1, 2, 3, \ldots$

Iteration p here refers to the pth training example presented to the perceptron.

- If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$.

The perceptron learning rule

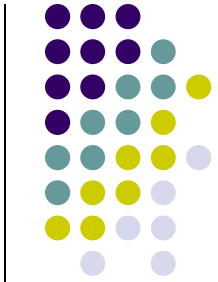
$$w_i(p+1) = w_i(p) + \alpha \cdot x_i(p) \cdot e(p)$$

where $p = 1, 2, 3, \ldots$

α is the **learning rate**, a positive constant less than unity.

The perceptron learning rule was first proposed by **Rosenblatt** in 1960. Using this rule we can derive the perceptron training algorithm for classification tasks.

Perceptron's training algorithm

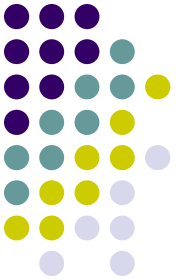


Step 1: Initialisation

Set initial weights w_1, w_2,..., w_n and threshold θ to random numbers in the range [–0.5, 0.5].

If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$.

Perceptron's training algorithm (continued)



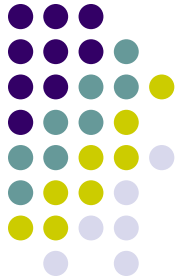
Step 2: Activation

Activate the perceptron by applying inputs $x_1(p)$, $x_2(p)$,..., $x_n(p)$ and desired output $Y_d(p)$.
Calculate the actual output at iteration $p = 1$

$$Y(p) = step\left[\sum_{i=1}^{n} x_i(p)\, w_i(p) - \theta\right]$$

where n is the number of the perceptron inputs, and *step* is a step activation function.

Perceptron's training algorithm (continued)



Step 3: Weight training

Update the weights of the perceptron

$$w_i(p+1) = w_i(p) + \Delta w_i(p)$$

where $\Delta w_i(p)$ is the weight correction at iteration p.

The weight correction is computed by the **delta rule**:

$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)$$

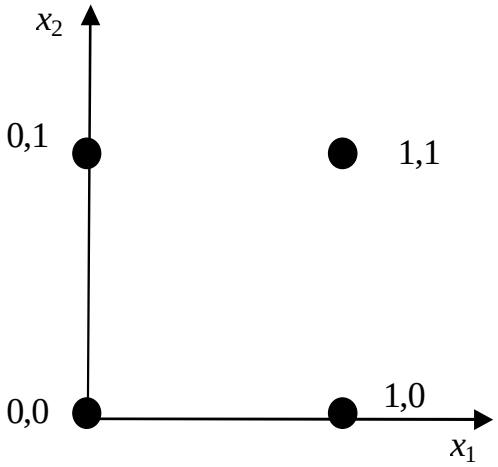
Step 4: Iteration

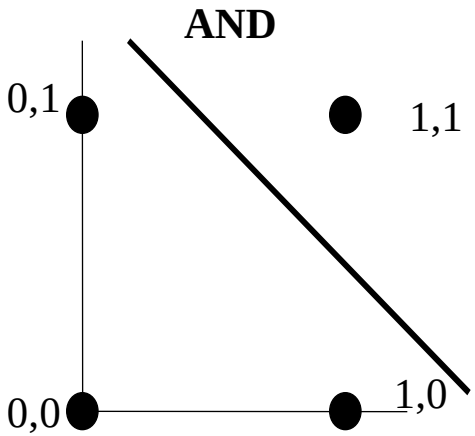
Increase iteration p by one, go back to *Step 2* and repeat the process until convergence.

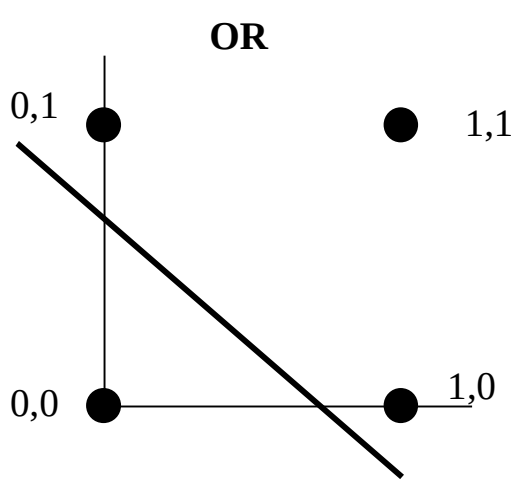
Perceptron and the perceptron rule (the limitations)

Linear separable problems:
Learning Boolean functions

Input		Output	
x_1	x_2	AND	OR
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1







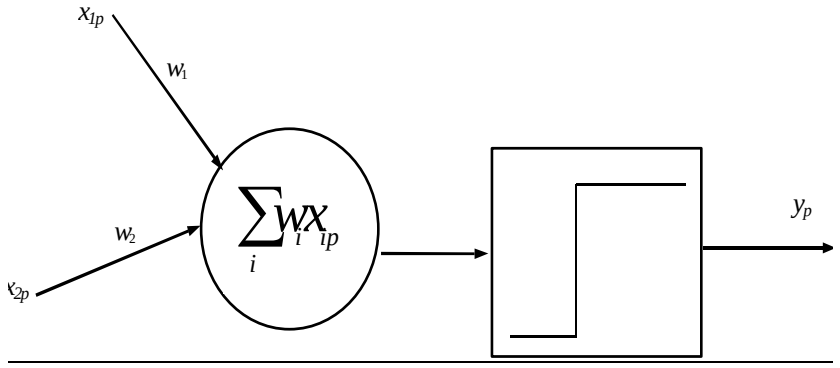
Numeric example: A single node with two weights

A single node with two weights is trained using the following rule:

$$w_i^{t+1} = w_i^t + \sum_{p=1}^{P} x_{ip} y_p, \quad i = 1,2,$$

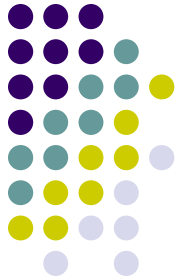
where t denotes iterations, P is the number of patterns in the training set and the inputs, x_{ip}, and outputs, y_p, for each pattern, p=1,…,4, are shown in the following truth table

p	x_{1p}	x_{2p}	y_p
1	0	0	0
2	0	1	0
3	1	0	0
4	1	1	1



The weights are all zero at the start, i.e. $w_i^0 = 0$, for all $i = 1,2$. The node is trained by presenting it with input and output pairs in the same order as the truth table, starting form the top, then the first input pattern that is used (p=1) is [0, 0] and the corresponding output pattern is 0.

Numeric example: A single node with two weights



What would the weights be after the presentation of each pattern?

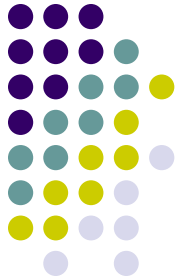
p=1: w_1=0+x_1y=0; w_2=0+x_2y=0

p=2: w_1=0+x_1y=0; w_2=0+x_2y=0

p=3: w_1=0+x_1y=0; w_2=0+x_2y=0

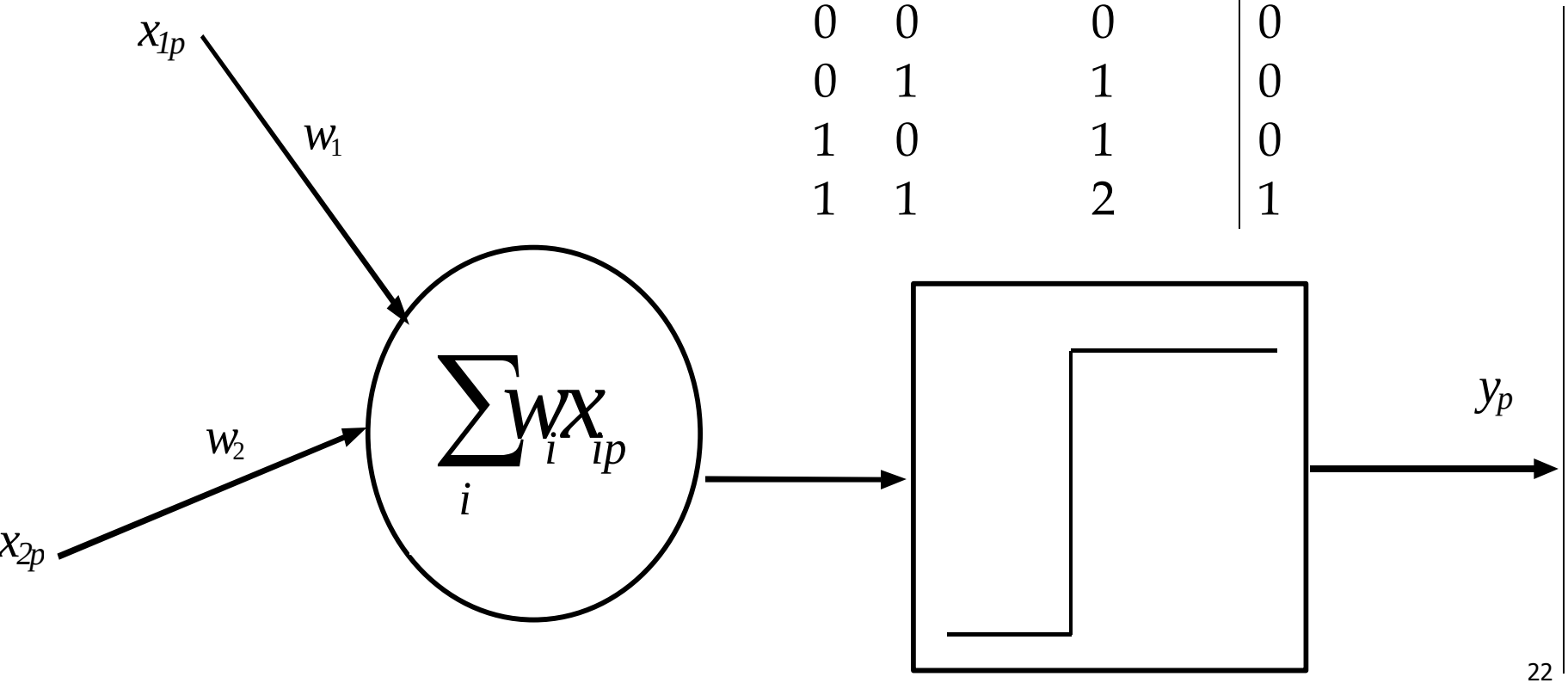
p=4: w_1=0+x_1y=1; w_2=0+x_2y=1

Numeric example: A single node with two weights



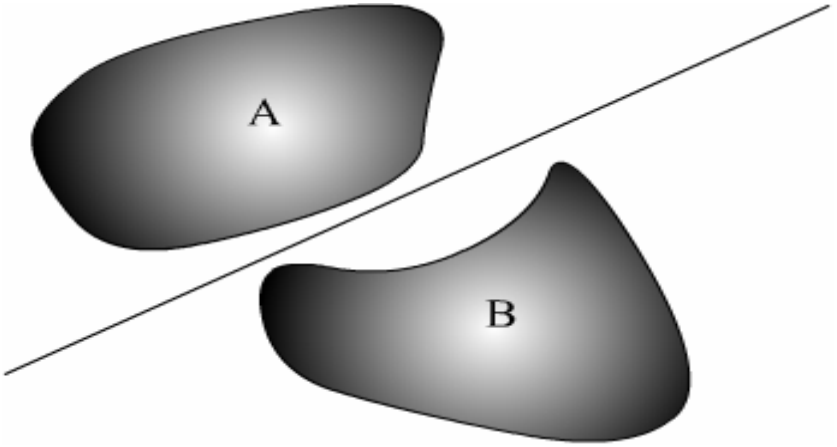
What output values are produced with a threshold of 1.1?

x_1	x_2	$w_1x_1 + w_2x_2$	y
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

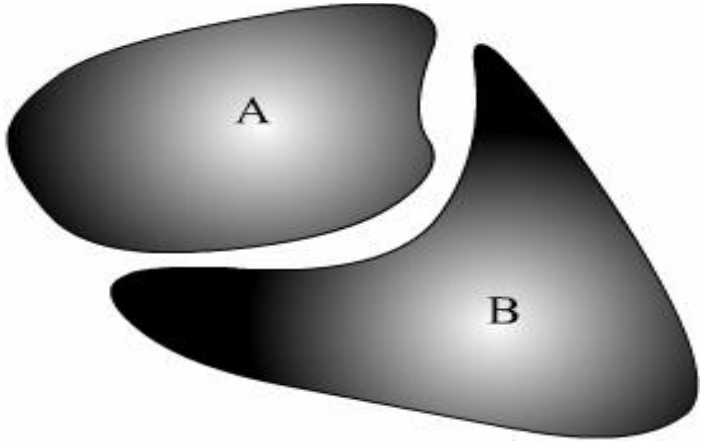


Multilayer networks and backpropagation

Non linear separable problems: Training patterns belonging to one output class cannot be separated from training patterns belonging to another class by a straight line, plane or hyperplane.

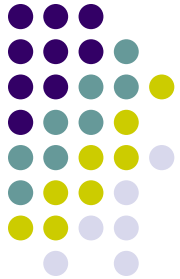


Linear separable



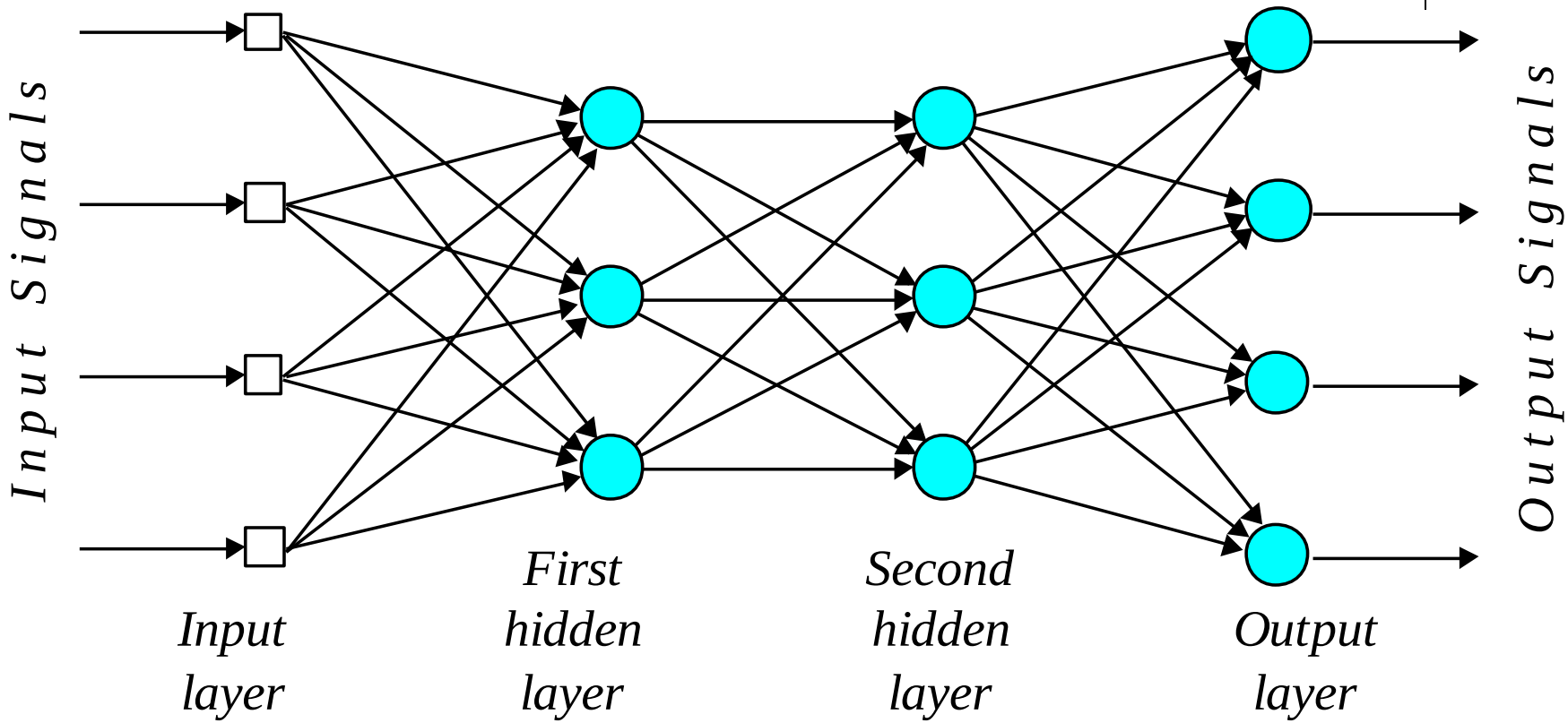
Nonlinear separable

Multilayer neural networks

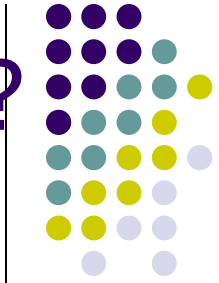


- A multilayer perceptron is a feedforward neural network with one or more hidden layers.
- The network consists of an **input layer** of source neurons, at least one middle or **hidden layer** of computational neurons, and an **output layer** of computational neurons.
- The input signals are propagated in a forward direction on a layer-by-layer basis.

Multilayer perceptron with two (or more) hidden layers

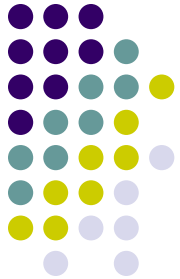


What does the middle layers hide?



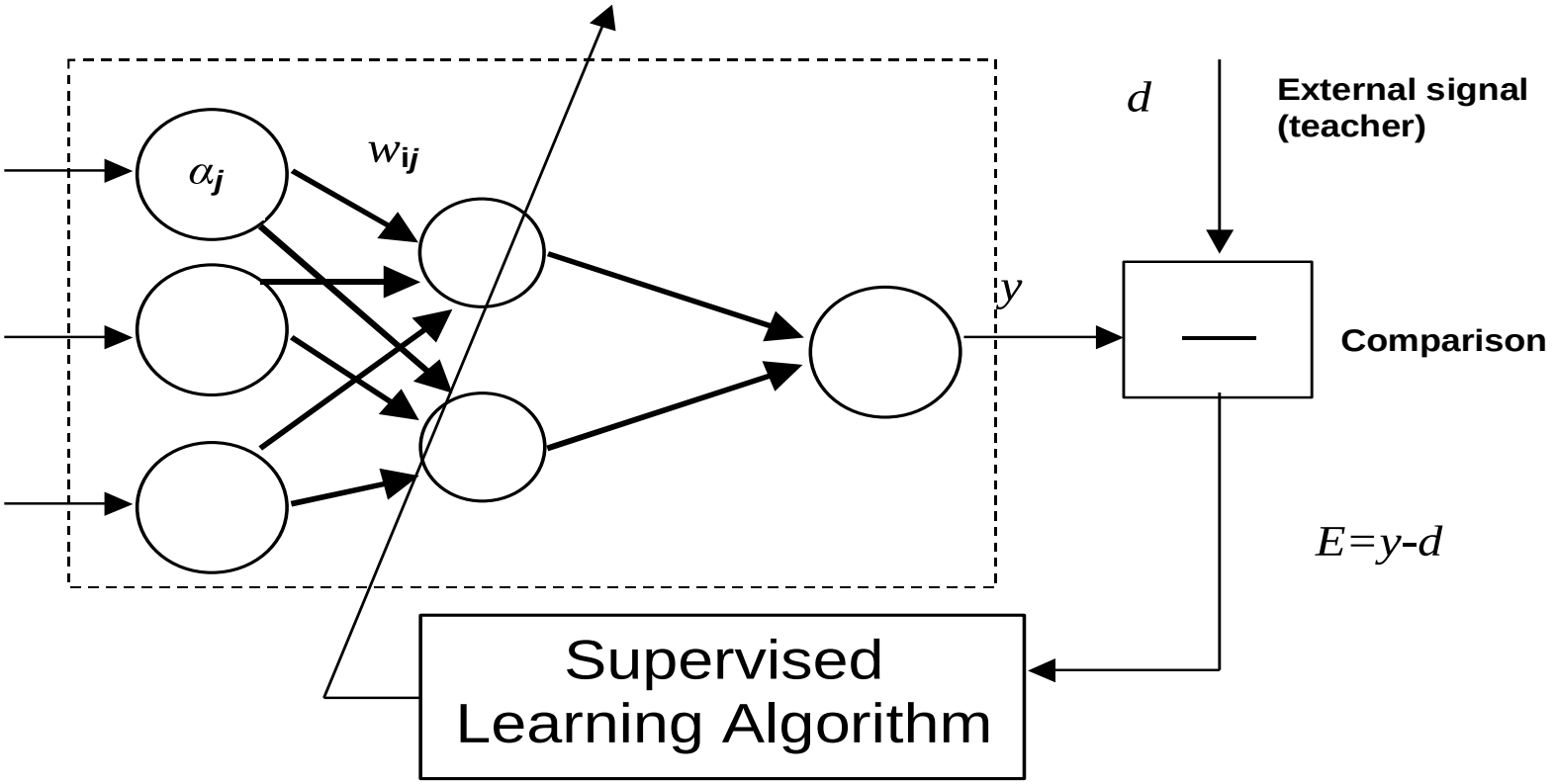
- A hidden layer "hides" its desired output. Neurons in the hidden layer cannot be observed through the input/output behaviour of the network. There is no obvious way to know what the desired output of the hidden layer should be.
- Shallow ANNs incorporate three and sometimes four layers, including one or two hidden layers. Each layer can contain from 10 to 1000 neurons. Deep ANNs may have five or more layers and utilise millions of neurons.

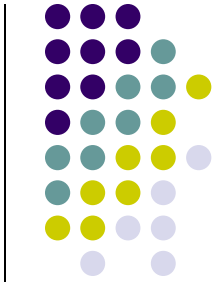
Back-propagation neural network



- Learning in a multilayer network proceeds the same way as for a perceptron.
- A training set of input patterns is presented to the network.
- The network computes its output pattern, and if there is an error – or in other words a difference between actual and desired output patterns – the weights are adjusted to reduce this error.

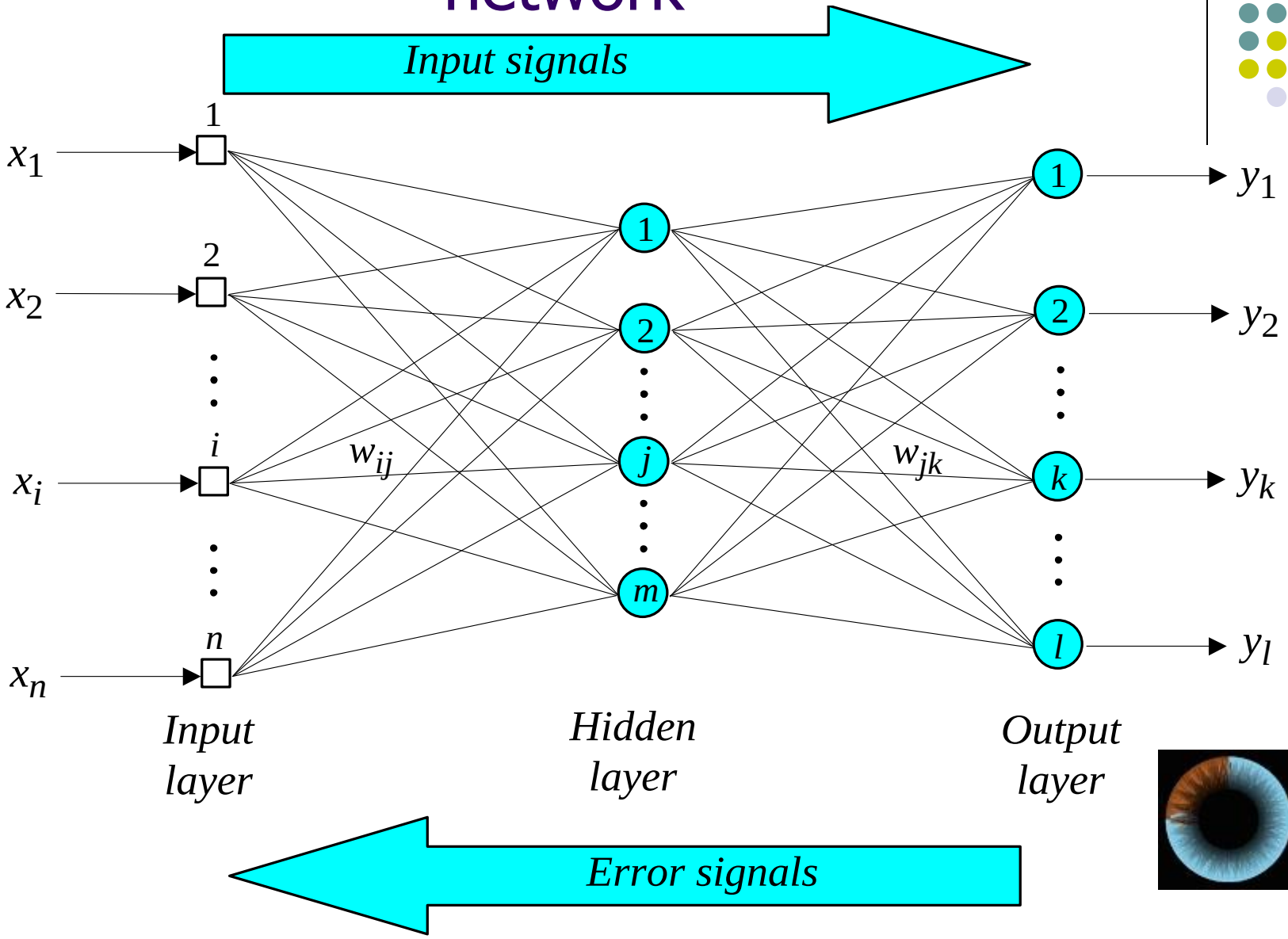
This form of training is called **Supervised learning:** The response that the backpropagation network is required to learn is presented to the network during training. The desired response of the network acts as an explicit teacher signal.



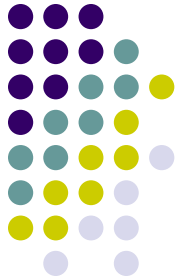


- In a back-propagation neural network, the learning algorithm has two phases.
- First, a training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer.
- If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.

Three-layer back-propagation neural network



Formulation as a minimisation of the error



$$\min_{\mathbf{w}} E(\mathbf{w}).$$

General weight update rule:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{d}_k,$$

$\mathbf{d}_k$: search direction

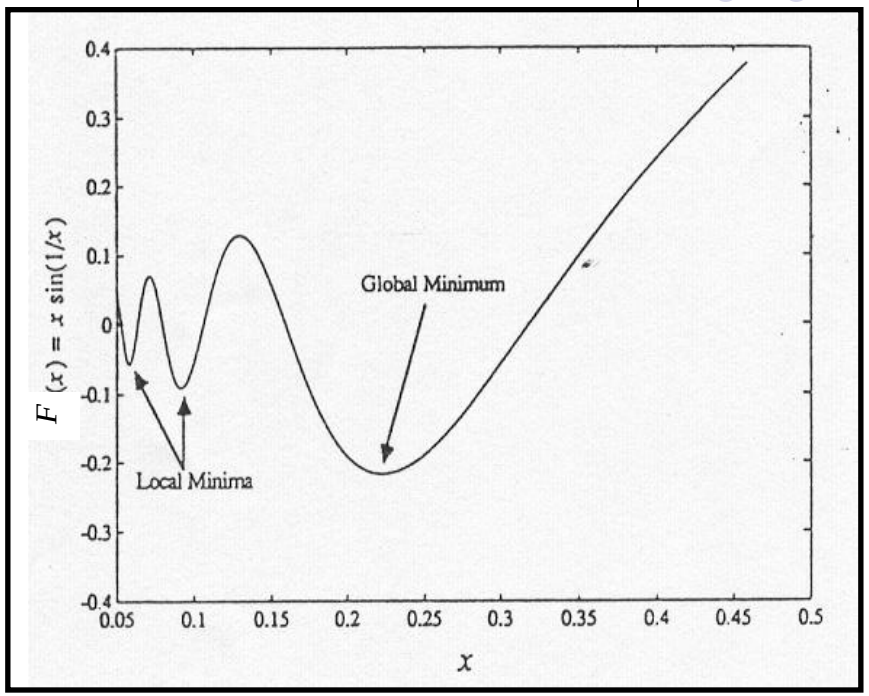
α_k : stepsize

Formulation as a minimisation of the error

Gradient vector

of function $F(\mathbf{x})$:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \\ \cdots \\ \frac{\partial}{\partial x_n} F(\mathbf{x}) \end{bmatrix}$$



First derivative of $F(\mathbf{x})$ with respect

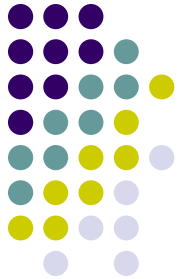
to x_i (ith element of gradient vector):

$$\partial F(\mathbf{x}) / \partial x_i$$

Optimality Condition

$$\nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}^*} = \mathbf{0}$$

Steepest Descent



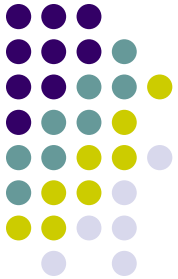
Gradient descent

Choose the next $\mathbf{x}$ so that: $F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$

Maximise the decrease by choosing: $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$

where $\mathbf{g}_k \equiv \nabla F(\mathbf{x})\big|_{\mathbf{x} = \mathbf{x}_k}$

Steepest Descent



$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

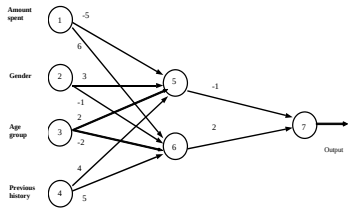
$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \qquad \alpha = 0.1$$

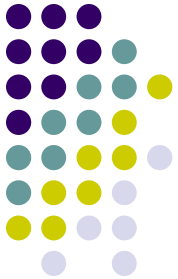
$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \qquad \mathbf{g}_0 = \nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

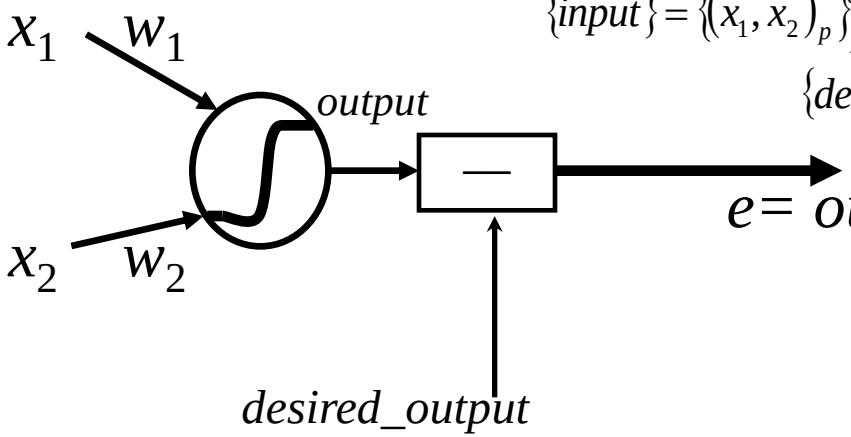
$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.1 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix}$$

$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha \mathbf{g}_1 = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix} - 0.1 \begin{bmatrix} 1.8 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.08 \end{bmatrix}$$

Nonlinear neuron







$$\{input\} = \{(x_1, x_2)_p\}_{p=1}^{8} = \{(-6,1),(-6.1,1),(-4.1,1),(-4,1),(4,1),(4.1,1),(6,1),(6.1,1)\}$$

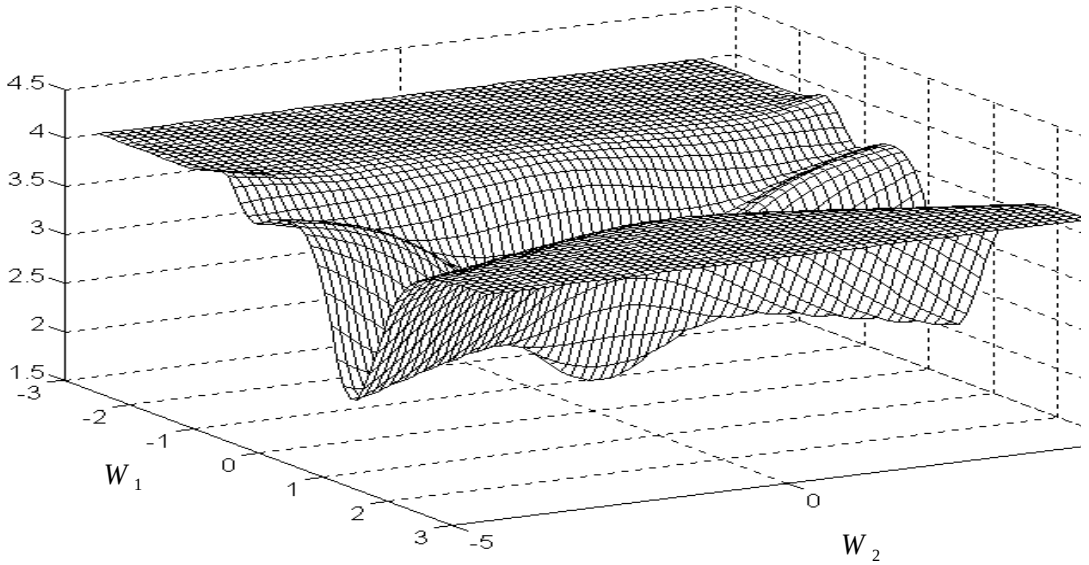
$$\{desired\_output\} = \{0,0,0.97,0.99,0.01,0.03,1,1\}$$

$$e = output - desired\_output$$

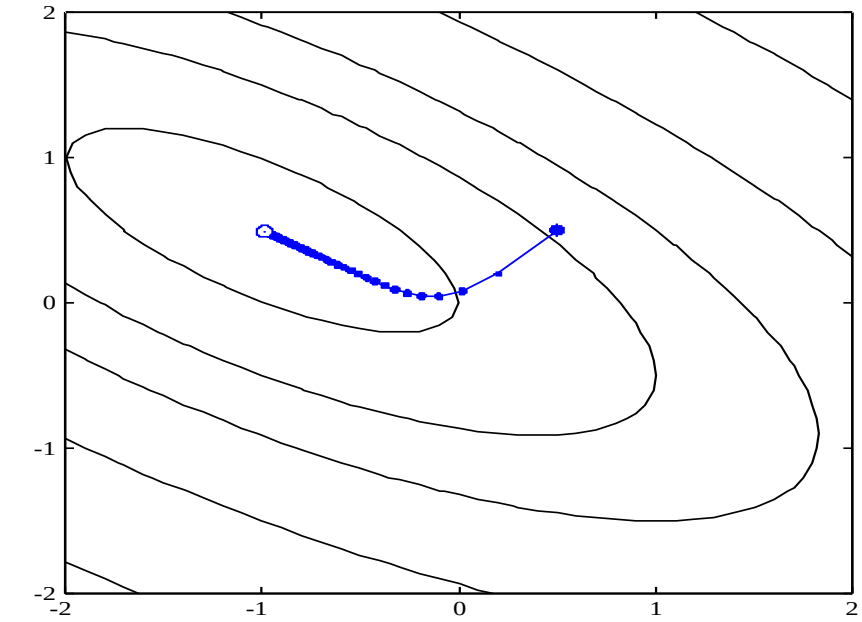
$$output = \frac{1}{1+e^{(w_1x_1+w_2x_2)}}$$

$$E = \sum_{i=1}^{p} e_i^2$$

Sum squared error function

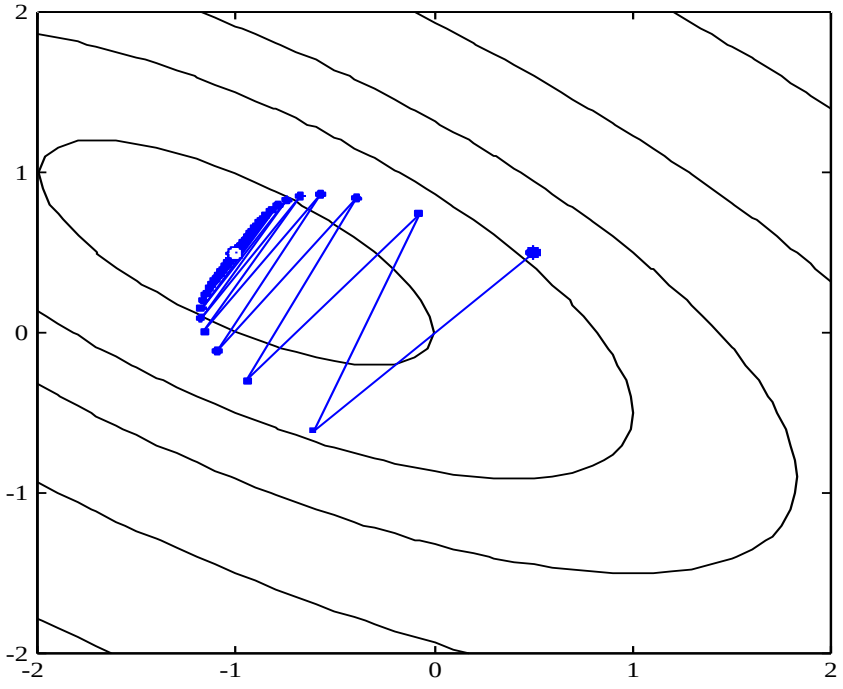


Effective backpropagation training with variable step size, Neural Networks, 1997

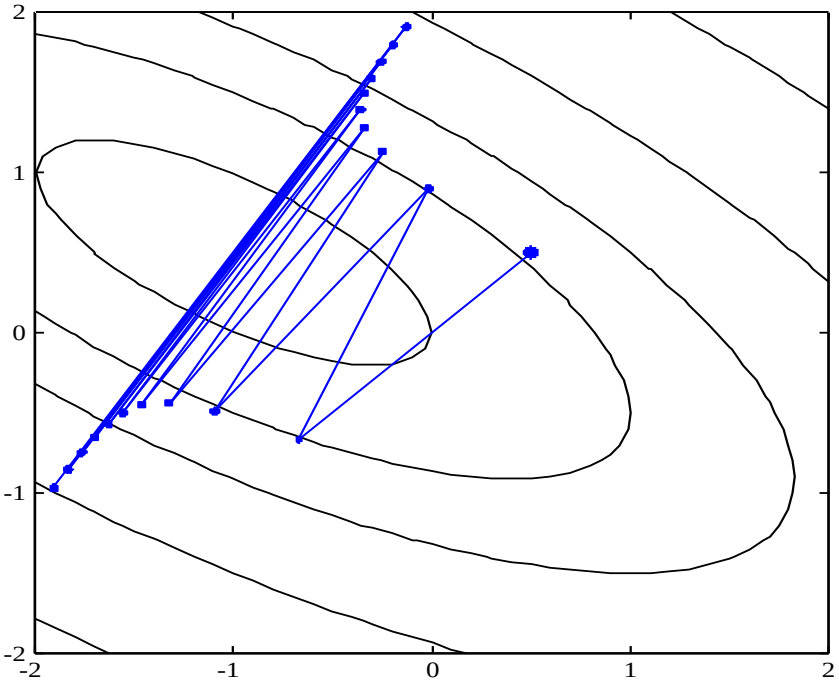


α = learning rate

α = 0.37

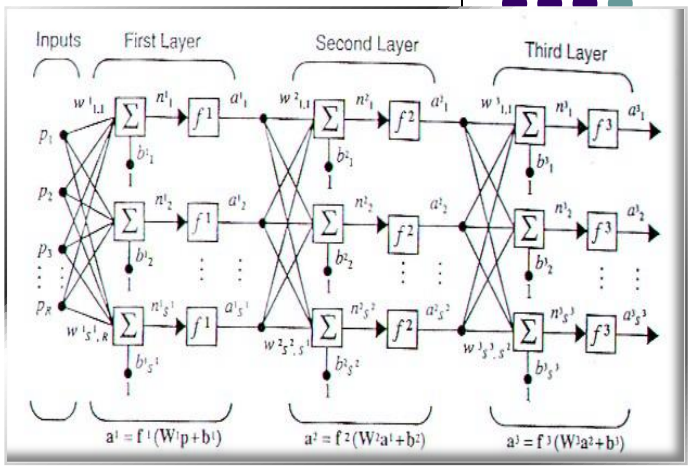


α = 0.39



The Chain Rule

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw}$$



Example **composite function**

$$f(n) = \cos(n) \qquad n = e^{2w} \qquad f(n(w)) = \cos(e^{2w})$$

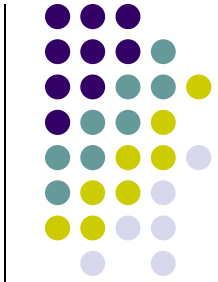
$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw} = (-\sin(n))(2e^{2w}) = (-\sin(e^{2w}))(2e^{2w})$$

Application to Gradient Calculation

$$\frac{\partial \hat{F}}{\partial w_{i,j}^{m}} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^{m}} \qquad \frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m}$$

Function:	Derivative:
$y = (f(x))^n$	$\frac{dy}{dx} = nf'(x)(f(x))^{n-1}$
$y = e^{f(x)}$	$\frac{dy}{dx} = f'(x)e^{f(x)}$
$y = \ln(f(x))$	$\frac{dy}{dx} = \frac{f'(x)}{f(x)}$
$y = \sin(f(x))$	$\frac{dy}{dx} = f'(x)\cos(f(x))$
$y = \cos(f(x))$	$\frac{dy}{dx} = -f'(x)\sin(f(x))$
$y = \tan(f(x))$	$\frac{dy}{dx} = f'(x)\sec^2(f(x))$

The back-propagation training algorithm



Step 1: Initialisation

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range:

$$\left(-\frac{2.4}{F_i}, \quad +\frac{2.4}{F_i} \right)$$

where F_i is the total number of inputs of neuron i in the network. The weight initialisation is done on a neuron-by-neuron basis.

Step 2: Activation

Activate the back-propagation neural network by applying inputs $x_1(p)$, $x_2(p)$,…, $x_n(p)$ and desired outputs $y_{d,1}(p)$, $y_{d,2}(p)$,…, $y_{d,n}(p)$.

(*a*) Calculate the actual outputs of the neurons in the hidden layer:

$$y_j(p) = sigmoid\left[\sum_{i=1}^{n} x_i(p) \cdot w_{ij}(p) - \theta_j\right]$$

where *n* is the number of inputs of neuron *j* in the hidden layer, and *sigmoid* is the *sigmoid* activation function.

Step 2: Activation (continued)

(*b*) Calculate the actual outputs of the neurons in the output layer:

$$y_k(p) = sigmoid\left[\sum_{j=1}^{m} x_{jk}(p) \cdot w_{jk}(p) - \theta_k\right]$$

where *m* is the number of inputs of neuron *k* in the output layer.

Step 3: Weight training

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.

(*a*) Calculate the error gradient for the neurons in the output layer:

$$\delta_k(p) = y_k(p) \cdot [1 - y_k(p)] \cdot e_k(p)$$

where

$$e_k(p) = y_{d,k}(p) - y_k(p)$$

Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \cdot y_j(p) \cdot \delta_k(p)$$

Update the weights at the output neurons:

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$

Step 3: Weight training (continued)

(*b*) Calculate the error gradient for the neurons in the hidden layer:

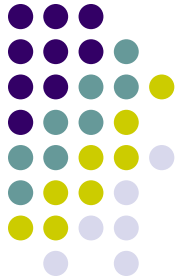
$$\delta_j(p) = y_j(p) \cdot [1 - y_j(p)] \cdot \sum_{k=1}^{l} \delta_k(p)\, w_{jk}(p)$$

Calculate the weight corrections:

$$\Delta w_{ij}(p) = \alpha \cdot x_i(p) \cdot \delta_j(p)$$

Update the weights at the hidden neurons:

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$



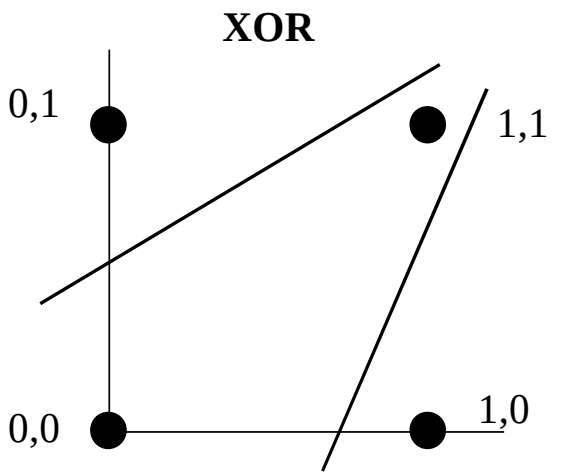
Step 4: Iteration

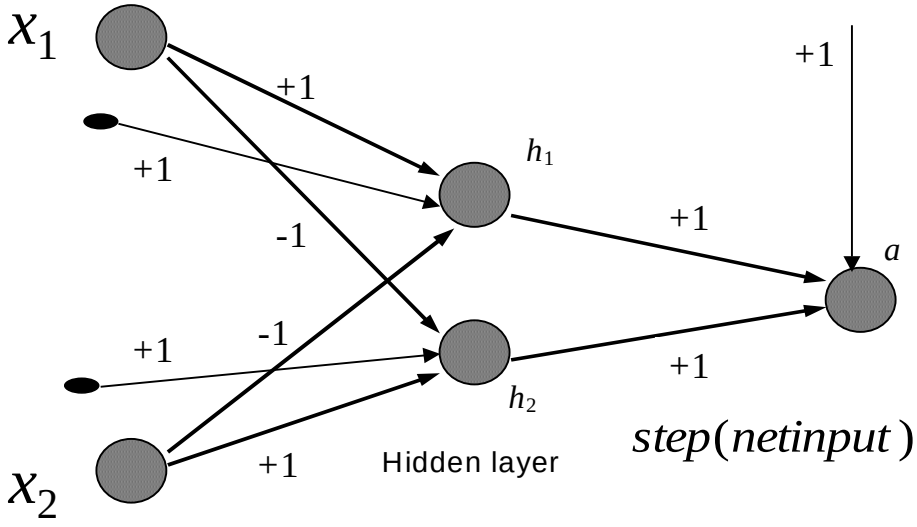
Increase iteration *p* by one, go back to *Step 2* and repeat the process until the selected error criterion is satisfied.

Multilayer networks and backpropagation

Let's see how a trained network operates on a nonlinear separable problem

Input x_1	Input x_2	Output XOR
0	0	0
1	0	1
0	1	1
1	1	0



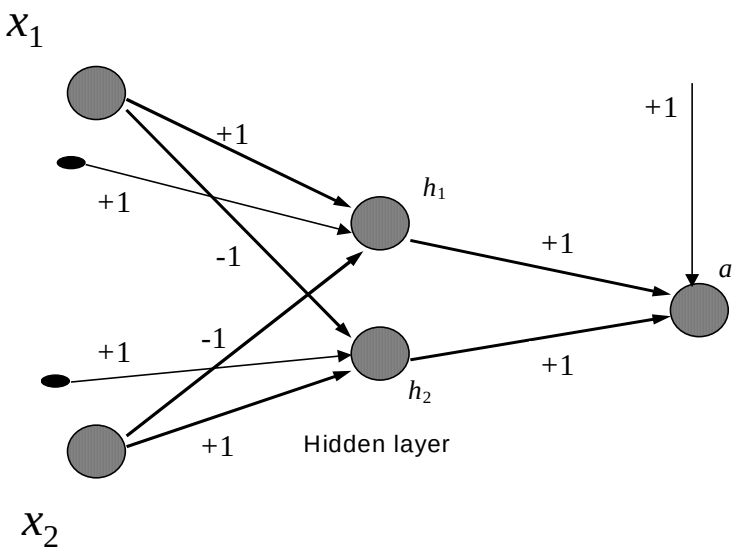


Each node in a **trained** MLP will have its own decision boundary

A single node can classify input vectors into two categories.

$$step(netinput) = \begin{cases} 1 & \text{if } netinput \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

$$step(netinput) = \begin{cases} 1 & \text{if } netinput > 1 \\ 0 & \text{otherwise} \end{cases}$$



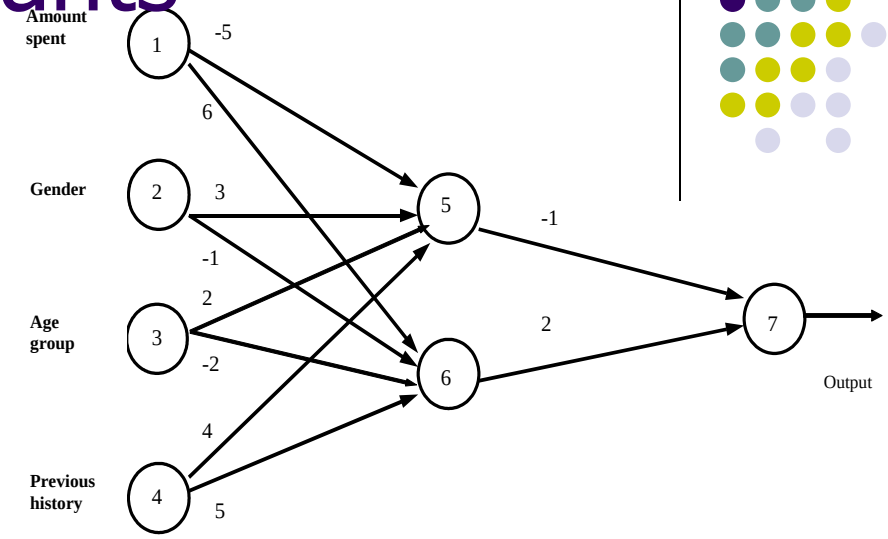
Hidden nodes' activities for the XOR problem

Input		Hidden		Targets
x_1	x_2	h_1	h_2	
0	0	0	0	0
1	0	1	0	1
0	1	0	1	1
1	1	0	0	0

$$h_1 = step[\ (+1) * x_1 + (-1) * x_2 + 1]$$

$$h_2 = step[\ (-1) * x_1 + (+1) * x_2 + 1]$$

Backpropagation variants



Backpropagation rule: $w^{new} = w^{old} + \Delta w$

$$w^{k+1} = w^k - \eta \nabla E(w^k)$$

Notation:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \\ \cdots \\ \frac{\partial}{\partial x_n} F(\mathbf{x}) \end{bmatrix}$$

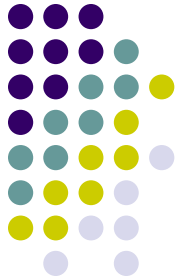
Consists of the first derivatives of $F(\mathbf{x})$ with respect to x_i (ith element of gradient vector): $\partial F(\mathbf{x}) / \partial x_i$

In multilayer networks is: $\frac{\partial E(w)}{\partial w_i} = \frac{\partial \left(\sum_{p=1}^{P} e_p^2 \right)}{\partial w_i}$

Backpropagation rule with momentum:

$$w^{k+1} = w^k - \eta \nabla E(w^k) + m \left(w^k - w^{k-1} \right)$$

Gradient-based algorithms for supervised learning



- **The Rprop method :** help to eliminate harmful influences of derivatives' magnitude on the weight updates.

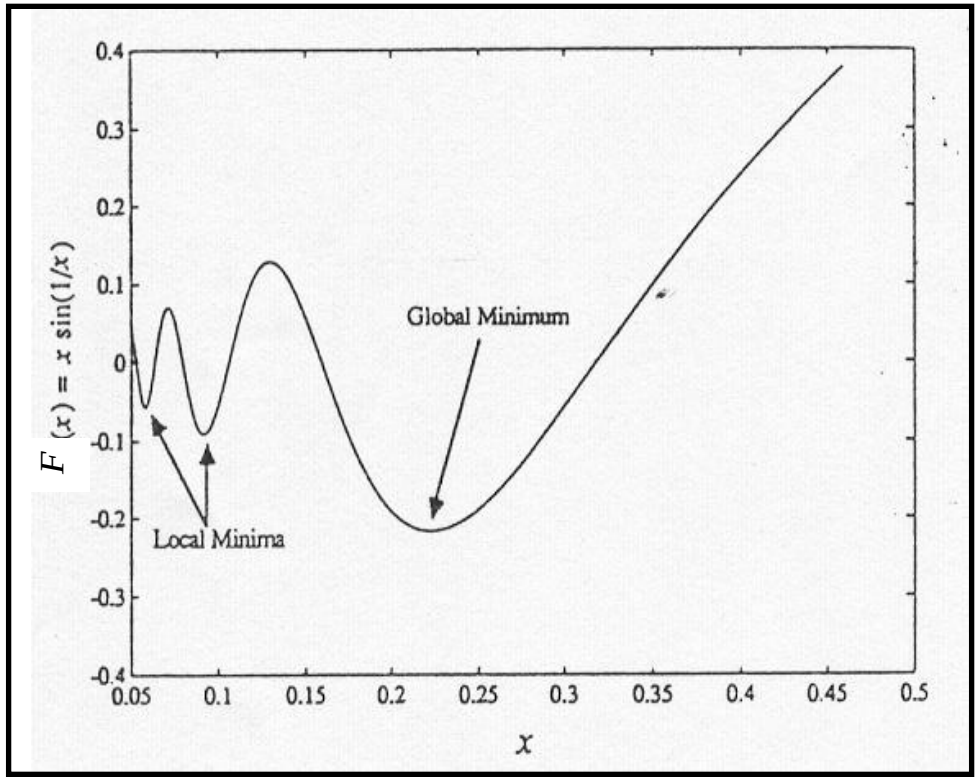
  Basic Idea: the sign of the derivative is used to determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update.

***The Resilient propagation update rule*:**

$$w^{k+1} = w^k - diag\{\eta_1^k, \cdots, \eta_i^k, \ldots \eta_n^k\} \cdot sign\left(g(w^k)\right)$$

$$\begin{aligned}
&if \quad \left(g_m(w^{k-1}) \cdot g_m(w^k) > 0\right) \quad then \quad \eta_m^k = \min\left(\eta_m^{k-1} \cdot \eta^+, \Delta_{\max}\right) \\
&if \quad \left(g_m(w^{k-1}) \cdot g_m(w^k) < 0\right) \quad then \quad \eta_m^k = \max\left(\eta_m^{k-1} \cdot \eta^-, \Delta_{\min}\right) \\
&if \quad \left(g_m(w^{k-1}) \cdot g_m(w^k) = 0\right) \quad then \quad \eta_m^k = \eta_m^{k-1}
\end{aligned}$$

Rprop



for each w_k do{

if $g_k^T * g_{k-1} > 0$ then{

$$\Delta_k = \min\left(\Delta_{k-1} \cdot \eta^+, \Delta_{\max}\right);$$

$$\Delta w_k = -\text{sign}\left(g_k\right) \cdot \Delta_k; \qquad \}$$

elseif $g_k^T * g_{k-1} < 0$ then{

$$\Delta_k = \max\left(\Delta_{k-1} \cdot \eta^-, \Delta_{\min}\right);$$

$$\Delta w_k = -\Delta w_{k-1};$$

$$g_k = 0; \qquad \}$$

elseif $g_k^T * g_{k-1} = 0$ then{.

$$\Delta_k = \Delta_{k-1};$$

$$\Delta w_k = -\text{sign}\left(g_k\right) \cdot \Delta_k; \qquad \}$$

$$w_{k+1} = w_k + \Delta w_k;$$

For deep networks- see Adapting Resilient Propagation for Deep Learning by Alan Mosca.

GRprop

Theorem 1. *Suppose that assumptions* (i) (iii) *are fulfilled, then for any* $w^0 \in \mathbb{R}^n$ *and any sequence* $\{w^k\}_{k=0}^{\infty}$ *generated by the Rprop scheme*

$$w^{k+1} = w^k - \tau^k \operatorname{diag}\{\eta_1^k, \ldots, \eta_i^k, \ldots, \eta_n^k\} \operatorname{sign}(g(w^k)), \quad k = 0, 1, \ldots, \tag{5}$$

where $\operatorname{sign}(g(w^k))$ *denotes the column vector of the signs of the components of* $g(w^k) = (g_1(w^k), g_2(w^k), \ldots, g_n(w^k))$, $\tau^k > 0$ *satisfying Wolfe's conditions,* $\eta_m^k (m = 1, 2, \ldots, i-1, i+1, \ldots, n)$ *are small positive real numbers generated by Rprop's learning rates schedule:*

$$\text{if } (g_m(w^{k-1}) \cdot g_m(w^k) > 0) \quad \text{then } \eta_m^k = \min(\eta_m^{k-1} \cdot \eta^+, \Delta_{\max}), \tag{6}$$

$$\text{if } (g_m(w^{k-1}) \cdot g_m(w^k) < 0) \quad \text{then } \eta_m^k = \max(\eta_m^{k-1} \cdot \eta^-, \Delta_{\min}), \tag{7}$$

$$\text{if } (g_m(w^{k-1}) \cdot g_m(w^k) = 0) \quad \text{then } \eta_m^k = \eta_m^{k-1}, \tag{8}$$

where $0 < \eta^- < 1 < \eta^+$, $\Delta_{\max}$ *is the learning rate upper bound,* $\Delta_{\min}$ *is the learning rate lower bound and*

$$\eta_i^k = -\frac{\sum_{j=1, j\neq i}^{n} \eta_j^k g_j(w^k) + \delta}{g_i(w^k)}, \quad 0 < \delta \ll \infty, \quad g_i(w^k) \neq 0, \tag{9}$$

it holds that $\lim_{k\to\infty} g(w^k) = 0$.

$$f(x^k + \tau^k d^k) - f(x^k) \leqslant \sigma_1 \tau^k g(x^k)^\top d^k, \tag{3}$$

$$g(x^k + \tau^k d^k)^\top d^k \geqslant \sigma_2 g(x^k)^\top d^k, \tag{4}$$

Included in R neuralnet- the R neural networks package by Frauke Günther and Stefan Fritsch.

Available online at www.sciencedirect.com
SCIENCE DIRECT
NEUROCOMPUTING
ELSEVIER
Neurocomputing 64 (2005) 253–270
www.elsevier.com/locate/neucom

New globally convergent training scheme based on the resilient propagation algorithm

Aristoklis D. Anastasiadis[a,*], George D. Magoulas[a], Michael N. Vrahatis[b]

[a]School of Computer Science and Information Systems, Birkbeck College, University of London, Malet Street, London WC1E 7HX, UK
[b]Computational Intelligence Laboratory, Department of Mathematics, University of Patras Artificial Intelligence Research Center (UPAIRC), University of Patras, GR-26110 Patras, Greece

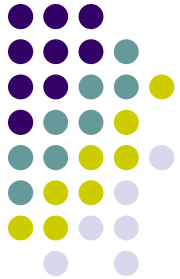
Available online 22 January 2005

Abstract

In this paper, a new globally convergent modification of the Resilient Propagation-Rprop algorithm is presented. This new addition to the Rprop family of methods builds on a mathematical framework for the convergence analysis that ensures that the adaptive local learning rates of the Rprop's schedule generate a descent search direction at each iteration. Simulation results in six problems of the PROBEN1 benchmark collection show that the globally convergent modification of the Rprop algorithm exhibits improved learning speed, and compares favorably against the original Rprop and the Improved Rprop, a recently proposed Rrpop modification.

Keywords: Supervised learning; Batch learning; First-order training algorithms; Convergence analysis; Global convergence property; Rprop; IRprop

Monotone convergence



Convergence condition: convergence requires that the search direction $\mathbf{d}_k$ is a *descent direction:*

$$\mathbf{d}_k^{\mathrm{T}} \nabla \mathbf{E}(\mathbf{w}_k) < 0,$$

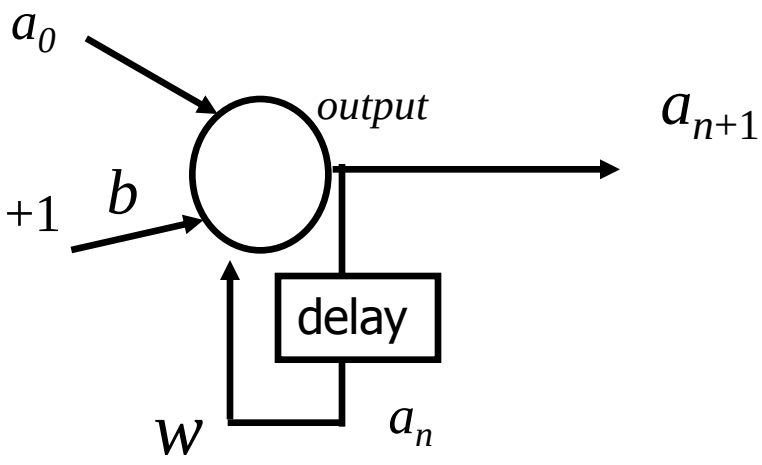
Monotone condition

$$f(w_k + a_k d_k) \leq f(w_k) + \delta a_k g_k^T d_k$$

It can be shown that if the learning rate satisfies the monotone condition then any algorithm of the form $\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{d}_k,$ converges to a local minimiser.

Recurrent Networks

model of a linear neuron



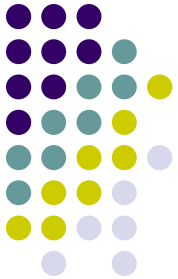
Consider a node with **linear activation function**

$$a_{n+1} = wa_n + b = 0.8a_n + 2$$

starting with $a_0 = 0$.

Compare it with the difference equation in example-1

example 1



Investigate the sequence obtained by iterating the following two difference equations starting with $x_0 = 0$

(i) $x_{n+1} = 0.2x_n + 2$

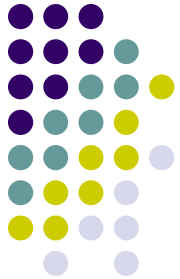
(ii) $x_{n+1} = 2x_n + 2$

	n	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
(i)	x_n	0	2	2.4	2.48	2.496	2.4992	2.49984	2.49996	2.5	2.5
(ii)	x_n	0	2	6	14	30	62	126	254	510	1022

- Equation (i) converges to the value of 2.5
- Equation (ii) diverges
- In general: the relation $x_{n+1} = ax_n + p$

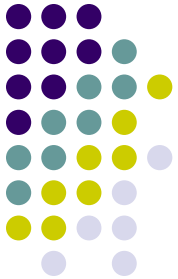
converges to a limit if $|a| < 1$ and diverges if $|a| \geq 1$

Discrete dynamic systems: definitions and examples



- These are systems in which the models are difference equations (also called *recurrence relations*). Because the dependent variable is found at discrete values of the independent variable, these are called **discrete models**.
- The *difference equation* is used repeatedly to generate a sequence of numbers once the initial term is known. The process of continually repeating an equation of this type is called **iteration**.

example 1



Investigate the sequence obtained by iterating the following two difference equations starting with $x_0 = 0$

(i) $x_{n+1} = 0.2x_n + 2$

(ii) $x_{n+1} = 2x_n + 2$

	n	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
(i)	x_n	0	2	2.4	2.48	2.496	2.4992	2.49984	2.49996	2.5	2.5
(ii)	x_n	0	2	6	14	30	62	126	254	510	1022

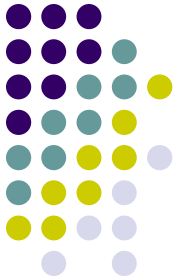
- Equation (i) converges to the value of 2.5
- Equation (ii) diverges
- In general: the relation $x_{n+1} = ax_n + p$

converges to a limit if $|a| < 1$ and diverges if $|a| \geq 1$

example 2

- Andrew, aged 18.5, won £20000 on the National Lottery. He invested it in the XTC Building Society at a fixed rate of interest of 6.5% compounded annually. How much would Andrew have on his 25th birthday?

example 2



- At the end of first year:

x_1 = initial amount + interest

x_1 = 20000 + (6.5/100) * 20000 = 21300

- At the end of second year:

x_2 = 21300 + (6.5/100) * 21300 = 22684.5

- At the end of the *n*th year

$x_n = x_{n\text{-}1} + (6.5/100) * x_{n\text{-}1}$

example 2

n	0	1	2	3	4	5	6
x_n	20000	21300	22684.5	24158.99	25729.33	27401.73	29182.85

In this case we can say that

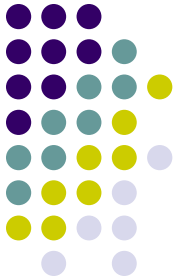
$x_{n+1} = x_n + (6.5/100) * x_n = 1.065\, x_n$

In general the equation model for a discrete dynamical system is:

$x_{n+1} = F(x_n)$

The function F is called **map** or **iteration function** of the system. If F is *linear* the difference equation is **linear**; otherwise it is **nonlinear**.

Linear discrete systems



- The general form of a linear difference equation is: $x_{n+1} = bx_n + a$

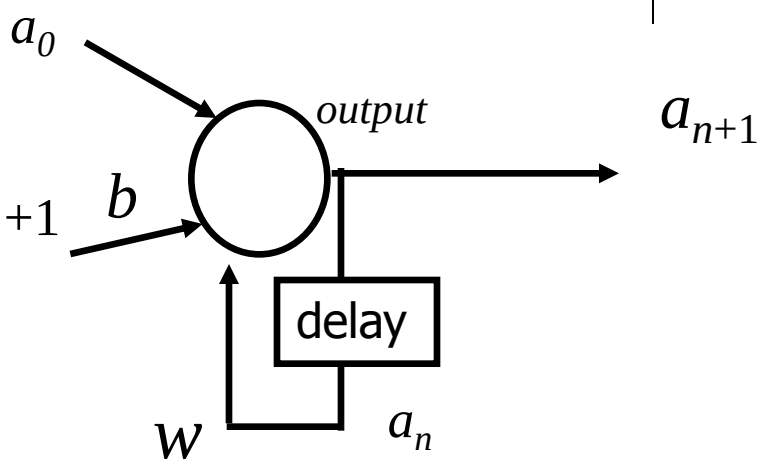
where a, b are constants and the *linear map* is: $F(x) = a + bx$

Linear recurrent neuron

- Consider the sequence generated by the equation

$$a_{n+1} = 0.8a_n + 2$$

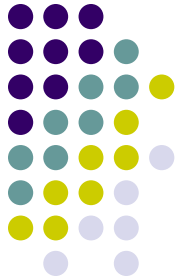
$$a_0 = 0$$



The sequence of points generated by the difference equation with initial value a_0 is called **orbit of a_n under mapping F**. The orbit of a dynamical system is at a **fixed point** a_f if $a_f = F(a_f)$. At the fixed point the system is fixed, it never moves.

The fixed points are clearly important in describing the motion of the system. If the initial state is at a fixed point then the system does not change.

Linear discrete systems



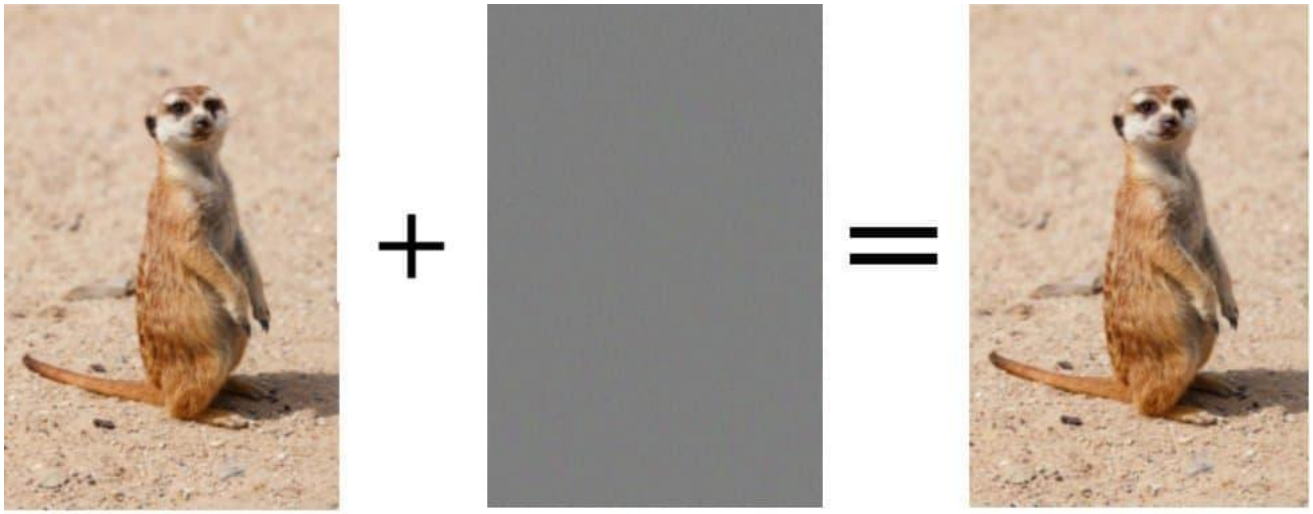
- If the system is given a small displacement from the fixed point then it will tend to return to the fixed point. The fixed point is said to be **stable** and is described as an **attractor**.
- If the system is given a small displacement from the fixed point then it will tend to move away from the fixed point. The fixed point is said to be **unstable** and is described as a **repellor**.

"panda"

57.7% confidence

"gibbon"

99.3% confidence



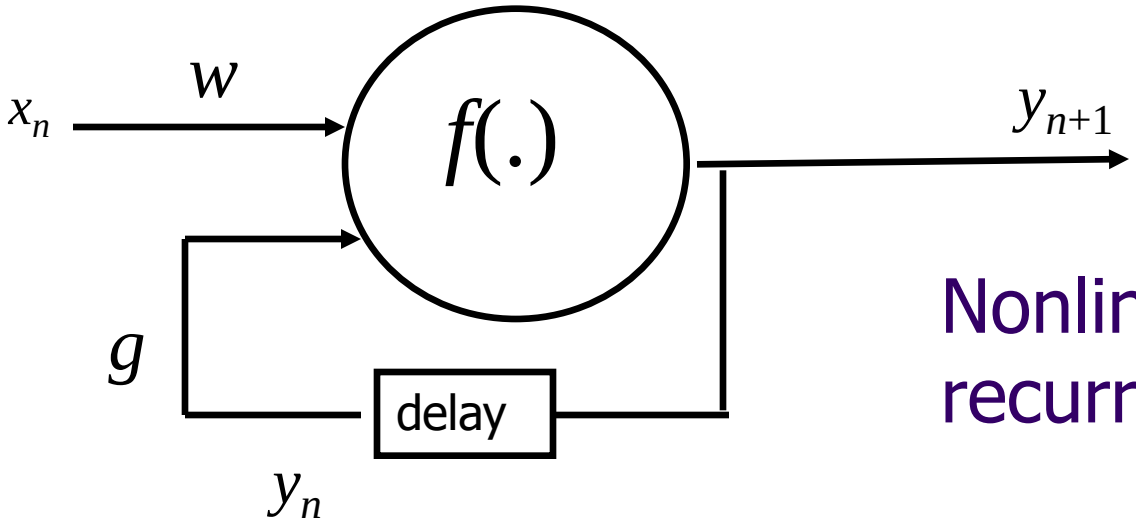
meerkat, mierkat (score = 0.90021)
mongoose (score = 0.02666)
Windsor tie (score = 0.00072)
otter (score = 0.00069)
doormat, welcome mat (score = 0.00055)

kite (score = 0.07896)
bald eagle, American eagle, Haliaeetus leucocephalus (score = 0.04153)
bee eater (score = 0.03940)
parachute, chute (score = 0.02724)
hummingbird (score = 0.02334)

doormat, welcome mat (score = 1.00000)
prayer rug, prayer mat (score = 0.00000)
manhole cover (score = 0.00000)
miniature poodle (score = 0.00000)
palace (score = 0.00000)

[Breaking Deep Learning with Adversarial examples using Tensorflow](#) by R. Awasthi

The Recurrent Neural Network architecture

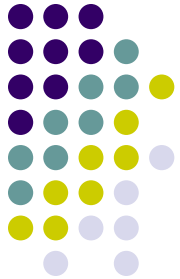


Nonlinear neuron with recurrent connection

$$y_{n+1} = f(w\,x_n + g\,y_n) \text{ or alternatively}$$

$$y(t+1) = f(w\,x(t) + g\,y(t))$$

Neuron with recurrent connection



$$y(t+1)= f\,(\, w\, x(t)+\, g\, y(t)\,)$$

Calculate the node's forward propagation:

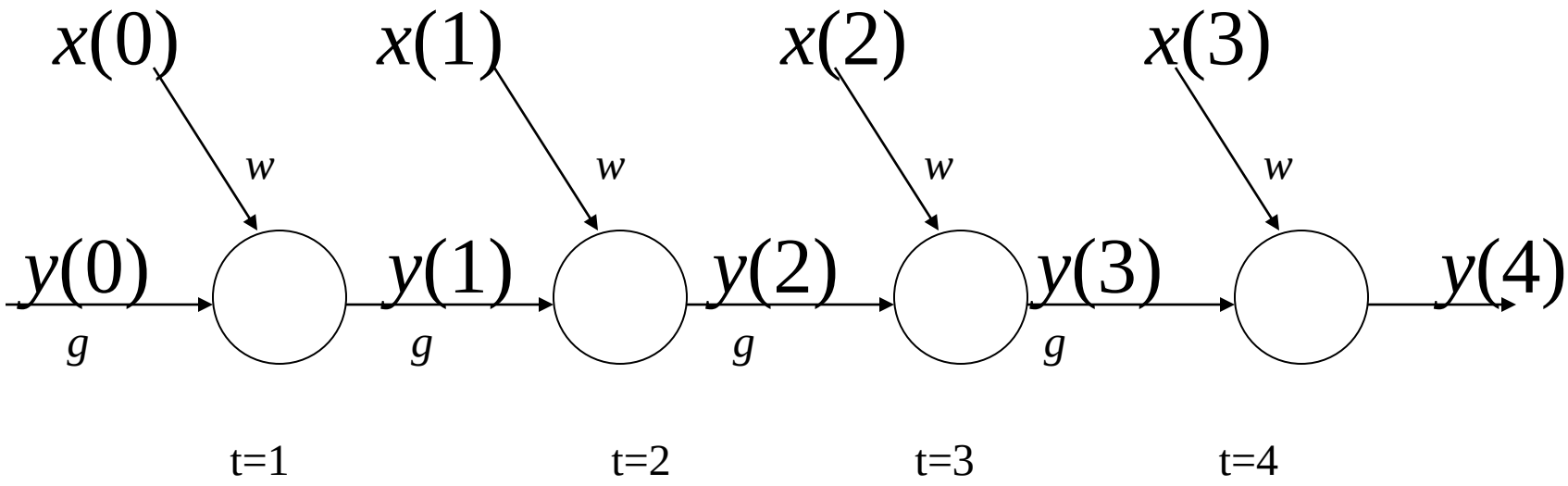
$$y(1)= f\,(\, w\, x(0)+\, g\, y(0)\,)$$

$$y(2)= f\,(\, w\, x(1)+\, g\, y(1)\,)$$

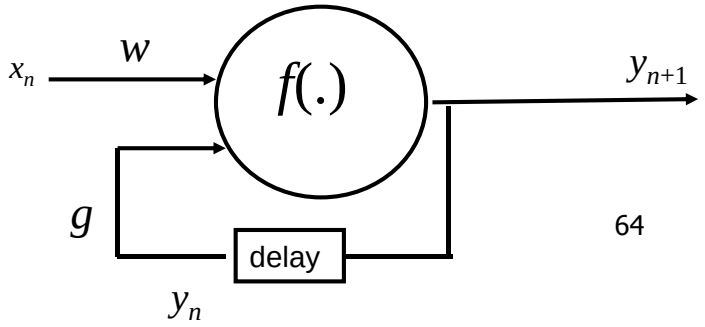
$$y(3)= f\,(\, w\, x(2)+\, g\, y(2)\,)$$

Neuron with recurrent connection

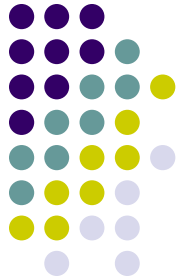
- **Unfolding** the node in time



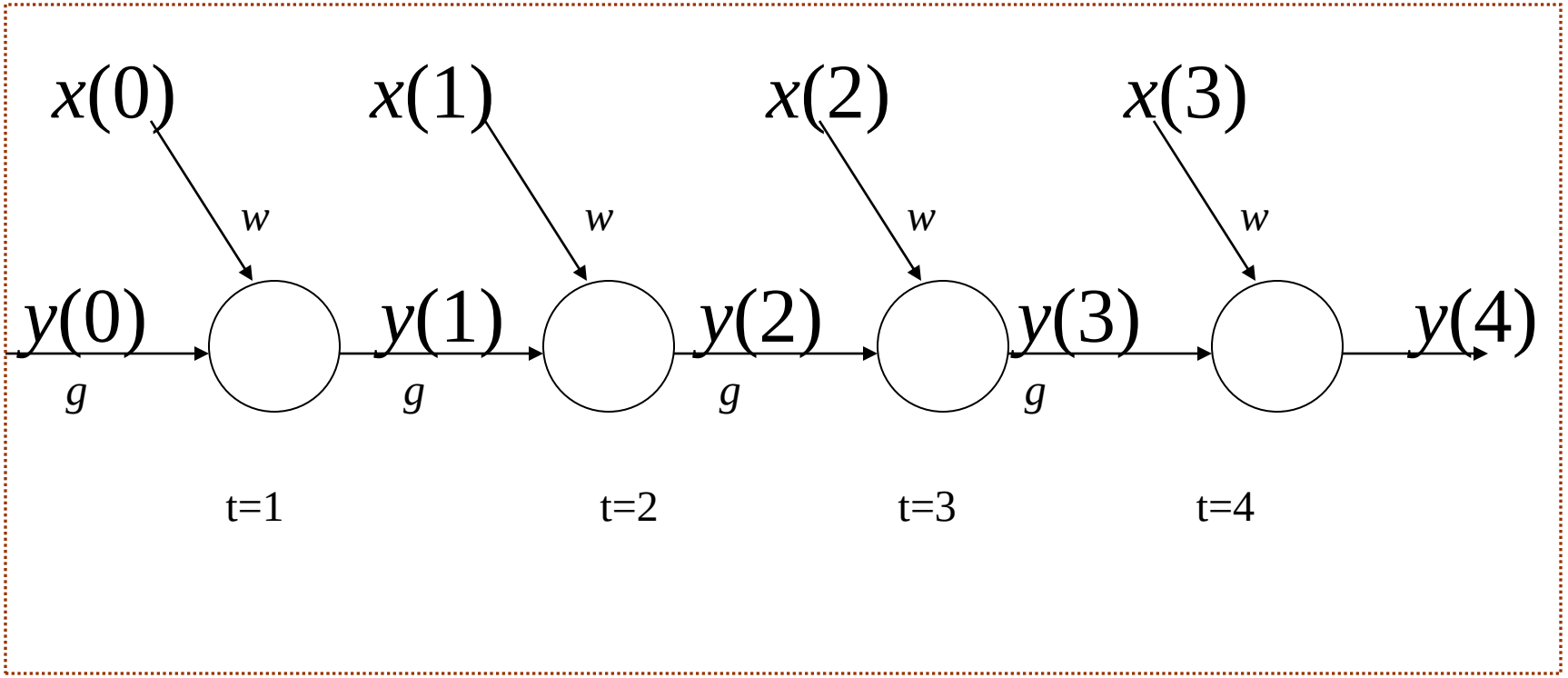
$$y(t+1) = f\left(w\, x(t) + g\, y(t) \right)$$



Neuron with recurrent connection



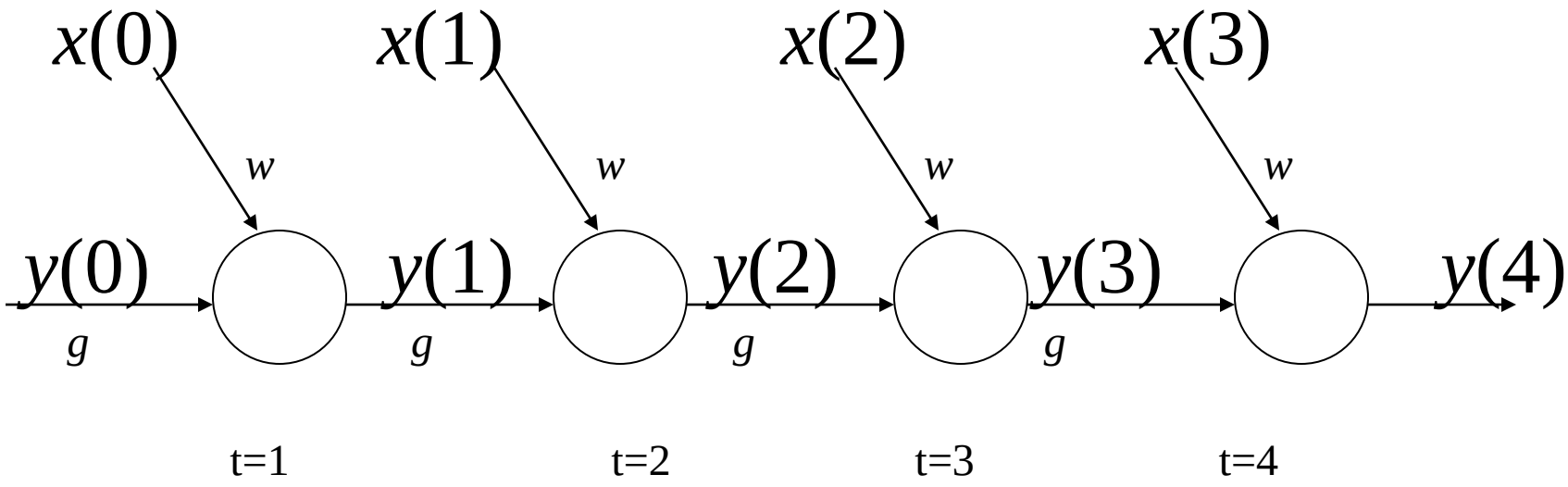
Equivalent with a **feedforward** neural network!



How many layers? a new layer is created for each time step of an input sequence processed by the network.

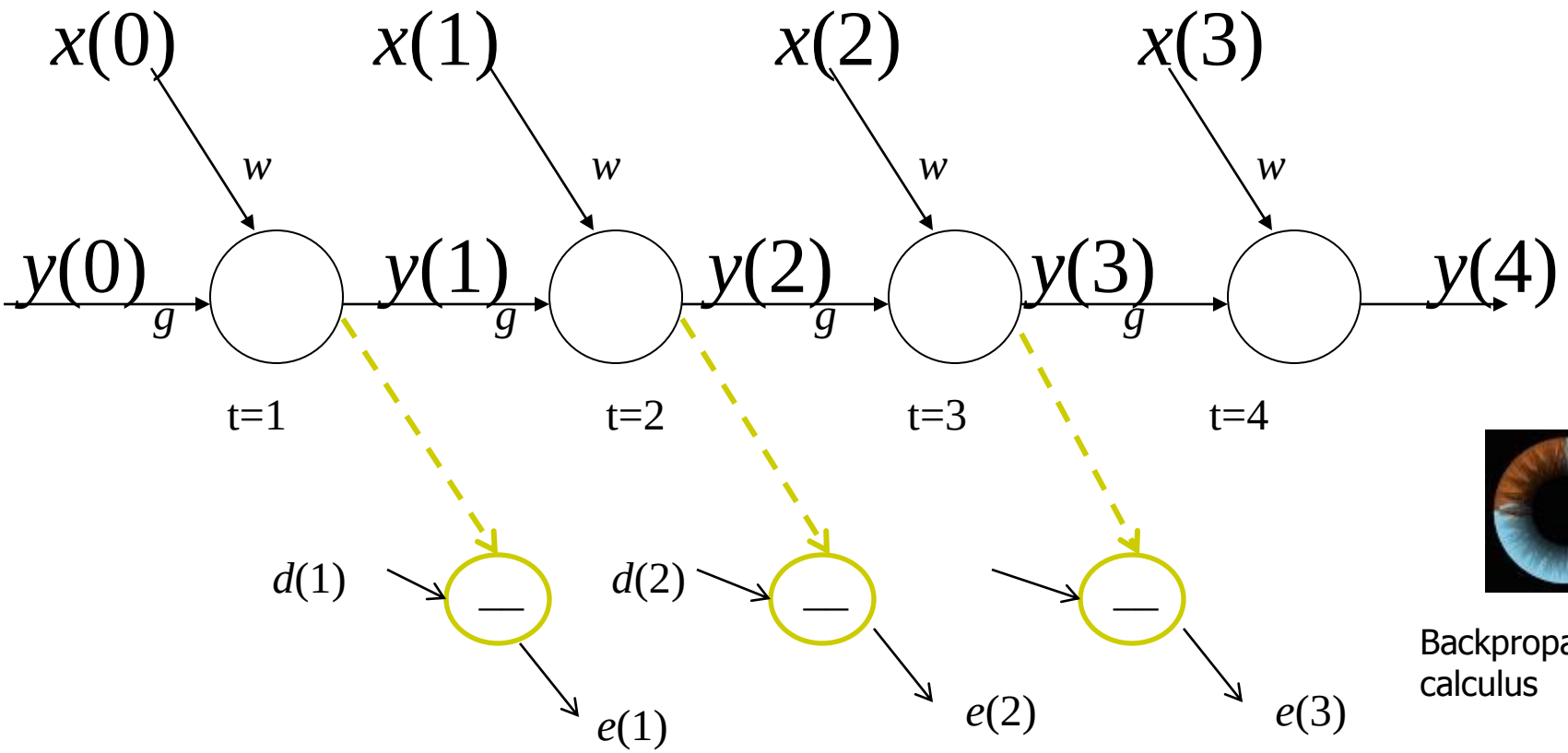
Neuron with recurrent connection

Train it like a feedforward neural network



Given the sequence $x(0), x(1), x(2), x(3)$ presented at the input and the initial condition $y(0)$ compute $y(1), y(2)\ldots$

Neuron with recurrent connection



Backpropagation calculus

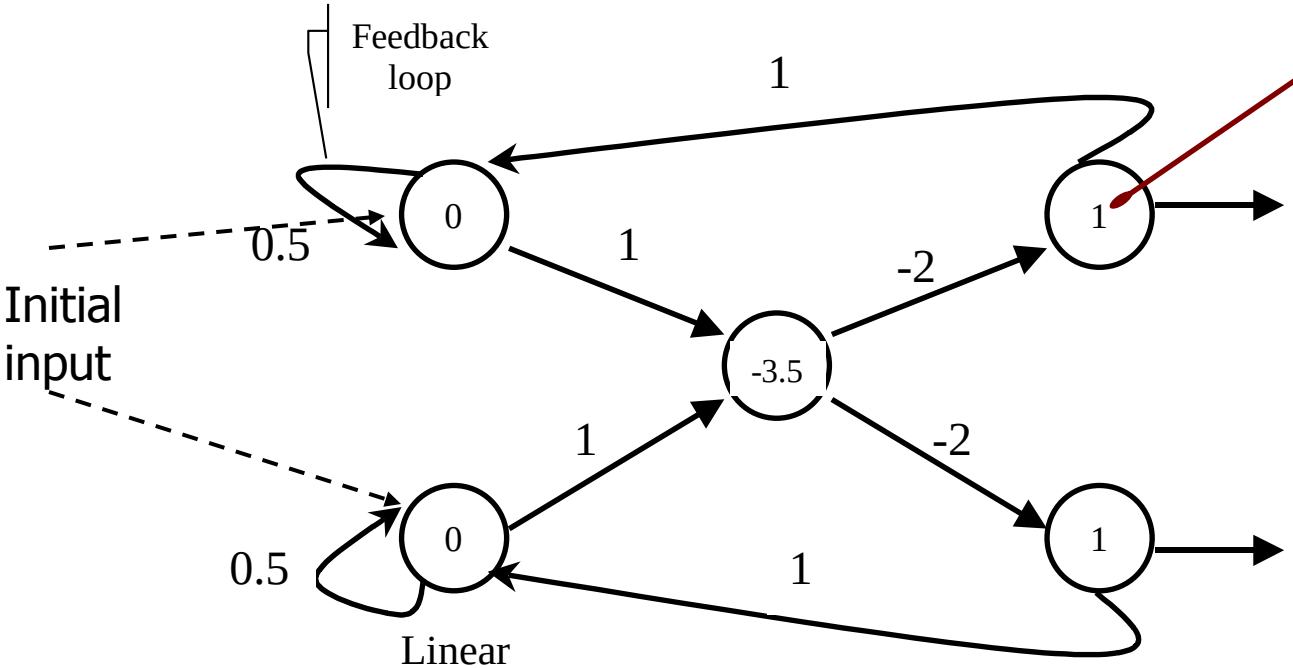
Calculate errors $e(1) = y(1) - d(1)$; $e(2) = y(2) - d(2)$; …

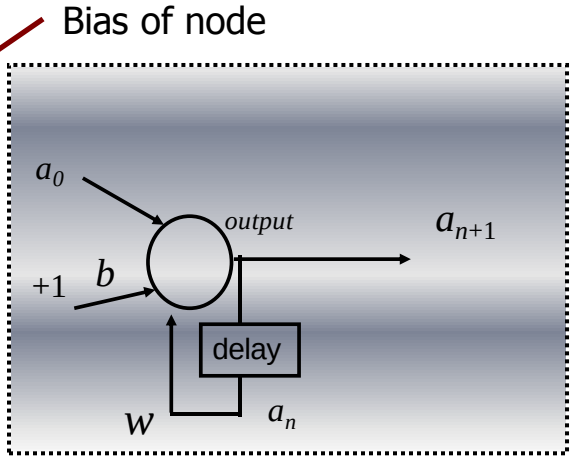
Let's see how a trained recurrent neural network operates

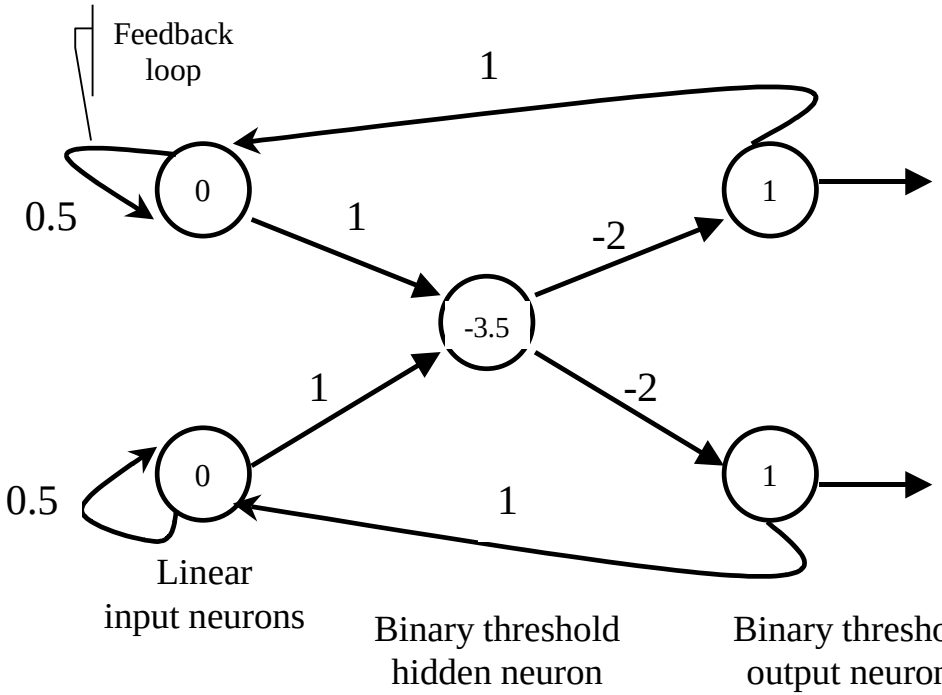
1.Predict the sequence AAAB, where A=(1,1) and B=(0,0)

2. Neurons are at 0 state. The input neurons receive an initial input (0,0) which is then removed;

3.Processing consists of allowing the network to fold its response back to the input units; neurons are activated when their input is ≥0.

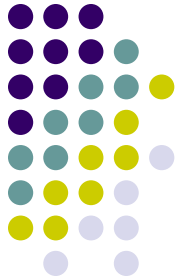






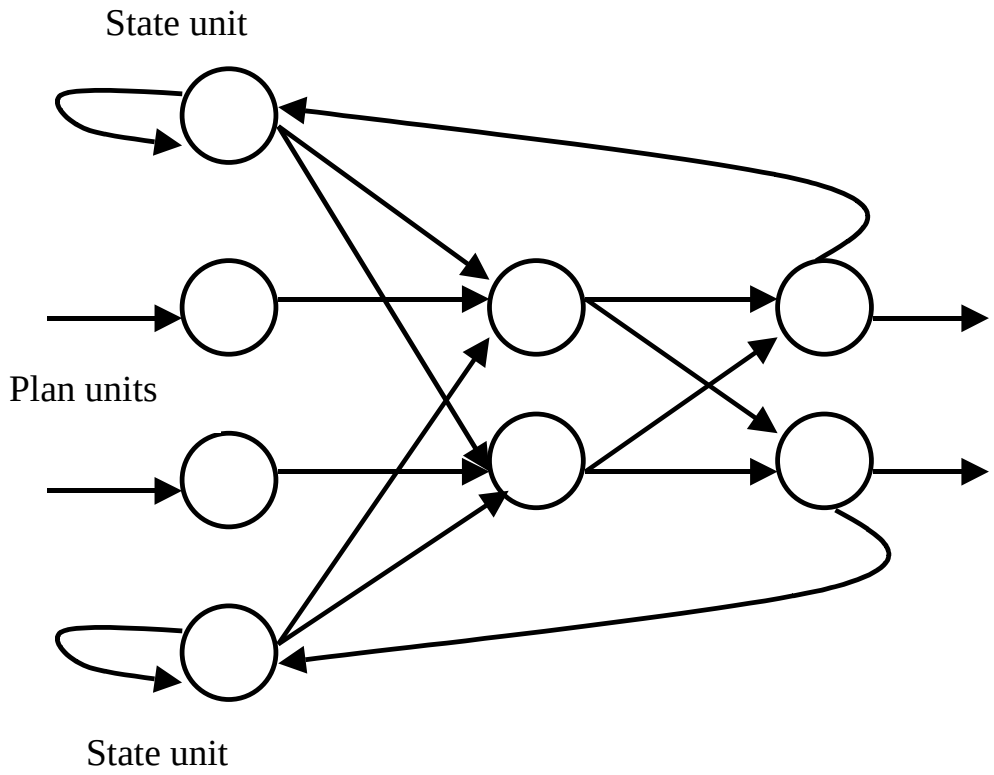
t	Input 1		Input 2		Hidden		Output 1		Output 2		Response
	In	Out	In	Out	In	Out	In	Out	In	Out	
1	0+0	0	0+0	0	0-3.5	0	0+1	1	0+1	1	A
2	1+0	1	1+0	1	2-3.5	0	0+1	1	0+1	1	A
3	1+0.5	1.5	1+0.5	1.5	3-3.5	0	0+1	1	0+1	1	A
4	1+0.75	1.75	1+0.75	1.75	3.5-3.5	1	-2+1	0	-2+1	0	B
5	0+0.875	0.875	0+0.875	0.875	1.75-3.5	0	0+1	1	0+1	1	A

The Recurrent Neural Network architecture



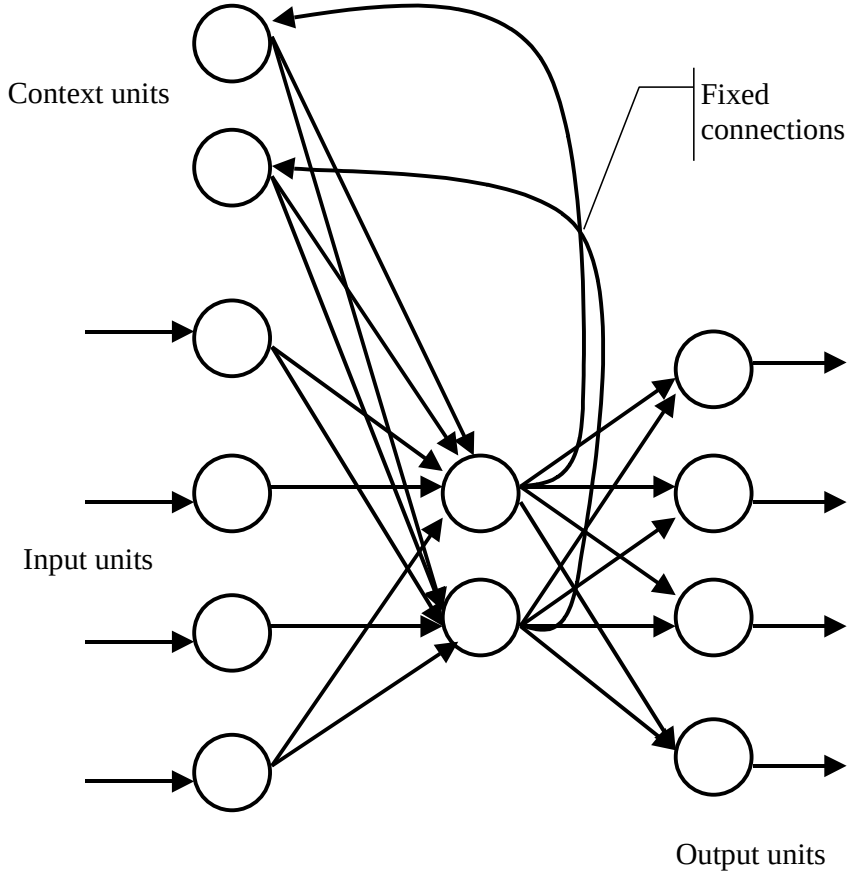
- The network can produce indefinitely long sequences of actions in a deterministic fashion.
- Given an initial state only one sequence of actions is produced.
- Can we make it learn more complex sequences?

RNN architecture 1



The *plan units* modulate the effects of the *state units*
The network is capable of learning more complex sequences

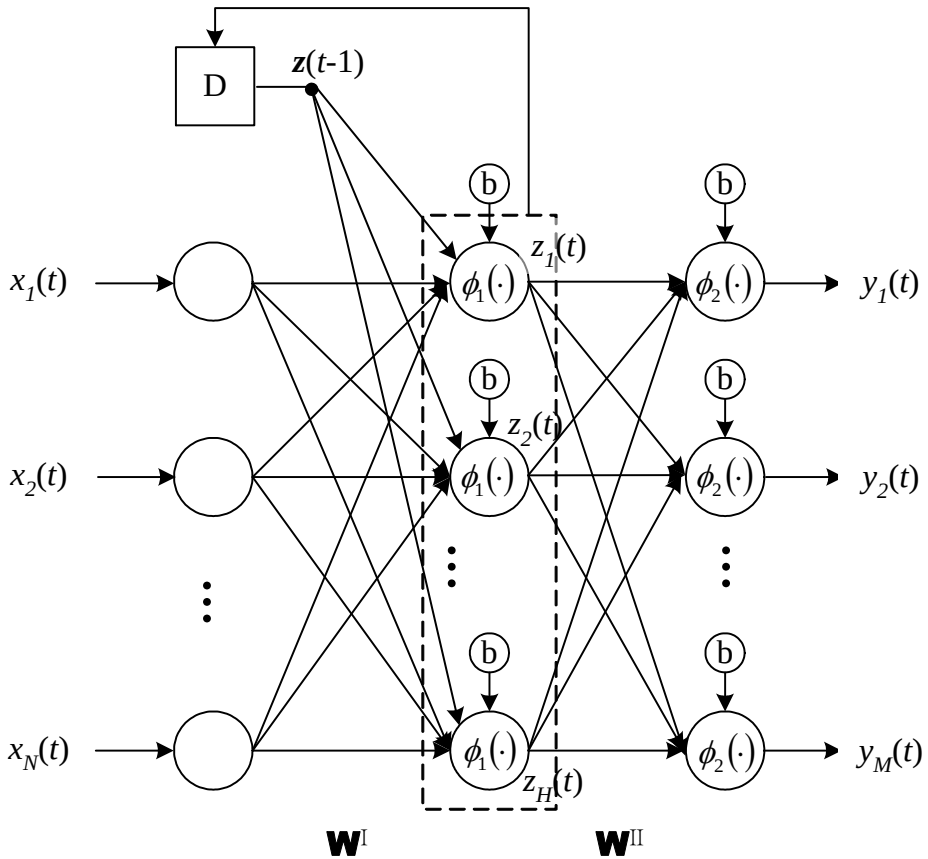
RNN architecture 2



ELMAN'S RECURRENT NETWORK

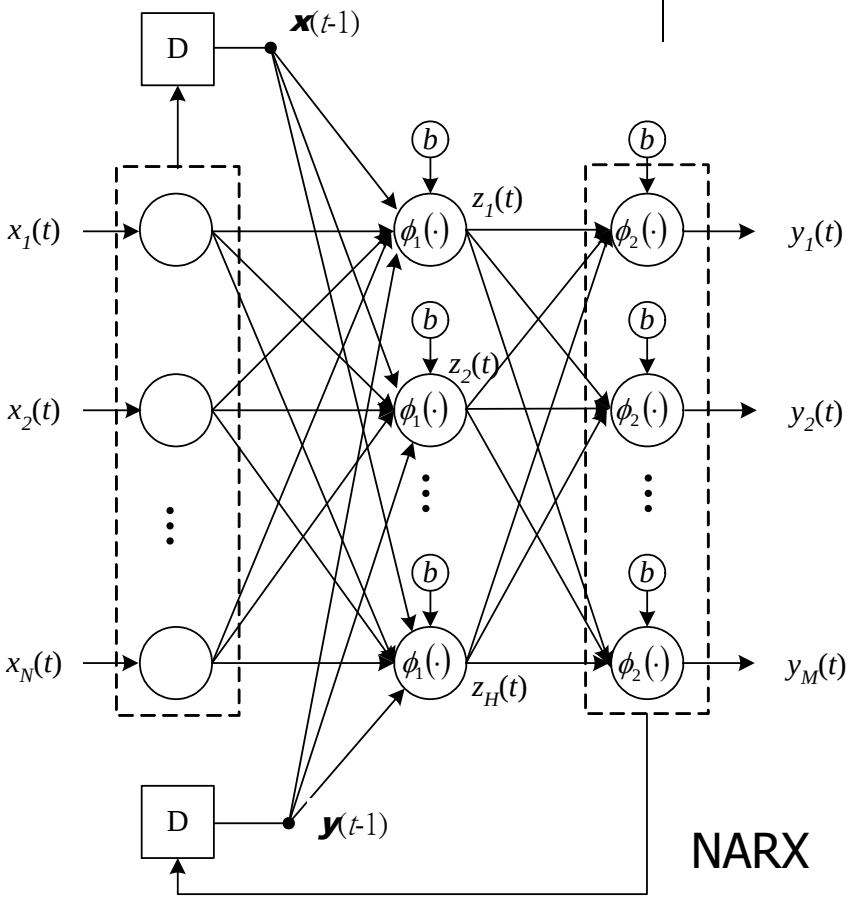
- The context units store the hidden unit activities for one time step
- The network can learn any sequence that is given as input
- The measure of learning performance is the ability to predict the next item in the sequence, i.e *learning to predict*
- Identical inputs are treated differently depending on the current status of the context

RNN architecture 3



LRN

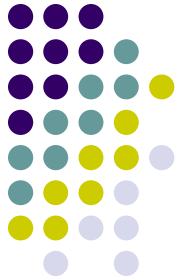
Input Layer	Hidden Layer	Output Layer



NARX

Input Layer	Hidden Layer	Output Layer

Backpropagation through time



Backpropagation rule: $w^{new} = w^{old} + \Delta w$

Notation:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \\ \cdots \\ \frac{\partial}{\partial x_n} F(\mathbf{x}) \end{bmatrix}$$

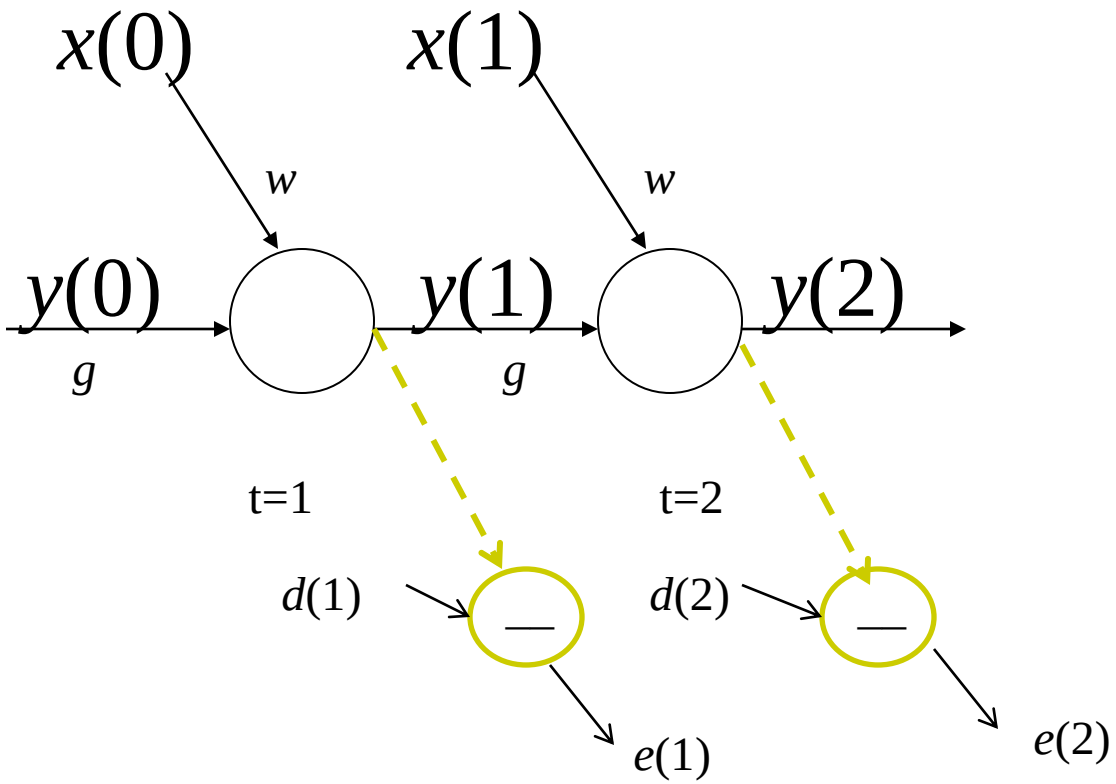
Consists of the first derivatives of $F(\mathbf{x})$ with respect to x_i (i th element of gradient vector)

In multilayer networks is:

$$\frac{\partial E(w)}{\partial w_i} = \frac{\partial \left(\sum_{p=1}^{P} e_p^2 \right)}{\partial w_i}$$

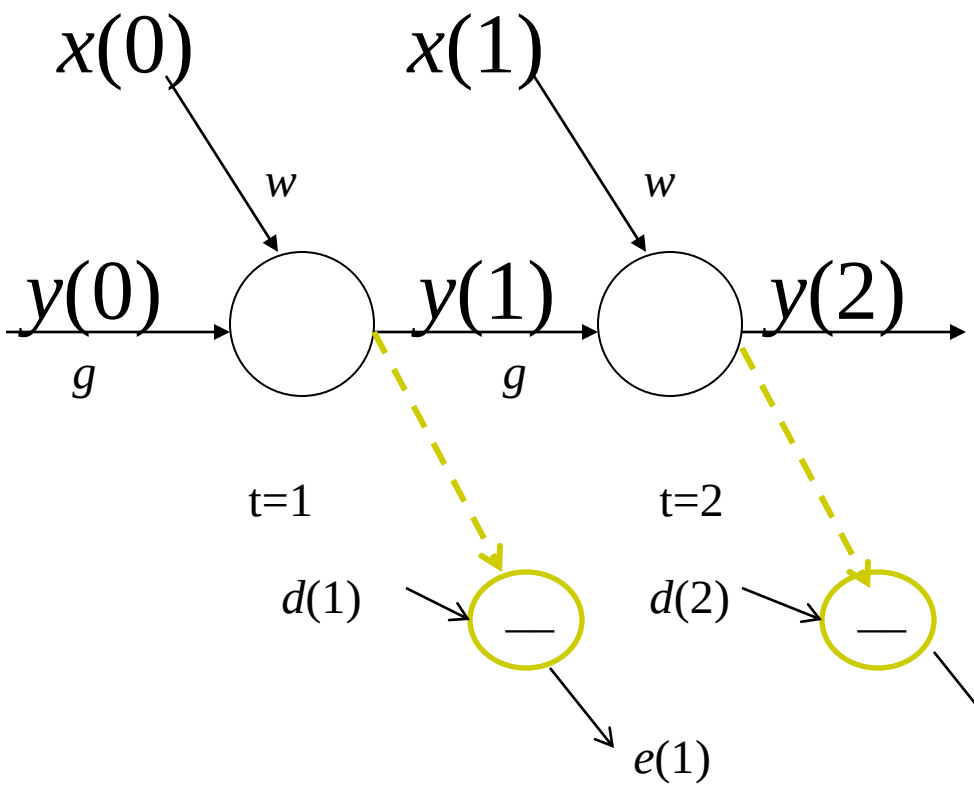
The recurrent version of the rule: **Backpropagation Through Time** - BPTT

Example: Neuron with recurrent connection



Calculate errors $e(1) = y(1) - d(1)$; $e(2) = y(2) - d(2)$; …

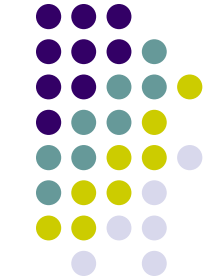
Example: Neuron with recurrent connection



1. Present data to the network as a time sequence of input and output pairs.
2. Unroll the network then calculate and accumulate errors across each time-step.
3. Roll-up the network and update weights.
4. Repeat.

Calculate errors $e(1) = y(1) - d(1)$; $e(2) = y(2) - d(2)$; …

Example: Neuron with recurrent connection



Layer number

- $\Delta w^2 = \eta \; e(2)\, y(2)\,[1- y(2)]\, x(1)$
- $\Delta g^2 = \eta \; e(2)\, y(2)\,[1- y(2)]\, y(1)$

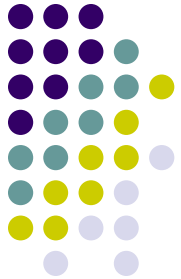
(for neurons with sigmoid activation functions) and

$e(2) = y(2) - d(2)$

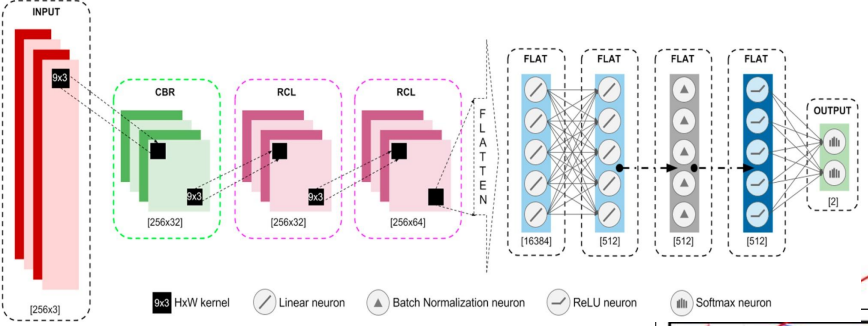
- $\Delta w^1 = \eta \; \delta^1 \; x(0)$
- $\Delta g^1 = \eta \; \delta^1 \; y(0)$

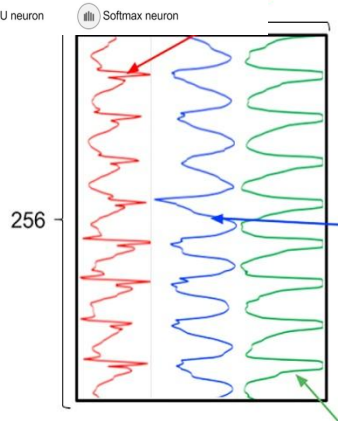
where $\delta^1 = y(1)\,[1- y(1)]\,\{e(1) + e(2)\, y(2)\,[1- y(2)]\, g\}$

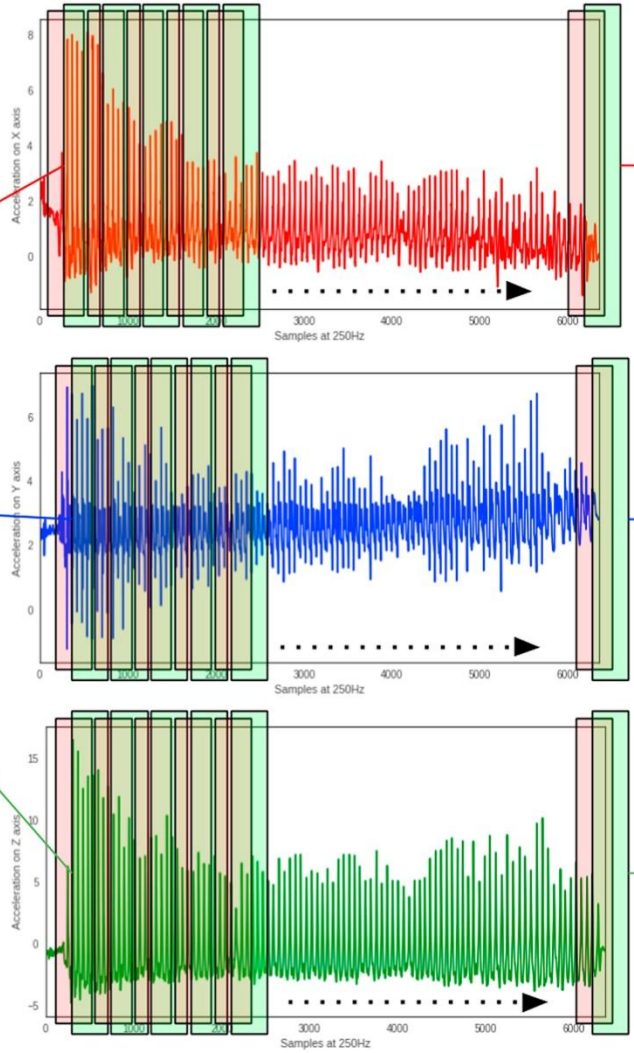
Backpropagation through time

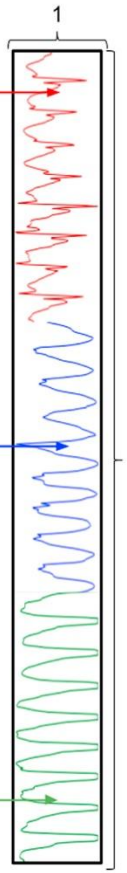


- BPTT can be computationally expensive as the number of time-steps gets higher.
- *If input sequences are comprised of 1000s of time-steps, then this will be the number of derivatives required for a single weight update.* This can cause weights to vanish or explode (go to zero or overflow) and make slow learning- the so-called **vanishing gradient** and **exploding gradient problems**- https://en.wikipedia.org/wiki/Vanishing_gradient_problem







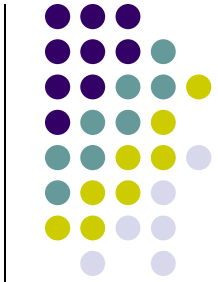


Reduce number of time-steps

Define sliding window length w and feed network shorter sequences of length w with predefined overlap

Sliding window of length 256 with 128-overlap applied on the recorded signals along the three acceleration axes to create the data set used by the recurrent network. Left side shows the data set in a 256 × 3 matrix form to feed the RCNN. Right side shows the same data in a 768 × 1 vector form.

Useful Reading



- Negnevitsky, Artificial Intelligence: a Guide to Intelligent Systems, 6.1-6.5. Available at the BBK Library.
- **Rprop**

  Anastasiadis A., Magoulas G.D., and Vrahatis M.N., New Globally Convergent Training Scheme Based on the Resilient Propagation Algorithm, Neurocomputing, vol. 64, 253-270, 2005.

Backpropagation neural networks and learning algorithms

- Rojas R. (1996), Neural Networks-A Systematic Introduction, chapters 7-8. Available online at:
- Ronald J. Williams and Jing Peng (1990), An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories.
- Werbos, P.J., (1990) Backpropagation through time: What it does and How to do it, Proceedings of the IEEE, vol. 78, 10.
- Hagan Martin T., Demuth Howard B., Beale Mark H. (1996), Neural Network Design, chapter 2, 9, 12, 14.

Next

- Genetic and evolutionary algorithms