COMP-4360 Machine Learning
Neural Networks

Jacky Baltes
Autonomous Agents Lab
University of Manitoba
Winnipeg, Canada
R3T 2N2
Email: jacky@cs.umanitoba.ca
WWW: http://www.cs.umanitoba.ca/~jacky
http://aalab.cs.umanitoba.ca

Introduction

- Threshold units
- Gradient descent
- Multilayer networks
- Backpropagation
- Hidden layer representation
- Example: Face recognition
- Extensions

Connectionist Models

- Human hardware
  - Neuron switching times approximately > 0.001s
  - Number of neurons approx. 10^10
  - Connections per neuron approx. 10^4 .. 10^5
  - Scene/face recognition 0.1s
  - 100 inference steps is very small
    - Massive parallelism

Connectionist Models

- Artificial Neural Nets
  - Many neuron-like threshold switching units
  - Many weighted connections between units
  - Highly parallel, distributed process
  - Emphasis on learning weights between connections

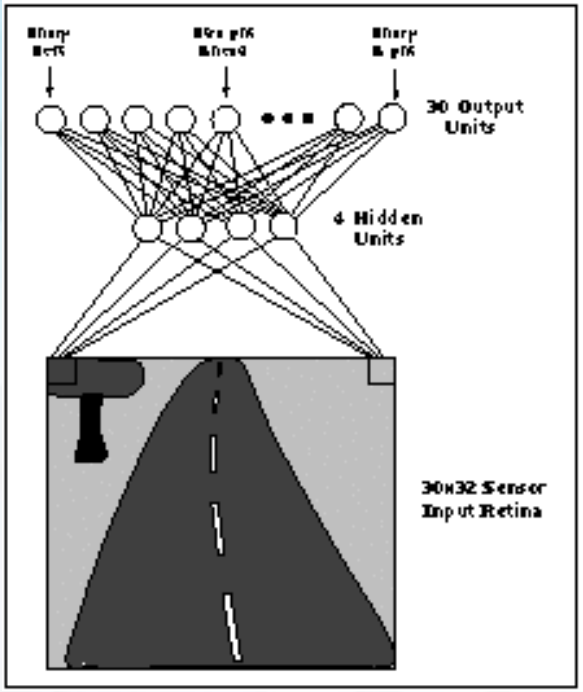
When to use Neural Nets?

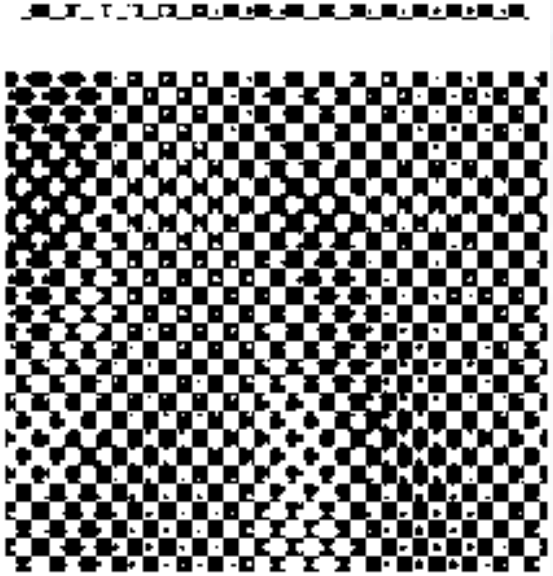
- Input is high-dimensional discrete or real-valued input (e.g. raw sensor data)
- Output is discrete or real-valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability is not important

When to use Neural Nets?

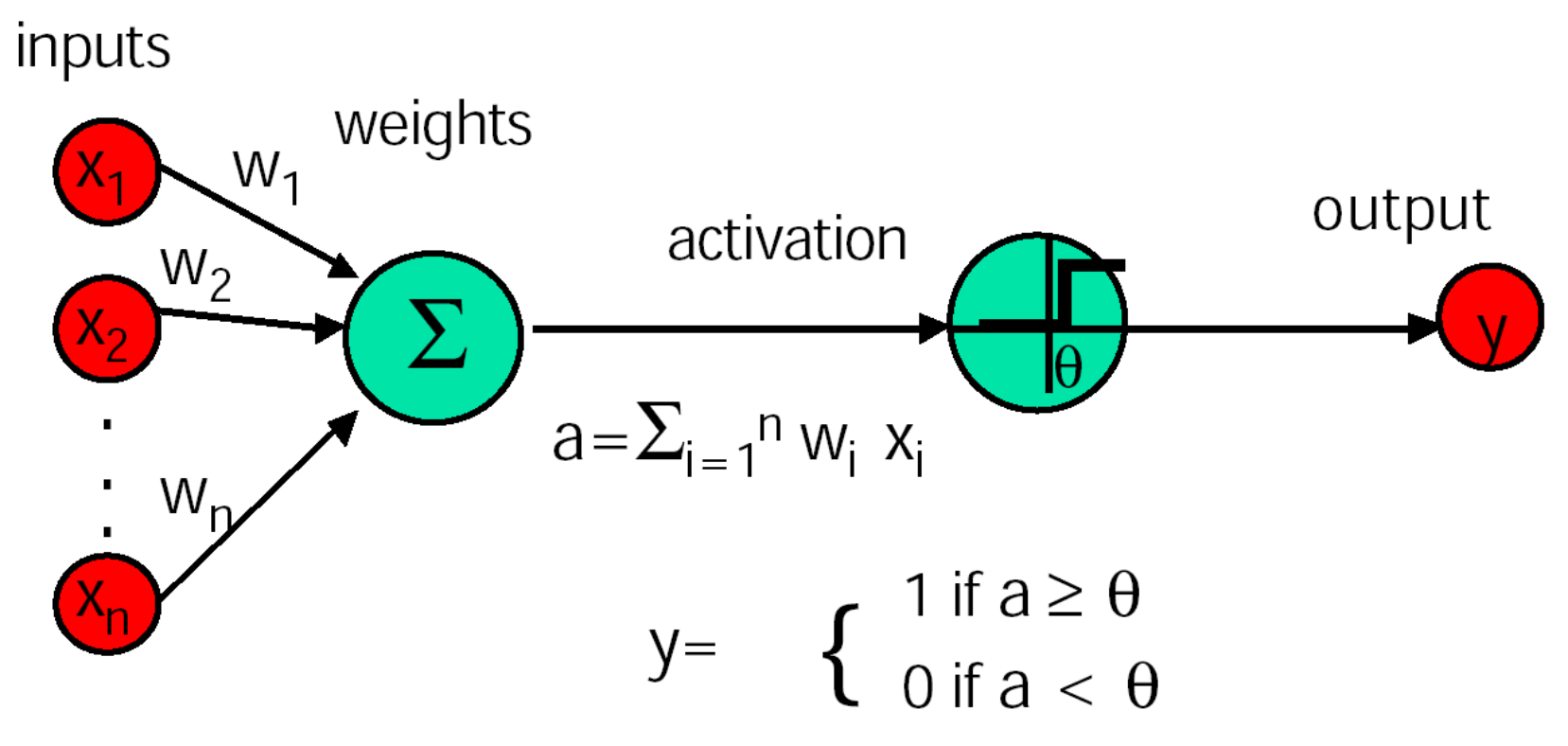
- Speech phoneme recognition (Waibel)
- Image classification (Kanade, Baluja, Rowley)
- Financial prediction
- Autonomous Driving

ALVINN Autonomous Driving





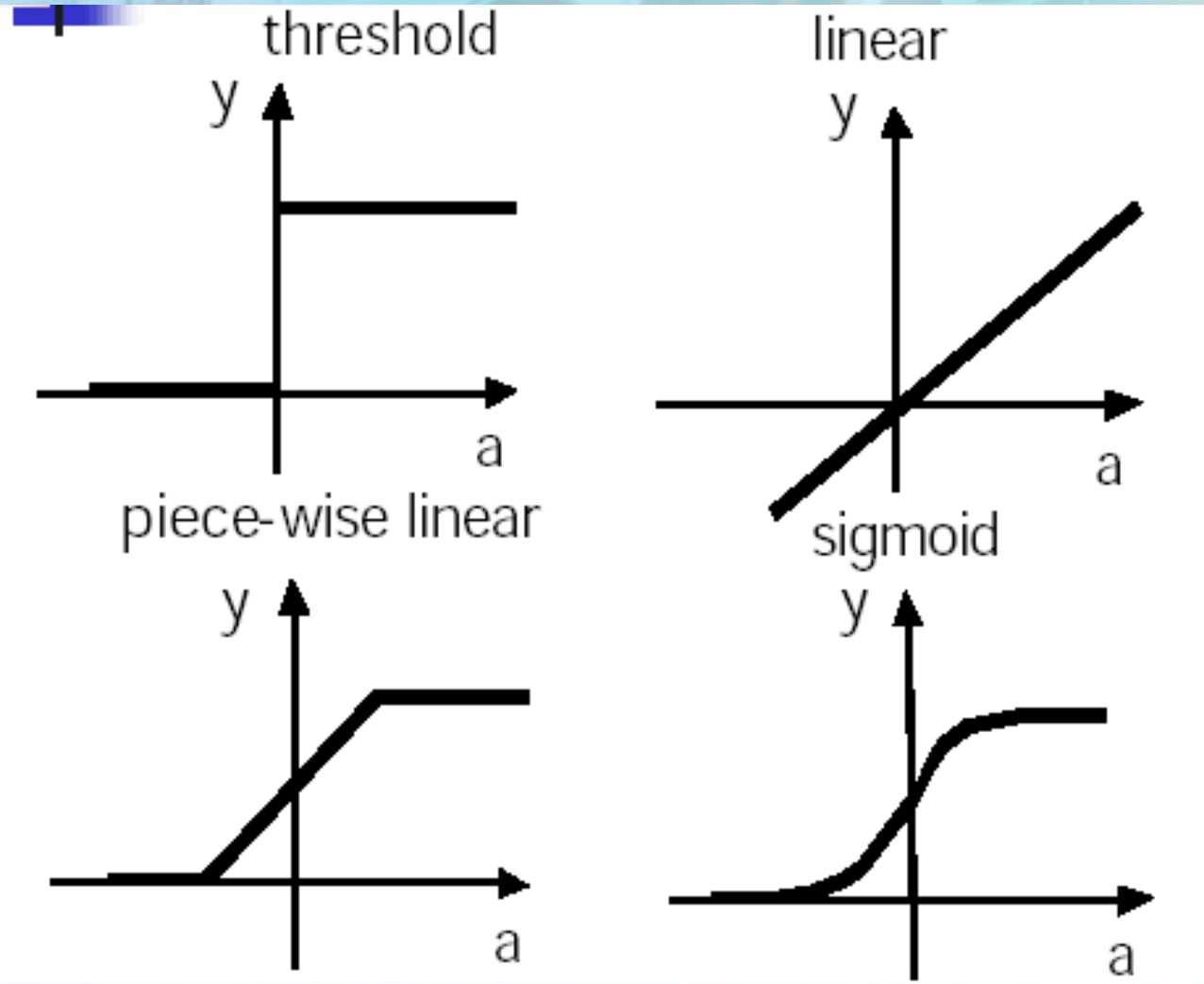
Perceptron (Threshold Learning Unit – TLU)



Sometimes we'll use simpler vector notation:

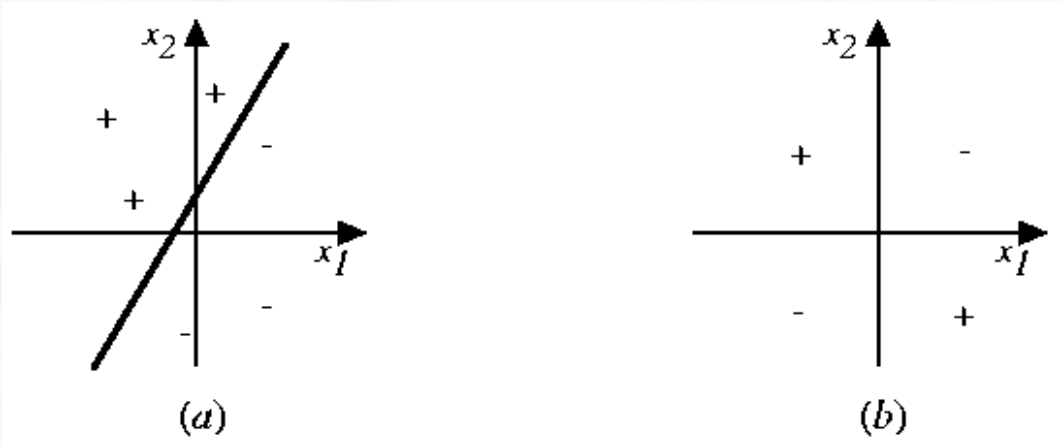
$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Activation Functions

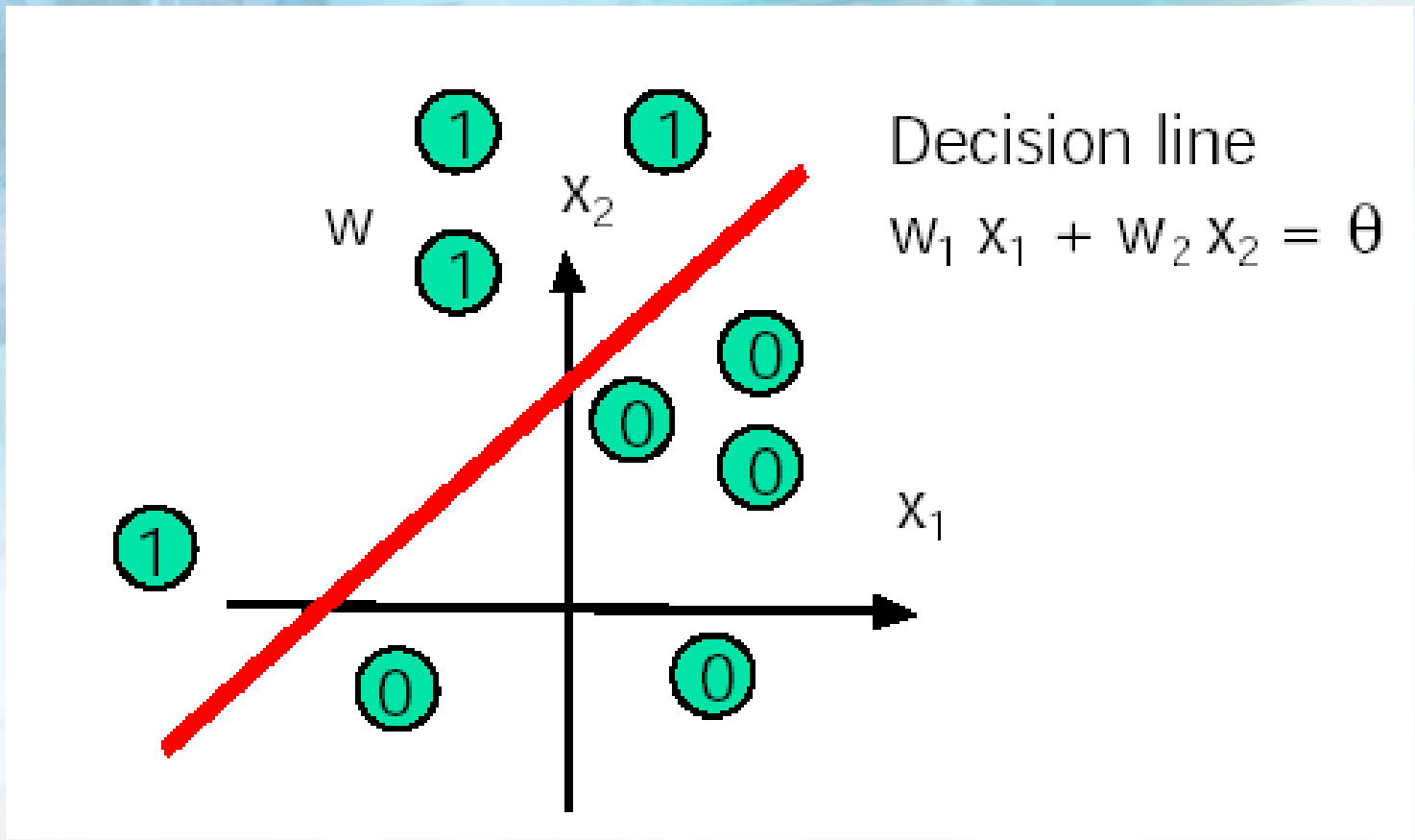


Decision Surface of a Perceptron

- Represents some useful functions
  - What are the weights for AND(x_1,x_2)
- But some functions can not be separated
  - Examples
- Therefore, we will use networks of these



Decision Surface of a Perceptron

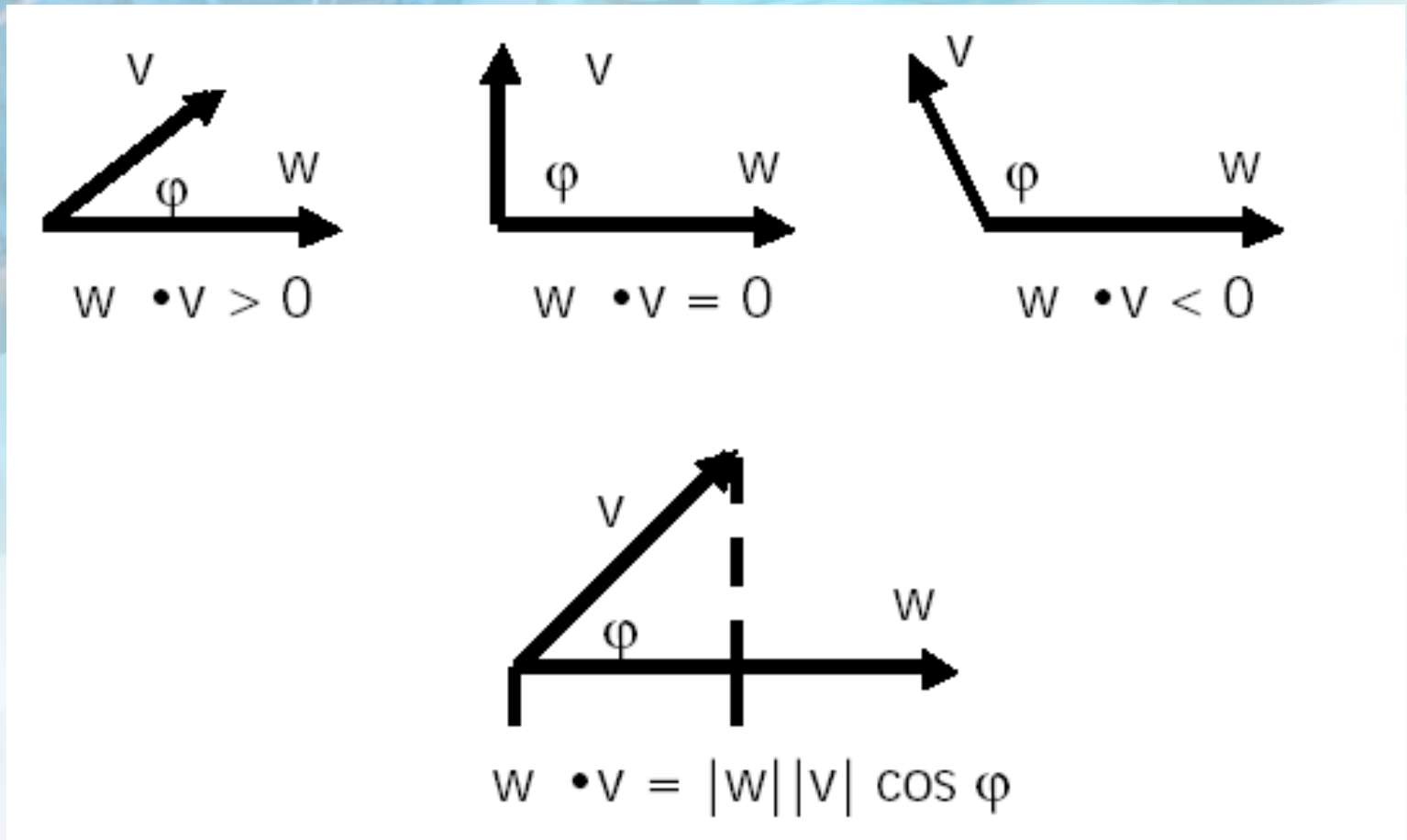


Dot Product

- Recall the dot product (·) of vectors w and x is $w_1x_1+w_2x_2+...$
- Identical equation as the decision line for a neuron in two dimensional space
- dot product also has a geometric interpretation in terms of vectors and their angles to one another

Scalar Products and Projections

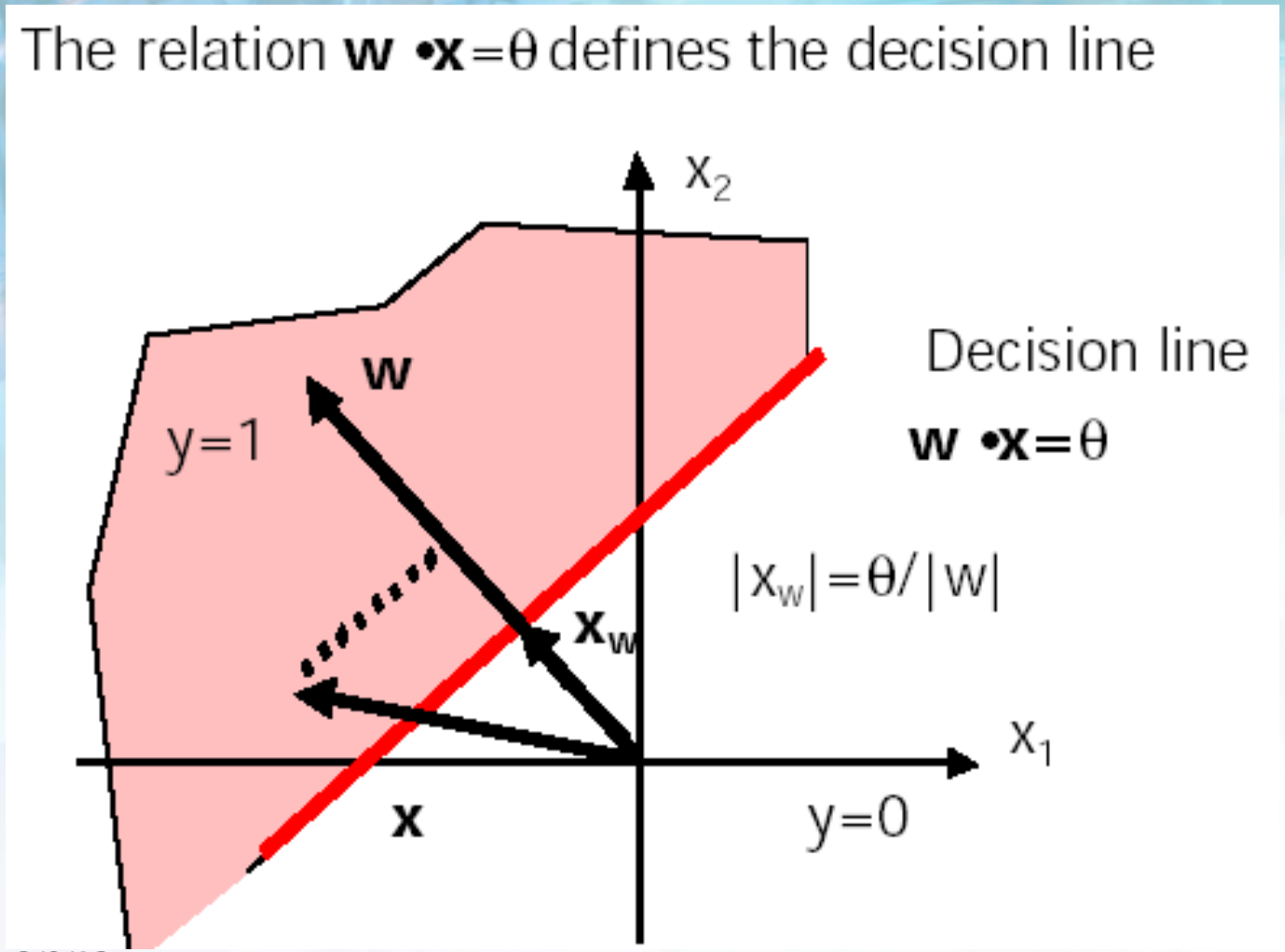
angle < 90 angle = 90 angle > 90



dot product of v & w is the projection of v onto w

Geometric Interpretation

If we examine this in terms of a neuron, x is the vector of the two inputs, w is the vector of the two weights

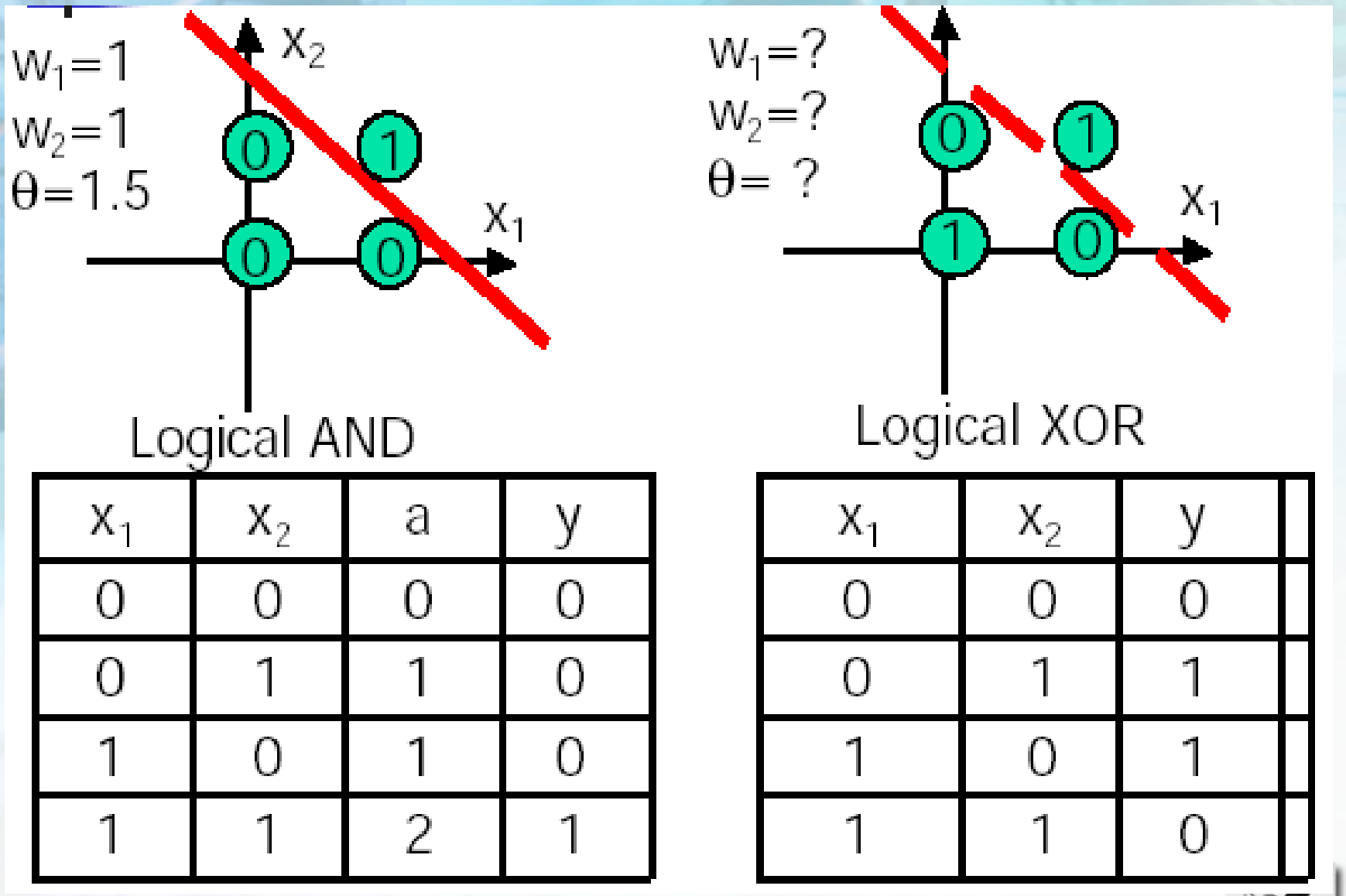


From the dot product, the decision line must be 90 degrees to the weight vector, Xw must be the distance from the origin to the decision line along W

Geometric Interpretation

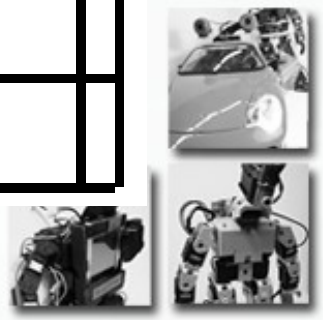
- In n-dimensions, the relationship w·x=theta defines an n-1 dimensional hyperplane, which is perpendicular to w
- On one side of the hyper-plane (w · x > theta), all instances are classified as 1 by the TLU. Those on the other side are classified as 0.
- If patterns can not be separated by a hyper-plane, they can not be recognized by a TLU.
  - Linear Separability

Linear Separability



x_1	x_2	a	y
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



Threshold vs. Weight

- So far, you can see that a threshold is very different from the weights on connections, even though both are adjustable
- They clearly interact geometrically
- Treating this differently makes it difficult to develop and analyze learning algorithms, since a threshold can be changed independently of weights
- We would like to have the threshold treated similarly to all the other weights

Threshold vs. Weight

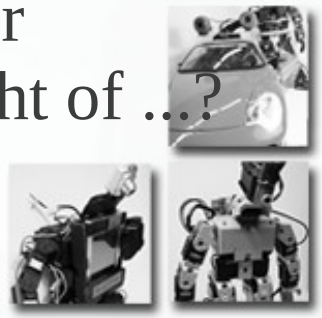
Recall the comparison of weighted inputs to the threshold value:

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

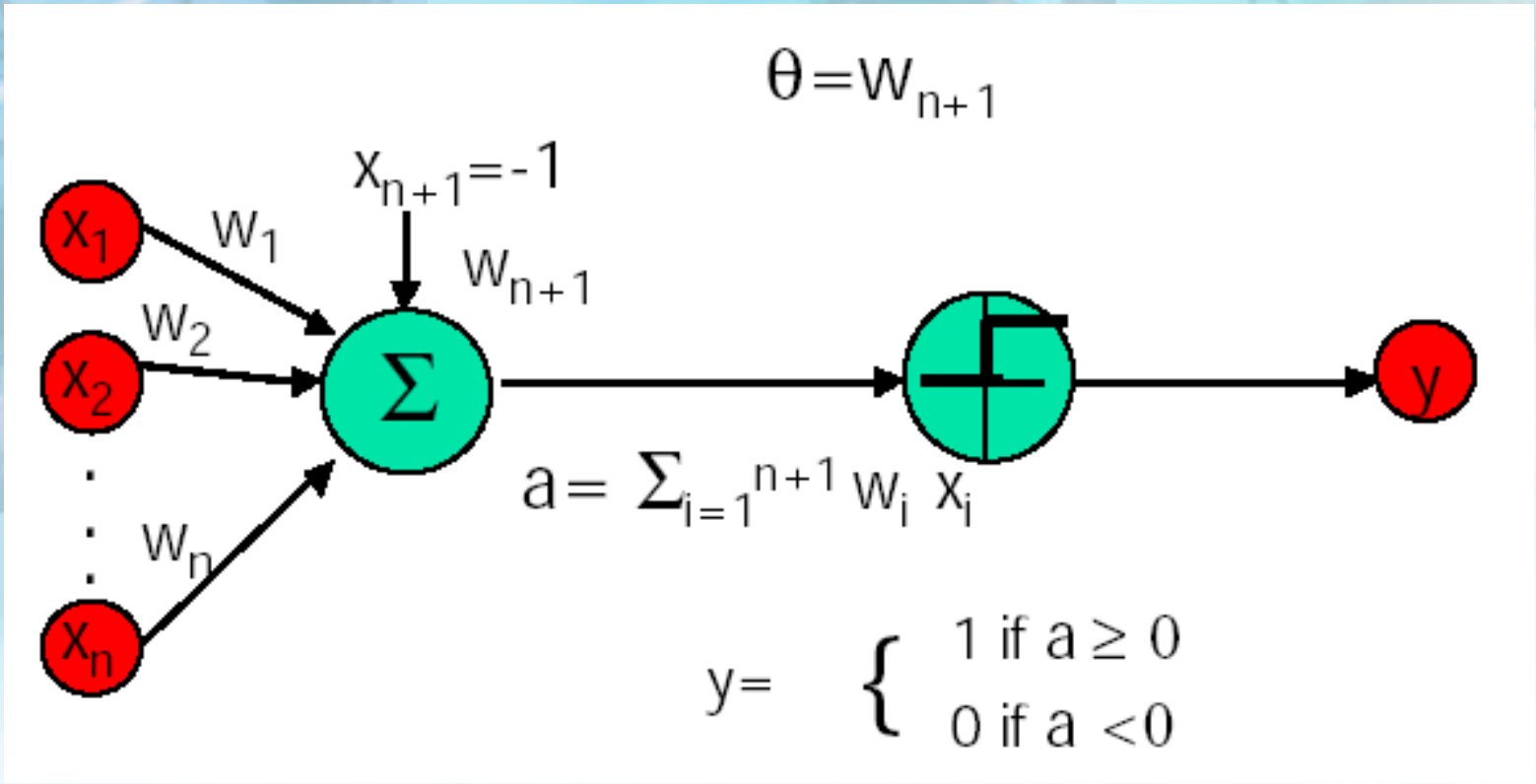
If we move the threshold value to the other side of the equation, we get:

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

- And so we can use a comparison value of 0, and treat our threshold as another input in the summation with a weight of ...?

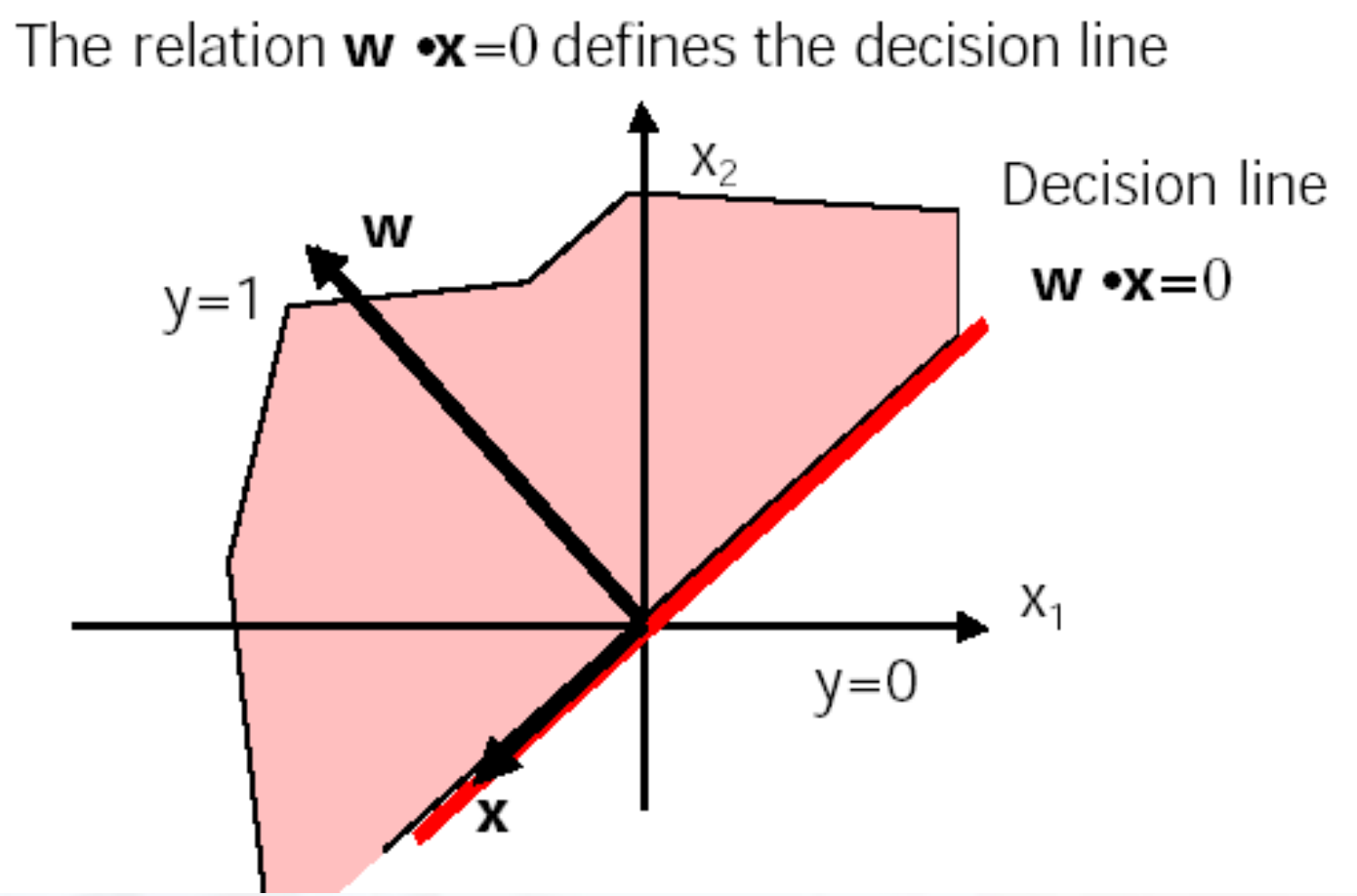


Threshold as Weight



- our summation goes to n+1, with an additional fixed weight of -1 for the threshold value
- Our comparison boundary is now 0

Geometric Interpretation



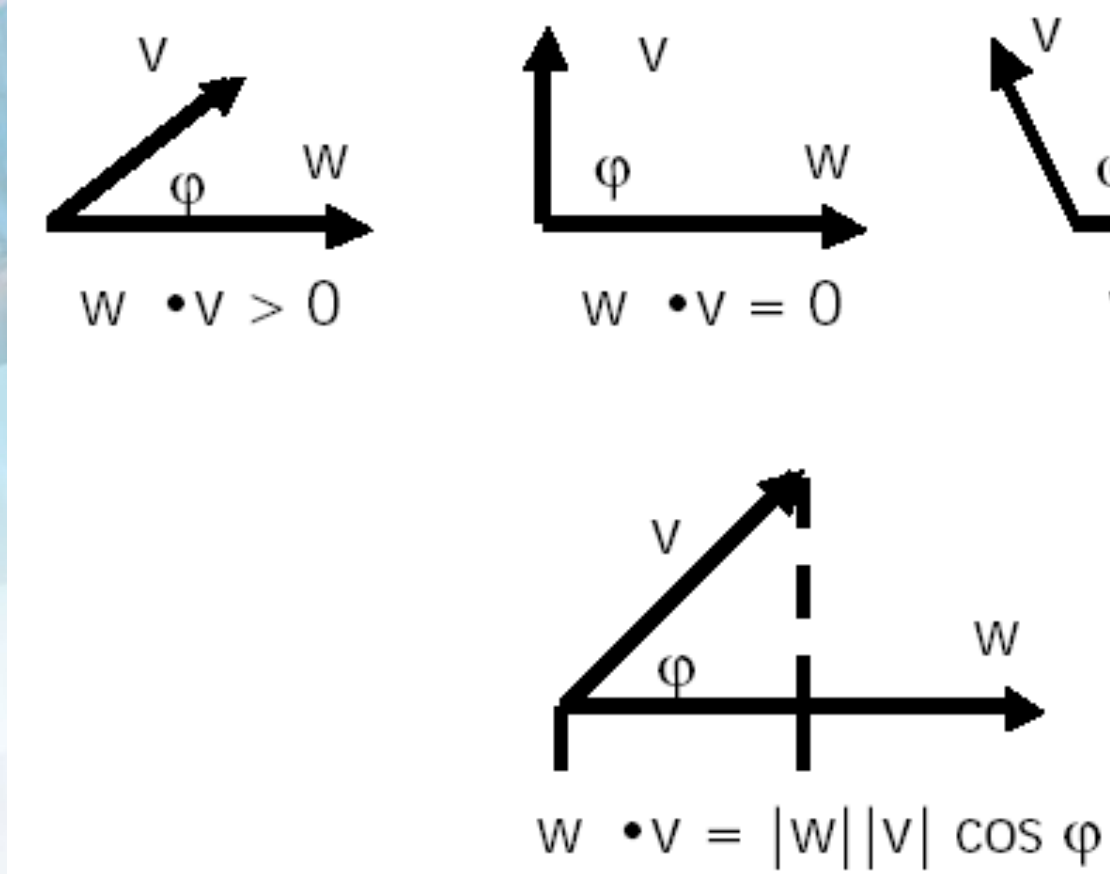
Decision line is now centered on the origin

Scalar Products and Projections

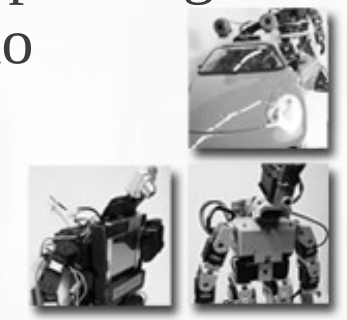
angle < 90
output y =0

angle = 90
output y = 0
(boundary)

angle > 90
output y = 1

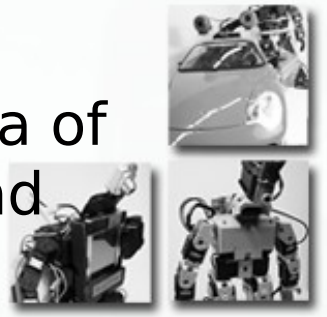


projection will be +ve or –ve depending on rho

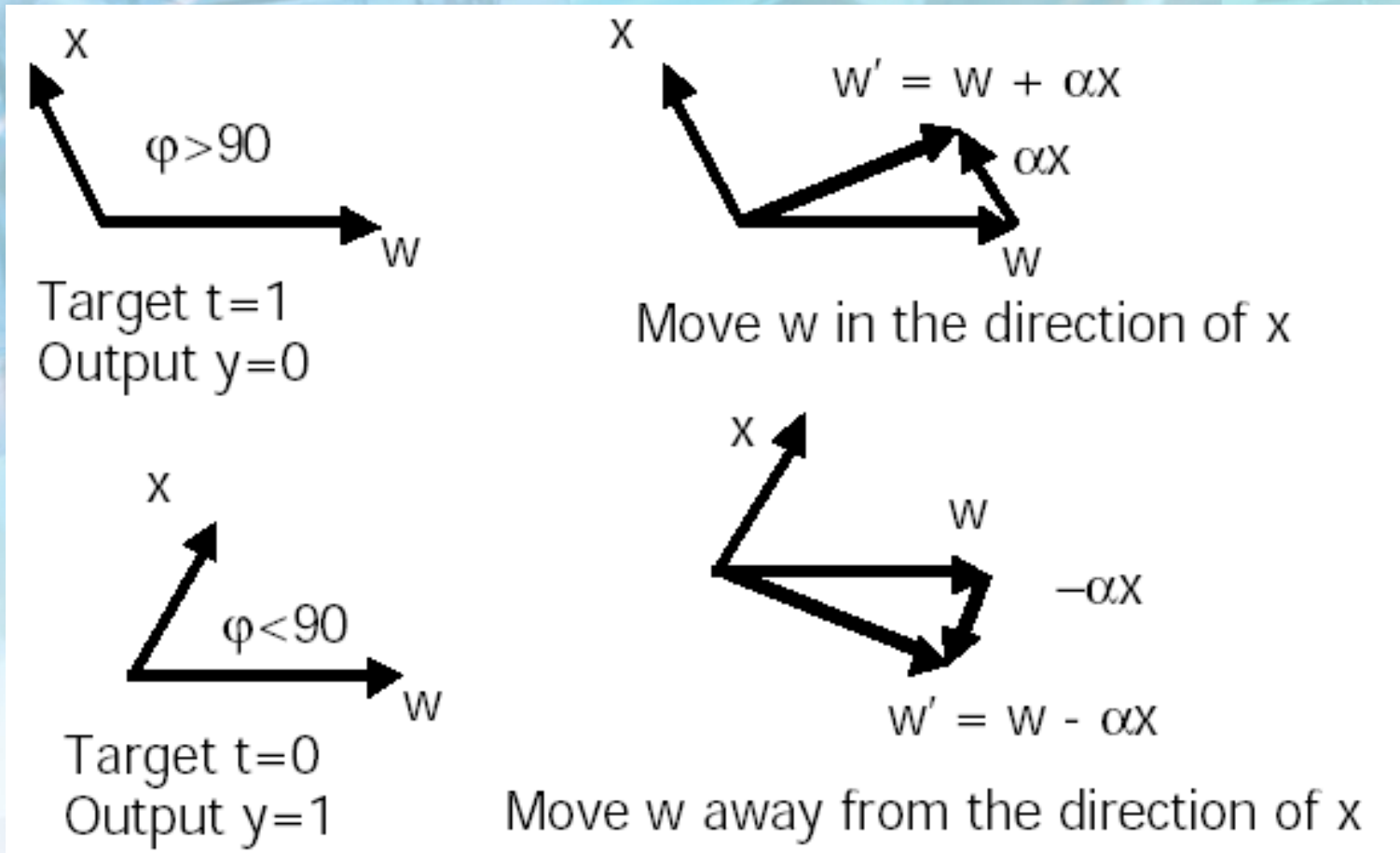


Training ANNs

- Training set of examples {x,t}
  - x is an input vector
  - t is the desired target vector
  - Example: Logical And
    - {<(0,1),0>,<(1,0),0>,<(1,1),1>}
- Iterative process
  - Present a training example x, compute output y.
  - Compare y to target t and compute error
  - Adjust weights and thresholds
- Learning rule
  - How to change the weights w and threshold theta of the network as a function of input x, output y, and target t

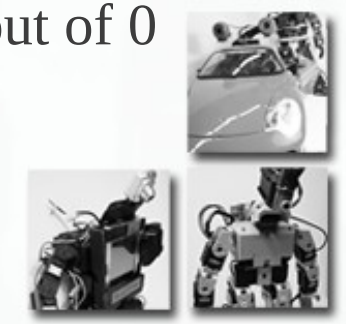


Adjusting the Weight Vector



move w in the direction of x to make angle smaller
repeating will eventually produce output of 1

move w away from the direction of x to make angle larger
repeating will eventually produce output of 0



Perceptron Learning Rule

- $\mathbf{w}' = \mathbf{w} + \alpha\,(t-y)\,\mathbf{x}$

Or in components

- $w'_i = w_i + \Delta w_i = w_i + \alpha\,(t-y)\,x_i \quad (i=1..n+1)$

With $w_{n+1} = \theta$ and $x_{n+1} = -1$

- The parameter α is called the *learning rate*. It determines the magnitude of weight updates Δw_i .
- If the output is correct ($t=y$) the weights are not changed ($\Delta w_i = 0$).
- If the output is incorrect ($t \neq y$) the weights w_i are changed such that the output of the TLU for the new weights w'_i is *closer/further* to the input x_i.

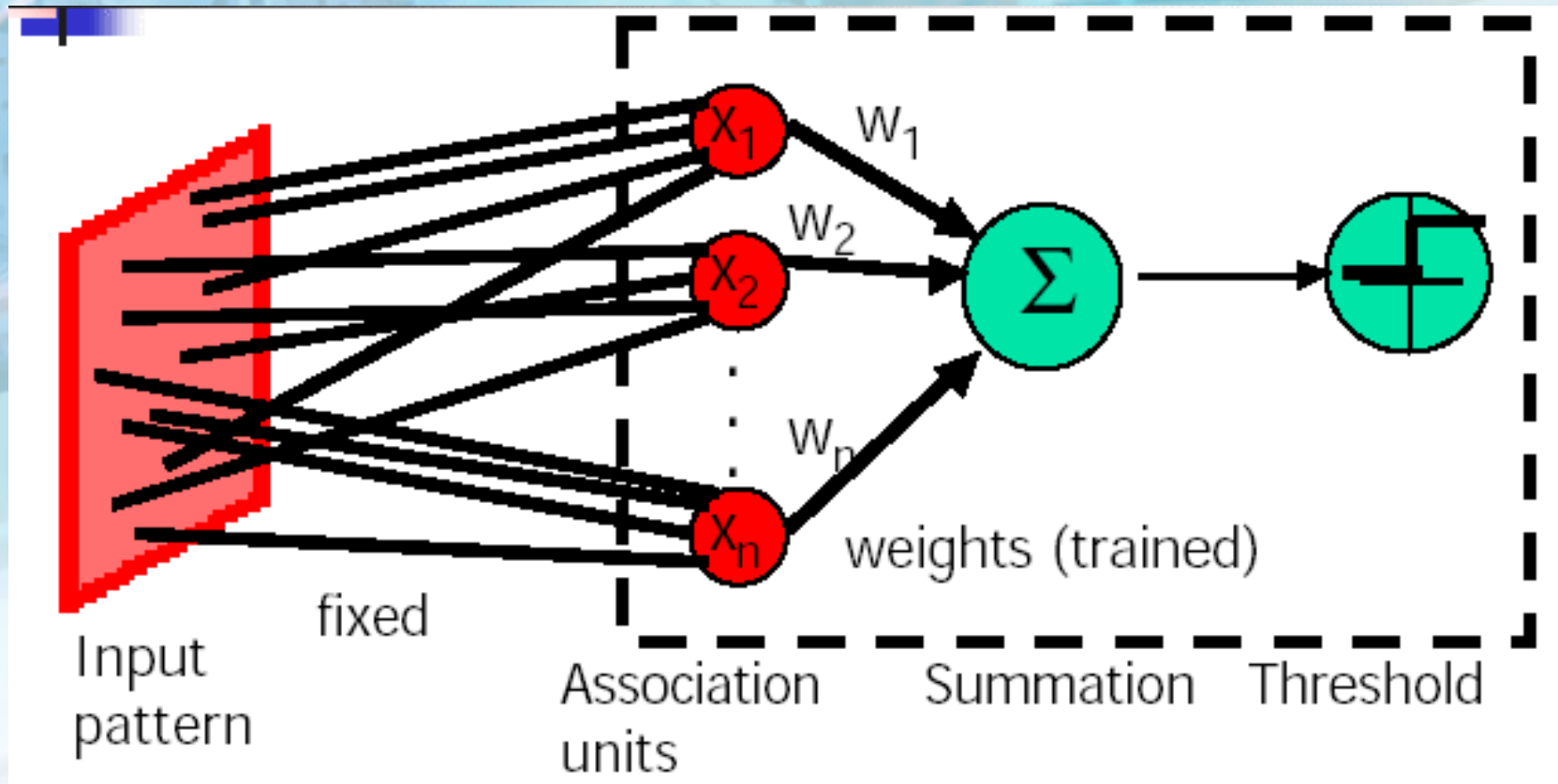
Perceptron Learning Algorithm

Repeat
  for each training vector pair ($\mathbf{x}$,t)
    evaluate the output y when $\mathbf{x}$ is the input
    if y≠t then
      form a new weight vector $\mathbf{w}'$ according to $\mathbf{w}' = \mathbf{w} + \alpha\,(t-y)\,\mathbf{x}$
    else
      do nothing
    end if
  end for
Until y=t for all training vector pairs

Perceptron Convergence Theorem

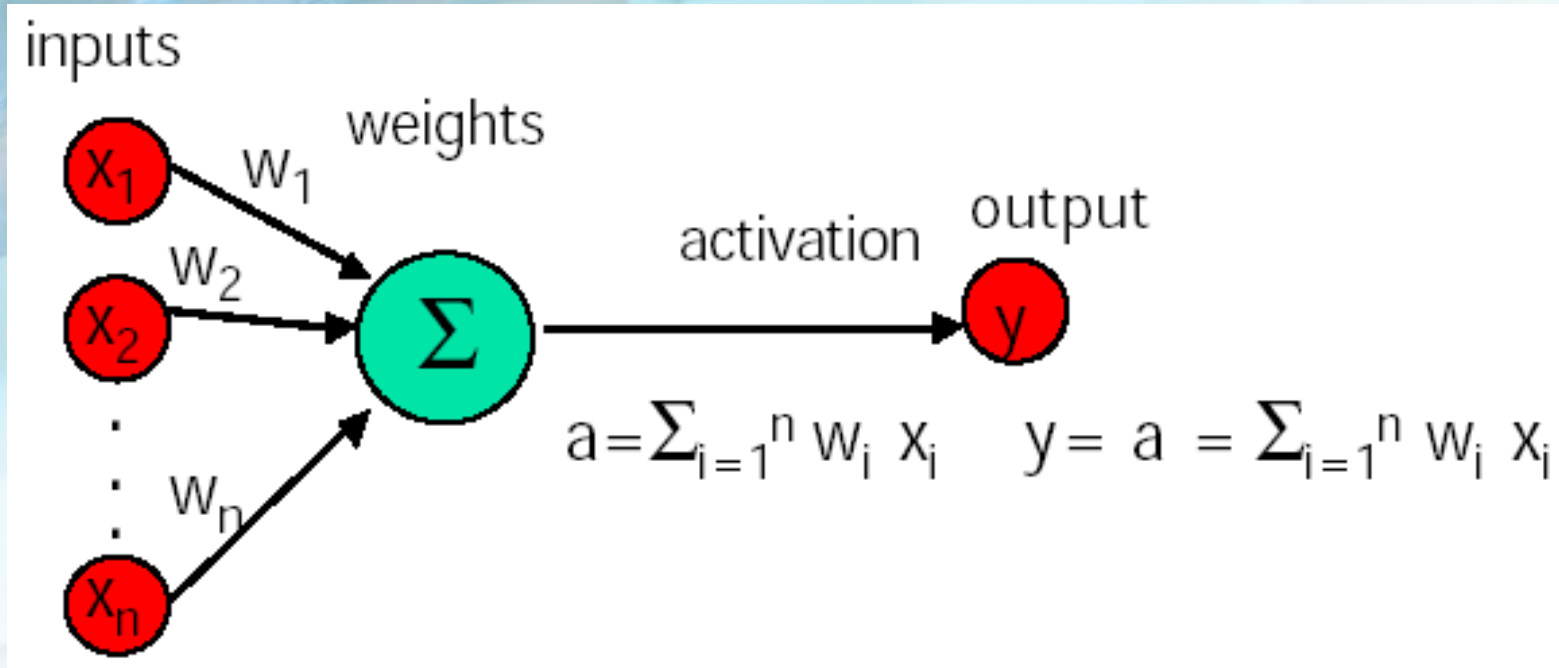
- The algorithm converges to the correct classification
  - if the training data is linearly separable
  - and η is sufficiently small
- If two classes of vectors X_1 and X_2 are linearly separable, the application of the perceptron training algorithm will eventually result in a weight vector $\mathbf{w_0}$, such that $\mathbf{w_0}$ defines a TLU whose decision hyper -plane separates X_1 and X_2 (Rosenblatt 1962).
- Solution $\mathbf{w_0}$ is not unique, since if $\mathbf{w_0}\,\mathbf{x} = 0$ defines a hyper -plane, so does $\mathbf{w'_0} = k\,\mathbf{w_0}$.

Perceptrons vs. TLU



Association units (A-units) can be assigned arbitrary Boolean functions of the input pattern.

Linear Unit


inputs
weights
x_1
w_1
x_2
w_2
w_n
x_n
Σ
activation
output
y
$a = \sum_{i=1}^{n} w_i x_i$
$y = a = \sum_{i=1}^{n} w_i x_i$

Gradient Descent Learning Rule

- Consider linear unit without threshold and continuous output o (not just –1,1)
  - $o = w_0 + w_1 x_1 + \ldots + w_n x_n$
- Train the w_i's such that they minimize the squared error
  - $E[w_1, \ldots, w_n] = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$

  where D is the set of training examples

Gradient Descent

$D=\{<(1,1),1>,<(-1,-1),1>,$
$<(1,-1),-1>,<(-1,1),-1>\}$

Gradient:

$\nabla E[w]=[\partial E/\partial w_0, \ldots \partial E/\partial w_n]$

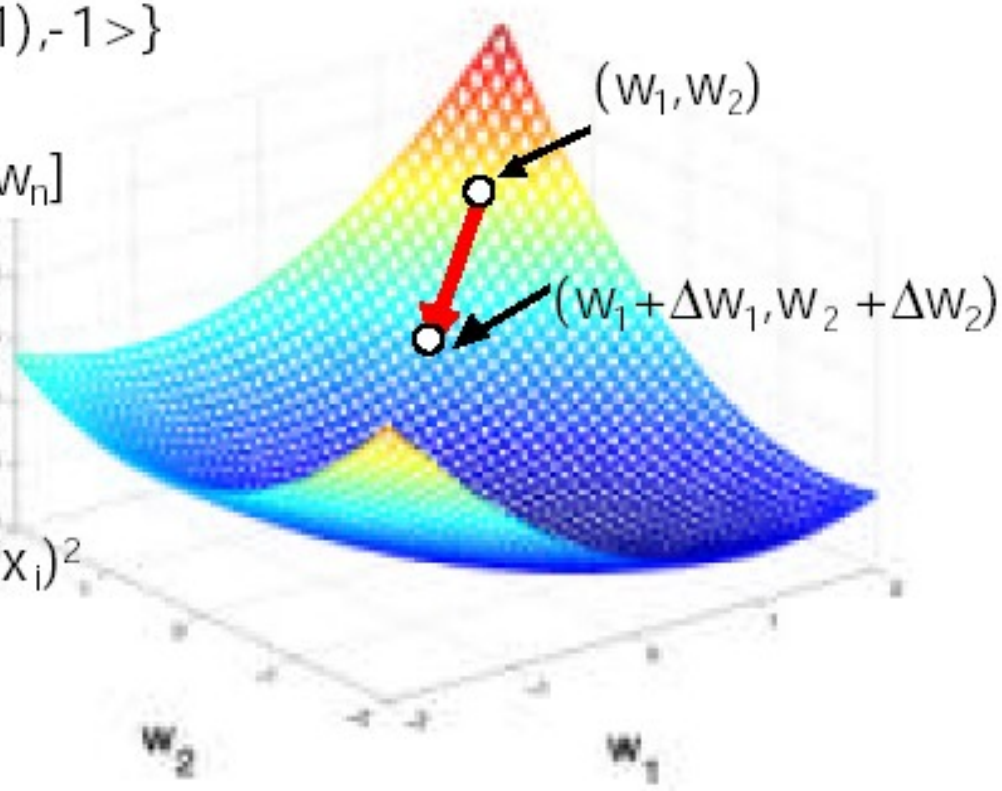
$\Delta w=-\eta \nabla E[w]$

$\Delta w_i=-\eta \partial E/\partial w_i$

$=\partial/\partial w_i \; 1/2\Sigma_d(t_d-o_d)^2$

$= \partial/\partial w_i \; 1/2\Sigma_d(t_d-\Sigma_i w_i x_i)^2$

$= \Sigma_d(t_d- o_d)(-x_i)$



Gradient Descent Training Algorithm

Gradient-Descent(*training_examples*, η)

Each training example is a pair of the form $<(x_1,...x_n),t>$ where $(x_1,...,x_n)$ is the vector of input values, and t is the target output value, η is the learning rate (e.g. 0.1)

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero
  - For each $<(x_1,...x_n),t>$ in *training_examples* Do
    - Input the instance $(x_1,...,x_n)$ to the linear unit and compute the output o
    - For each linear unit weight w_i Do
      - $\Delta w_i = \Delta w_i + \eta\ (t-o)\ x_i$
  - For each linear unit weight wi Do
    - $w_i = w_i + \Delta w_i$

Incremental Stochastic Gradient Descent

- Batch mode : gradient descent

  $w=w - \eta \nabla E_D[w]$ over the entire data D

  $E_D[w]=1/2\Sigma_d(t_d-o_d)^2$

- Incremental mode: gradient descent

  $w=w - \eta \nabla E_d[w]$ over individual training examples d

  $E_d[w]=1/2 (t_d-o_d)^2$

Incremental Gradient Descent can approximate Batch Gradient Descent arbitrarily closely if η is small enough

Perceptron vs. Gradient Descent Rule

- perceptron rule

  $w'_i = w_i + \alpha\ (t^p - y^p)\ x_i^p$

  derived from manipulation of decision surface.
- gradient descent rule

  $w'_i = w_i + \alpha\ (t^p - y^p)\ x_i^p$

  derived from minimization of error function

  $E[w_1,\ldots,w_n] = \frac{1}{2} \sum_p (t^p - y^p)^2$

  by means of gradient descent.

Perceptron vs. Gradient Descent Rule

Perceptron learning rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rules uses gradient descent

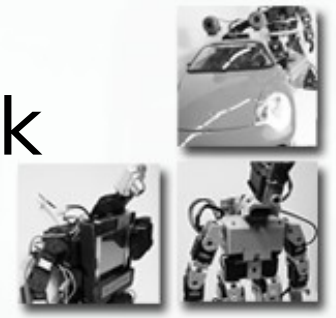
- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Presentation of Training Examples

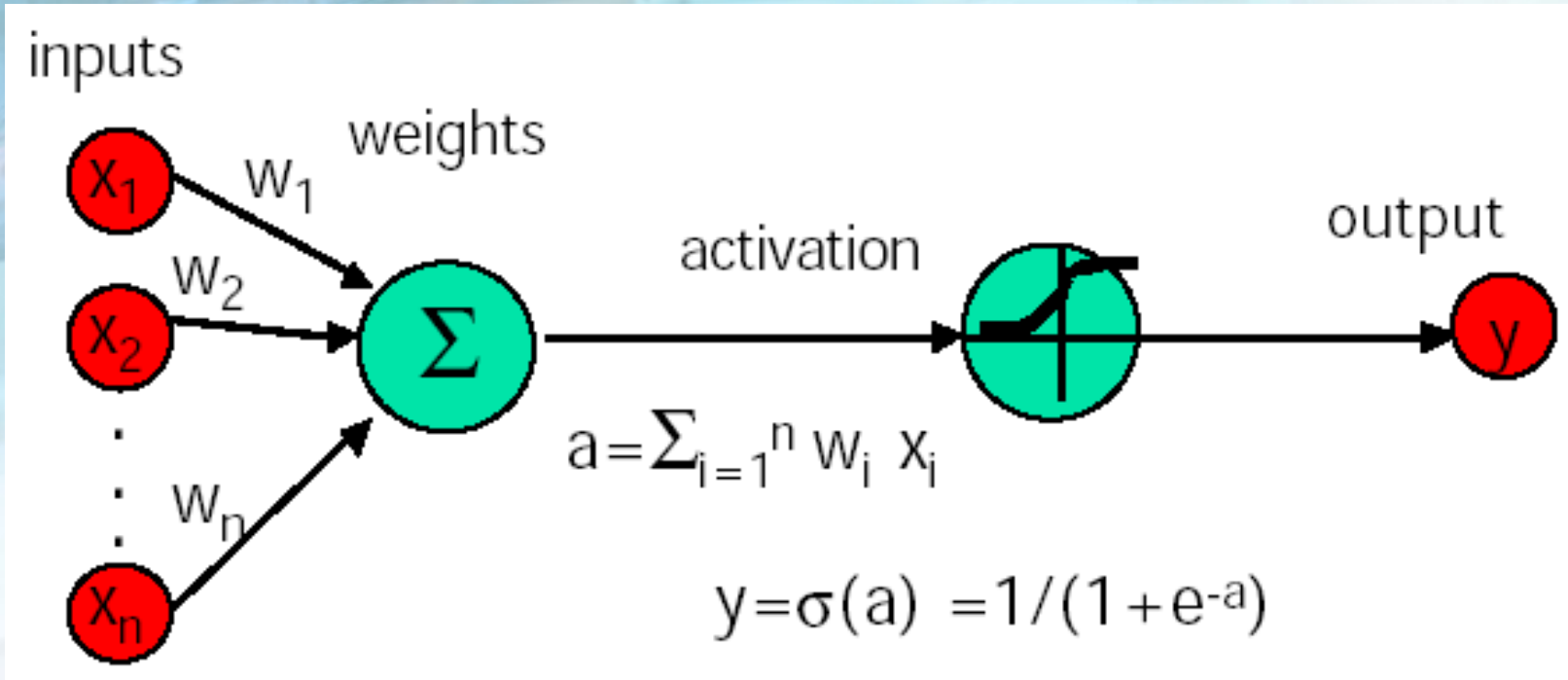
- Presenting all training examples once to the ANN is called an *epoch*.
- In incremental stochastic gradient descent training examples can be presented in
  - Fixed order (1,2,3...,M)
  - Randomly permutated order (5,2,7,...,3)
  - Completely random (4,1,7,1,5,4,.....)

Multi-level ANN with linear activation functions

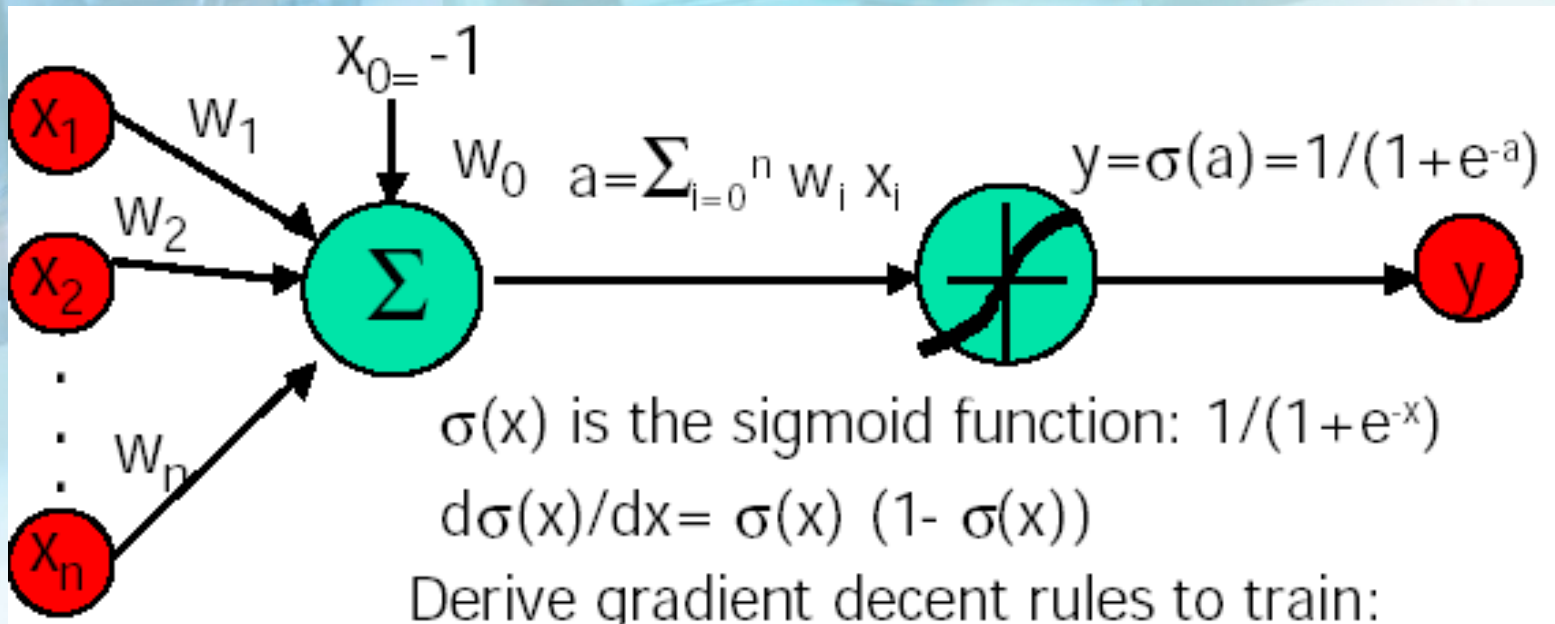
- To overcome the problem of requiring linear seperability, researchers proposed to use several layers of networks
- Networks of neurons with linear activation functions
  - y11 = x1*w11+x2*w12+…
  - y21 = y1*w21+y2*w22+…
  - y21=w21*(w11*x1+w12*x2….)
  - y21=(w21*w11)*x1+(w21*w12)*x2
- are equivalent to a single layer network



Neuron with Sigmoid Function



Neuron with Sigmoid Unit



$\sigma(x)$ is the sigmoid function: $1/(1+e^{-x})$

$d\sigma(x)/dx = \sigma(x)\ (1-\sigma(x))$

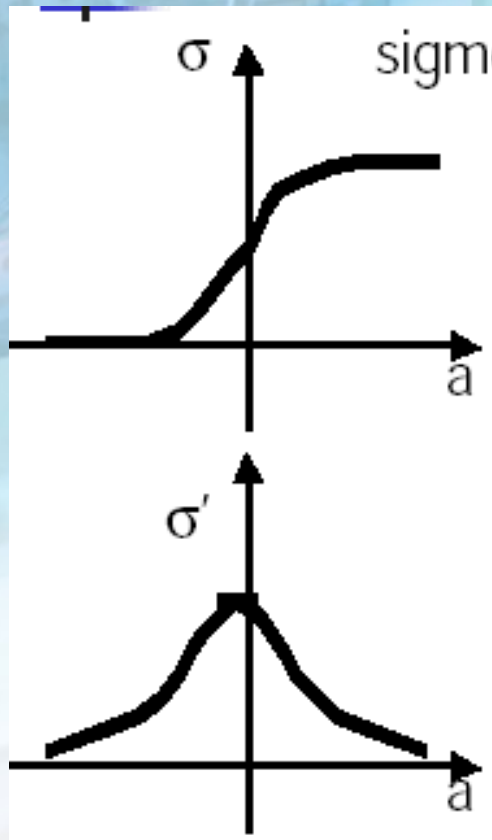
Derive gradient decent rules to train:

- one sigmoid function

$$\partial E/\partial w_i = -\sum_p (t^p - y)\ y\ (1-y)\ x_i^p$$

- Multilayer networks of sigmoid units backpropagation:

Gradient Descent Rule for Sigmoid Output Function



$$E^p[w_1,\ldots,w_n] = \tfrac{1}{2}\,(t^p - y^p)^2$$

$$\partial E^p/\partial w_i = \partial/\partial w_i\ \tfrac{1}{2}\,(t^p - y^p)^2$$

$$= \partial/\partial w_i\ \tfrac{1}{2}\,(t^p - \sigma(\textstyle\sum_i w_i\, x_i^p))^2$$

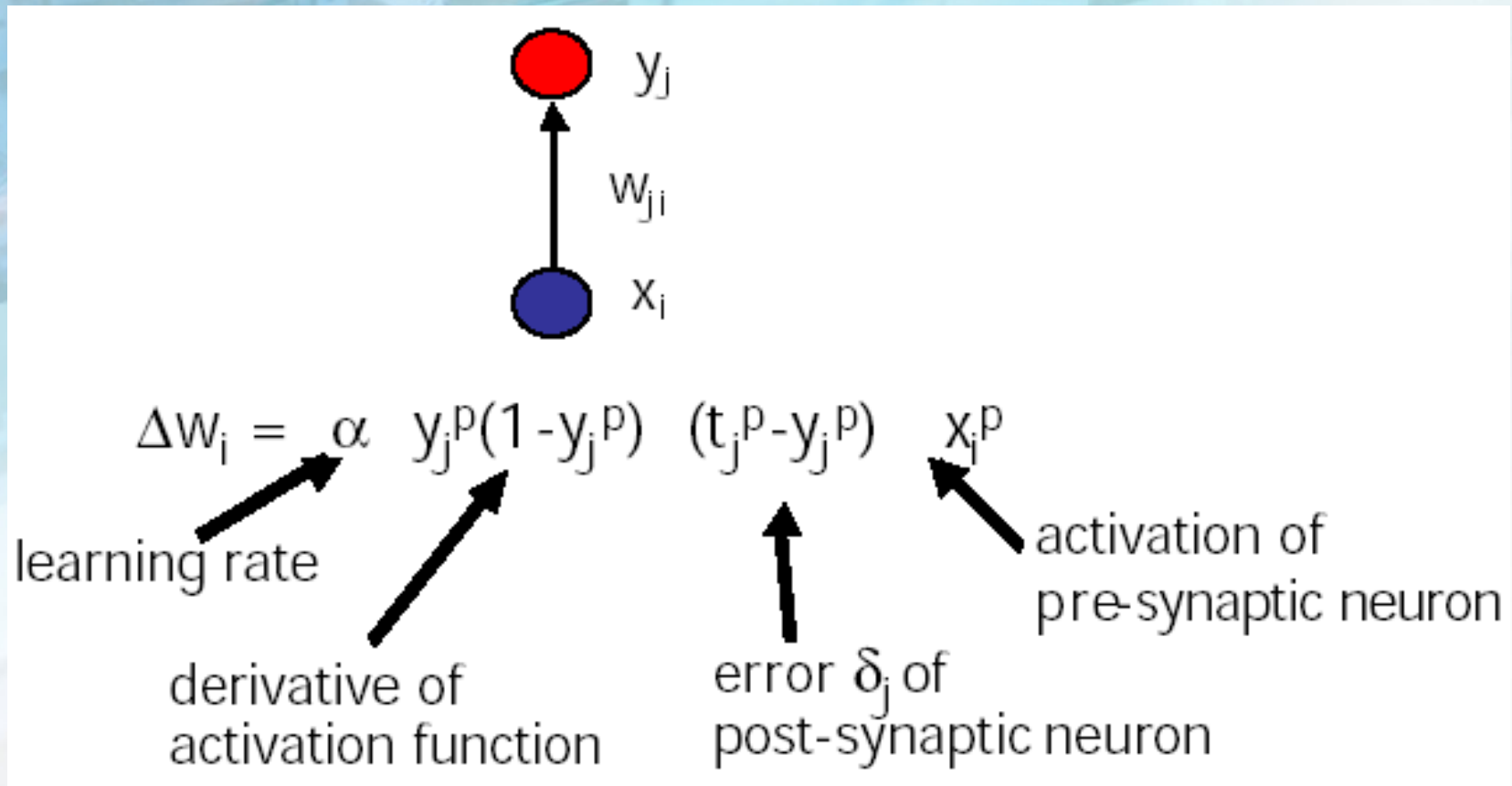
$$= (t^p - y^p)\,\sigma'(\textstyle\sum_i w_i\, x_i^p)\,(-x_i^p)$$

for $y=\sigma(a) = 1/(1+e^{-a})$

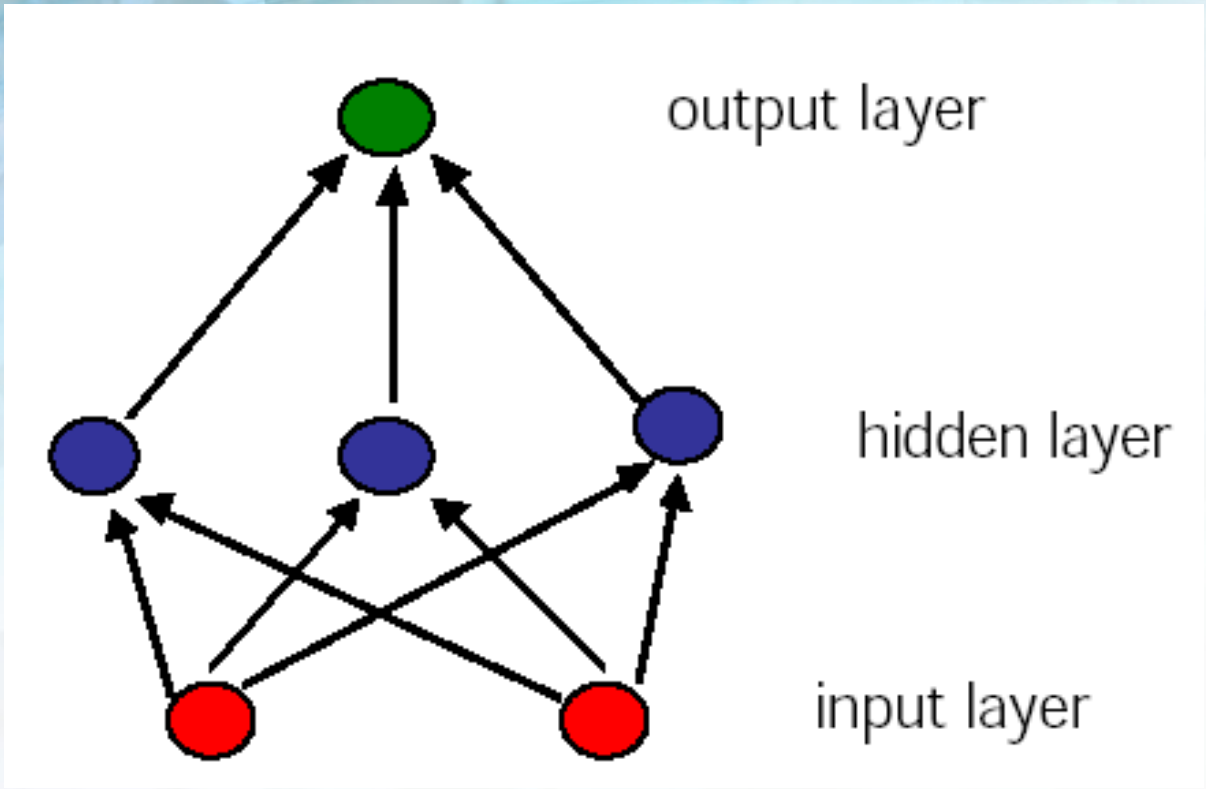
$\sigma'(a) = e^{-a}/(1+e^{-a})^2 = \sigma(a)\,(1-\sigma(a))$

$$w'_i = w_i + \Delta w_i = w_i + \alpha\ y(1-y)(t^p - y^p)\ x_i^p$$

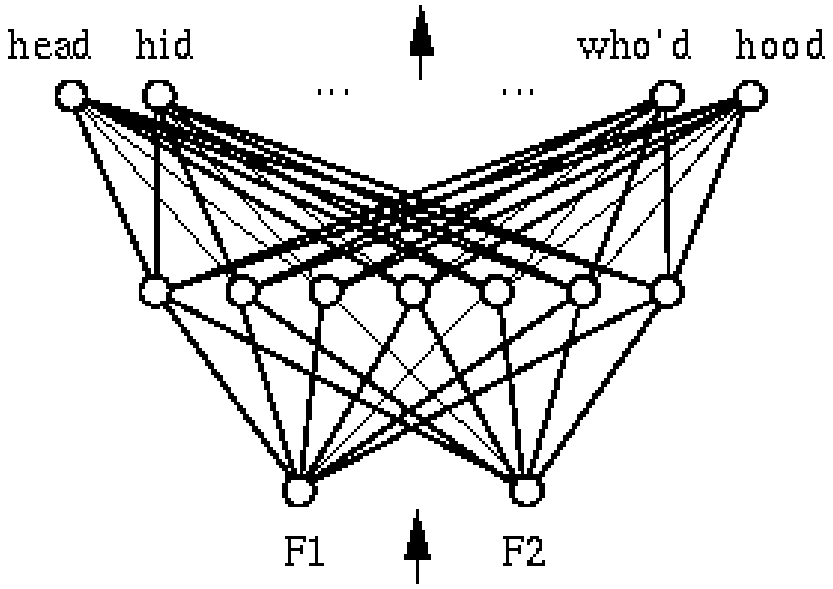
Gradient Descent Learning Rule

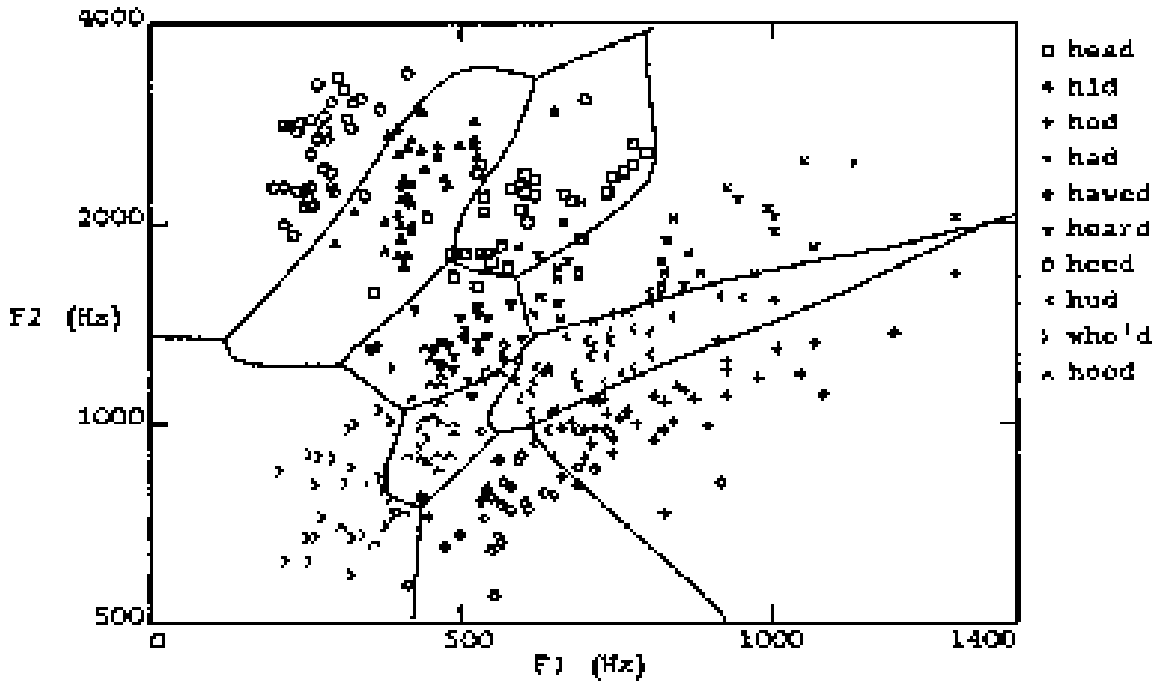


Multilayer Networks

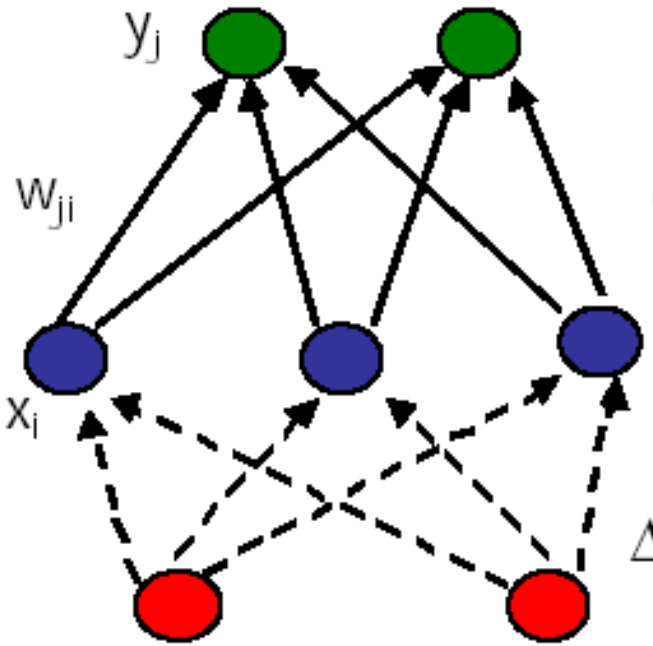


Multilayer Networks of Sigmoid Units





Training Rule for Weights to the Output Layer



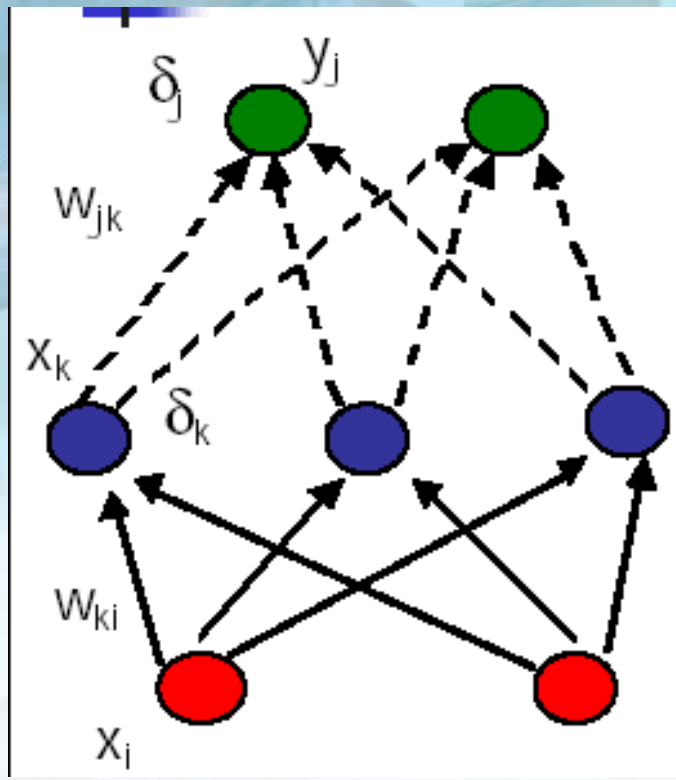
$$E^p[w_{ij}] = \tfrac{1}{2} \sum_j (t_j^p - y_j^p)^2$$

$$\partial E^p / \partial w_{ij} = \partial / \partial w_{ij} \; \tfrac{1}{2} \sum_j (t_j^p - y_j^p)^2$$
$$= \ldots$$
$$= - \, y_j^p (1 - y_j^p)(t_j^p - y_j^p) \; x_i^p$$

$$\Delta w_{ij} = \alpha \; y_j^p (1 - y_j^p) \; (t_j^p - y_j^p) \; x_i^p$$
$$= \alpha \; \delta_j^p \; x_i^p$$

with $\delta_j^p := y_j^p (1 - y_j^p) \; (t_j^p - y_j^p)$

Training Rule for Weights to the Hidden Layer



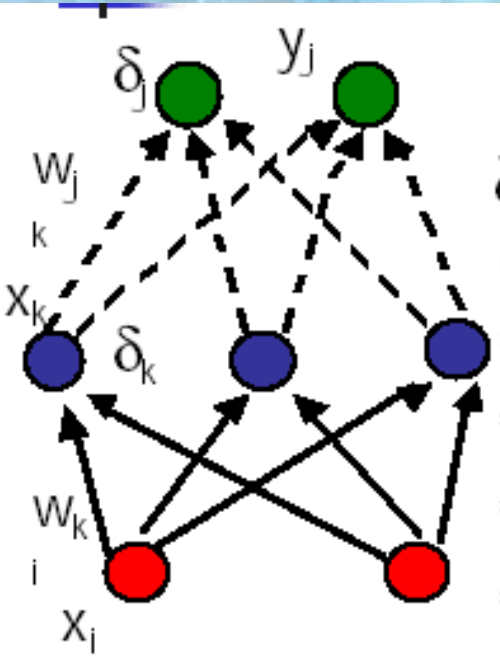
Credit assignment problem:
No target values t for hidden layer units.

Error for hidden units?

$$\delta_k = \Sigma_j \, w_{jk} \, \delta_j \, y_j \, (1-y_j)$$

$$\Delta w_{ki} = \alpha \; x_k^p (1 - x_k^p) \; \delta_k^p \; x_i^p$$

Training Rule for Weights to the Hidden Layer



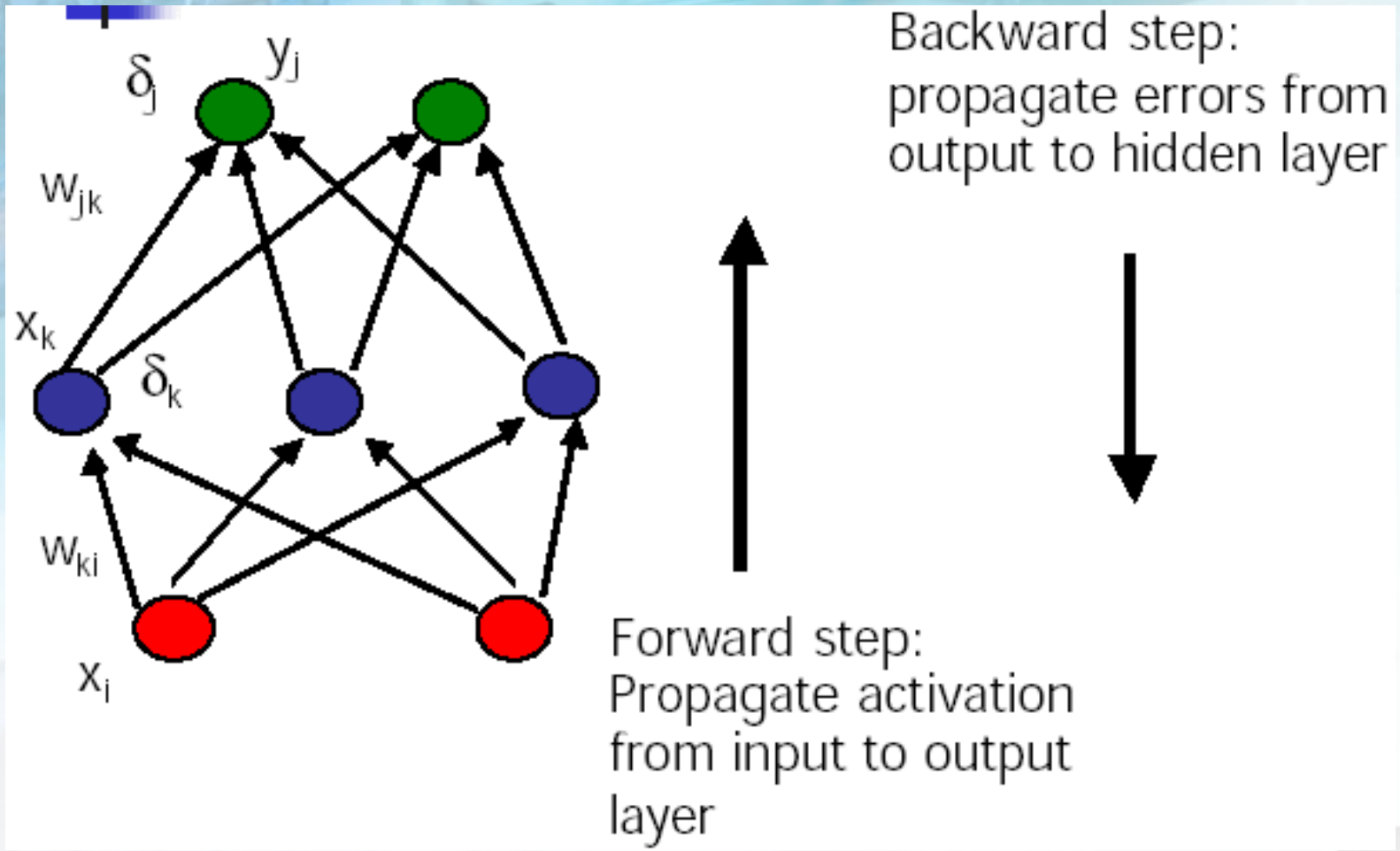
$$E^p[w_{ki}] = \tfrac{1}{2} \sum_j (t_j^p - y_j^p)^2$$

$$\partial E^p/\partial w_{ki} = \partial/\partial w_{ki}\ \tfrac{1}{2} \sum_j (t_j^p - y_j^p)^2$$
$$= \partial/\partial w_{ki}\ \tfrac{1}{2} \sum_j (t_j^p - \sigma(\sum_k w_{jk}\ x_k^p))^2$$
$$= \partial/\partial w_{ki}\ \tfrac{1}{2} \sum_j (t_j^p - \sigma(\sum_k w_{jk}\ \sigma(\sum_i w_{ki}\ x_i^p)))^2$$
$$= -\sum_j (t_j^p - y_j^p)\ \sigma'_j(a)\ w_{jk}\ \sigma'_k(a)\ x_i^p$$
$$= -\sum_j \delta_j\ w_{jk}\ \sigma'_k(a)\ x_i^p$$
$$= -\sum_j \delta_j\ w_{jk}\ x_k\ (1 - x_k)\ x_i^p$$

$$\Delta w_{ki} = \alpha\ \delta_k\ x_i^p$$

with $\delta_k = \sum_j \delta_j\ w_{jk}\ x_k(1 - x_k)$

Backpropagation

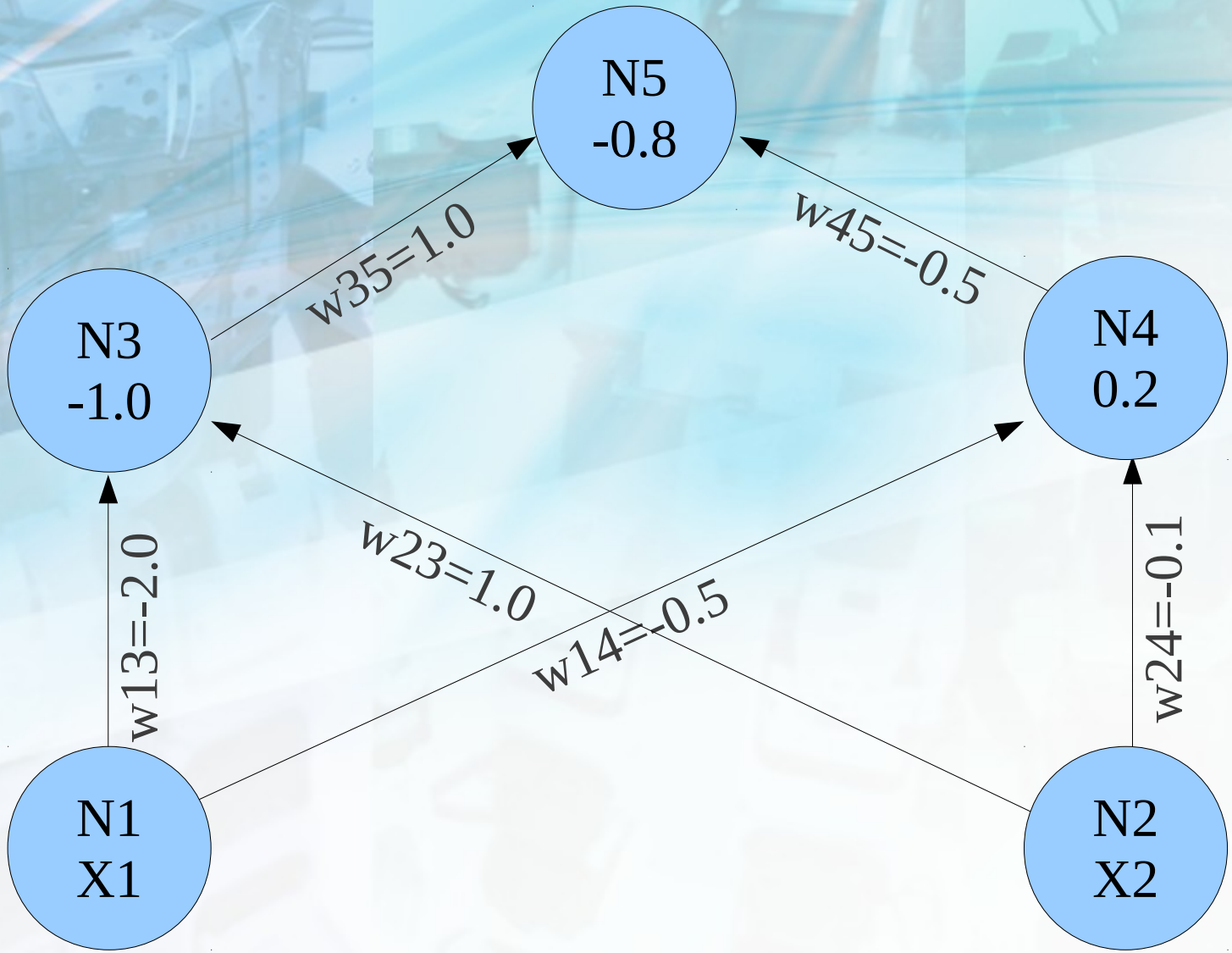


Backpropagation Algorithm

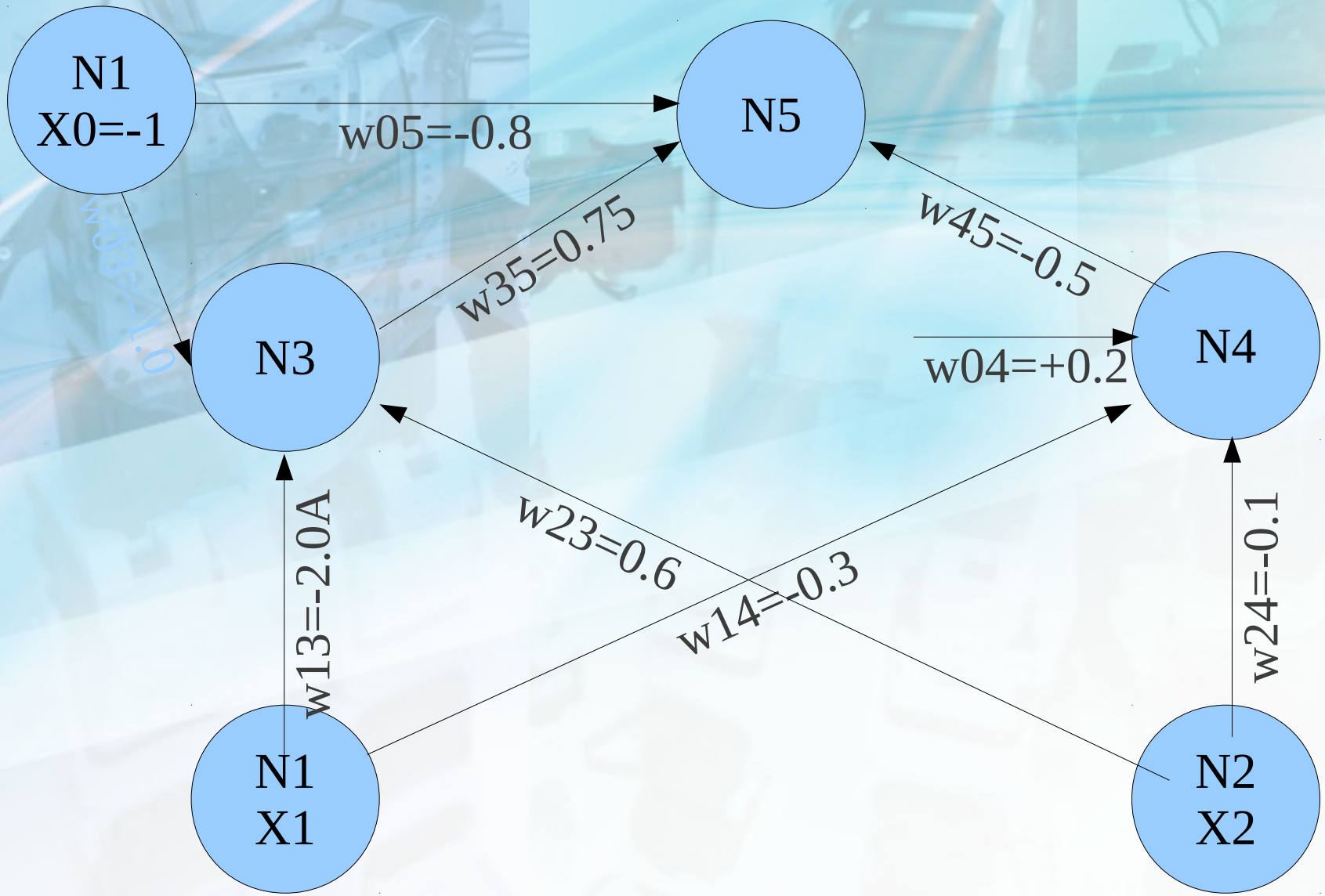
- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - For each training example $<(x_1,\ldots x_n),t>$ Do
    - Input the instance $(x_1,\ldots,x_n)$ to the network and compute the network outputs y_k
    - For each output unit k
      - $\delta_k=y_k(1-y_k)(t_k-y_k)$
    - For each hidden unit h
      - $\delta_h=y_h(1-y_h)\ \sum_k w_{h,k}\ \delta_k$
    - For each network weight $w_{i,j}$ Do
    - $w_{i,j}=w_{i,j}+\Delta w_{i,j}$ where

      $\Delta w_{i,j}=\eta\ \delta_j\ x_{i,j}$

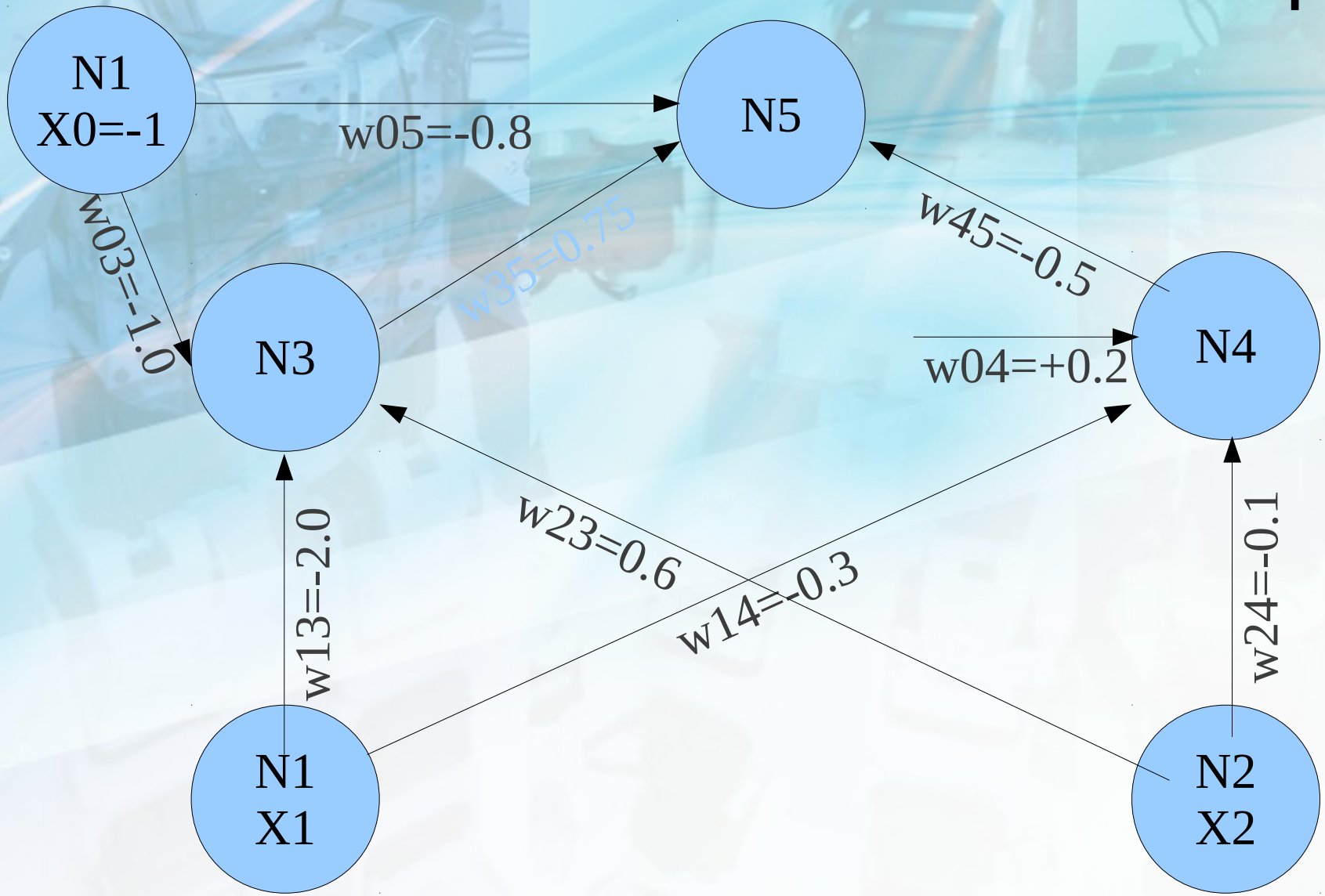
Backpropagation Example



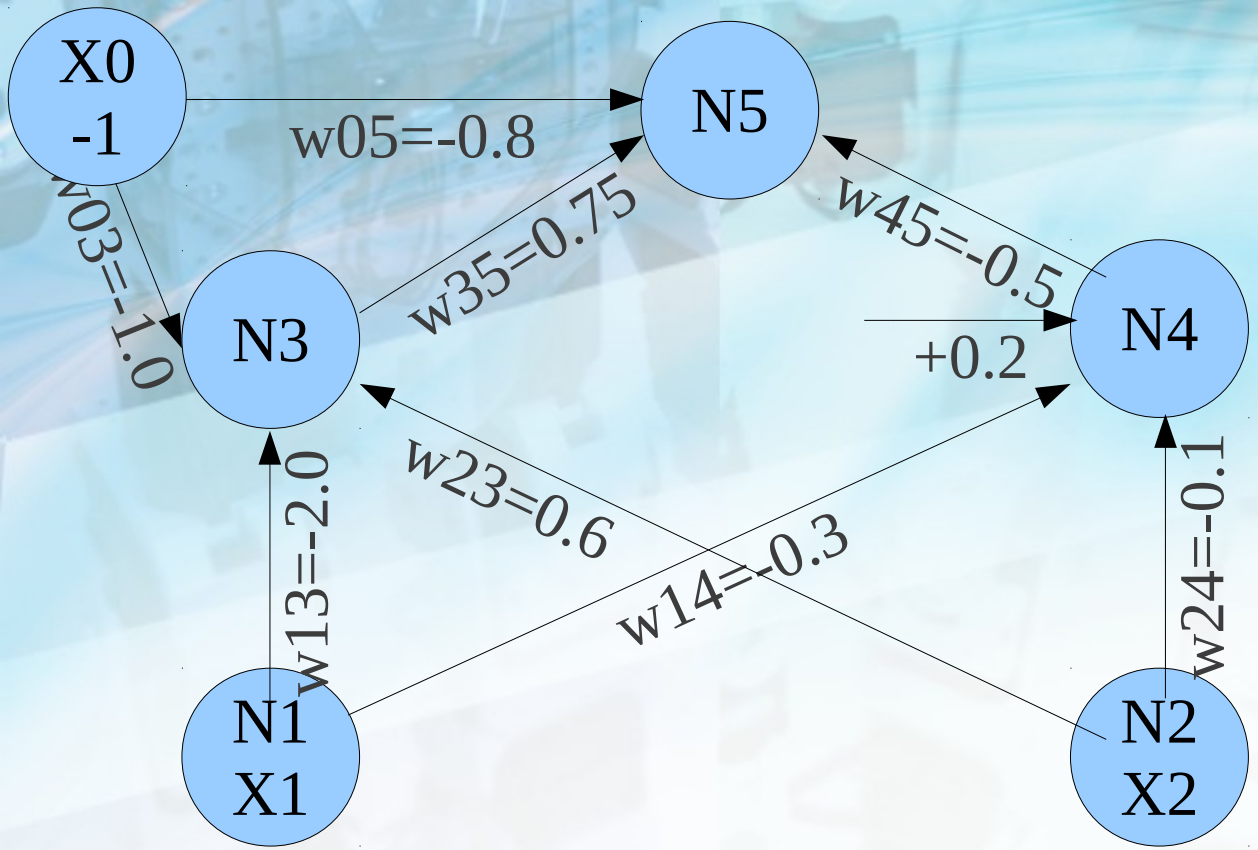
Backpropagation Example



Backpropagation Example: Calculate Activation and Output



Backpropagation Example: Calculate Activation and Output



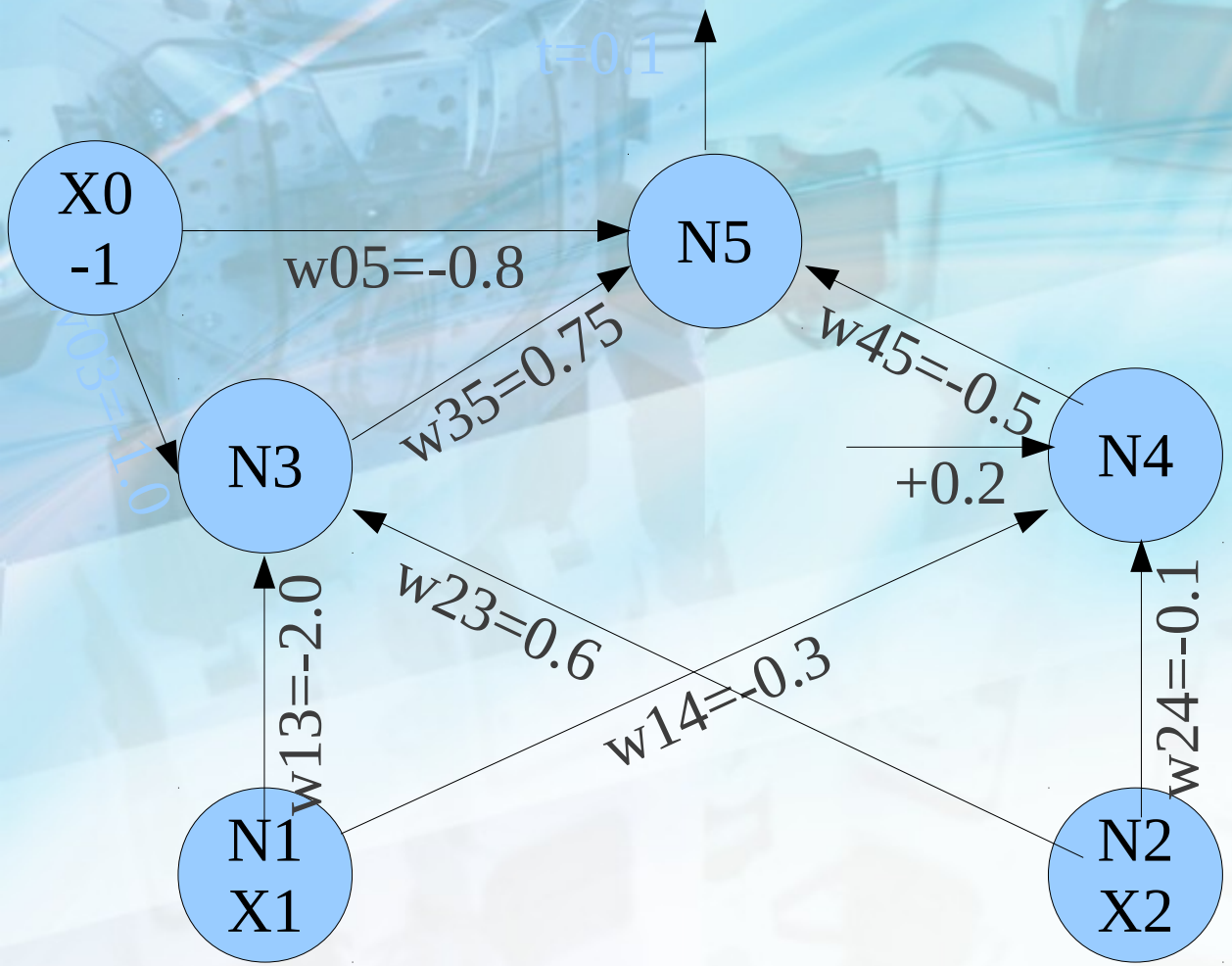
Training:<1,0>, 0.1

a(N3)=-1*-1+1*-2+0*0.6=-1.0
y(N3) = 1/(1+e^(-a))=0.27

a(N4)=-0.2+-0.3=-0.5
y(N4)=0.37

a(N5)=0.8+0.75*0.27
+(-0.5)*0.37=0.82
y(N5)=0.69

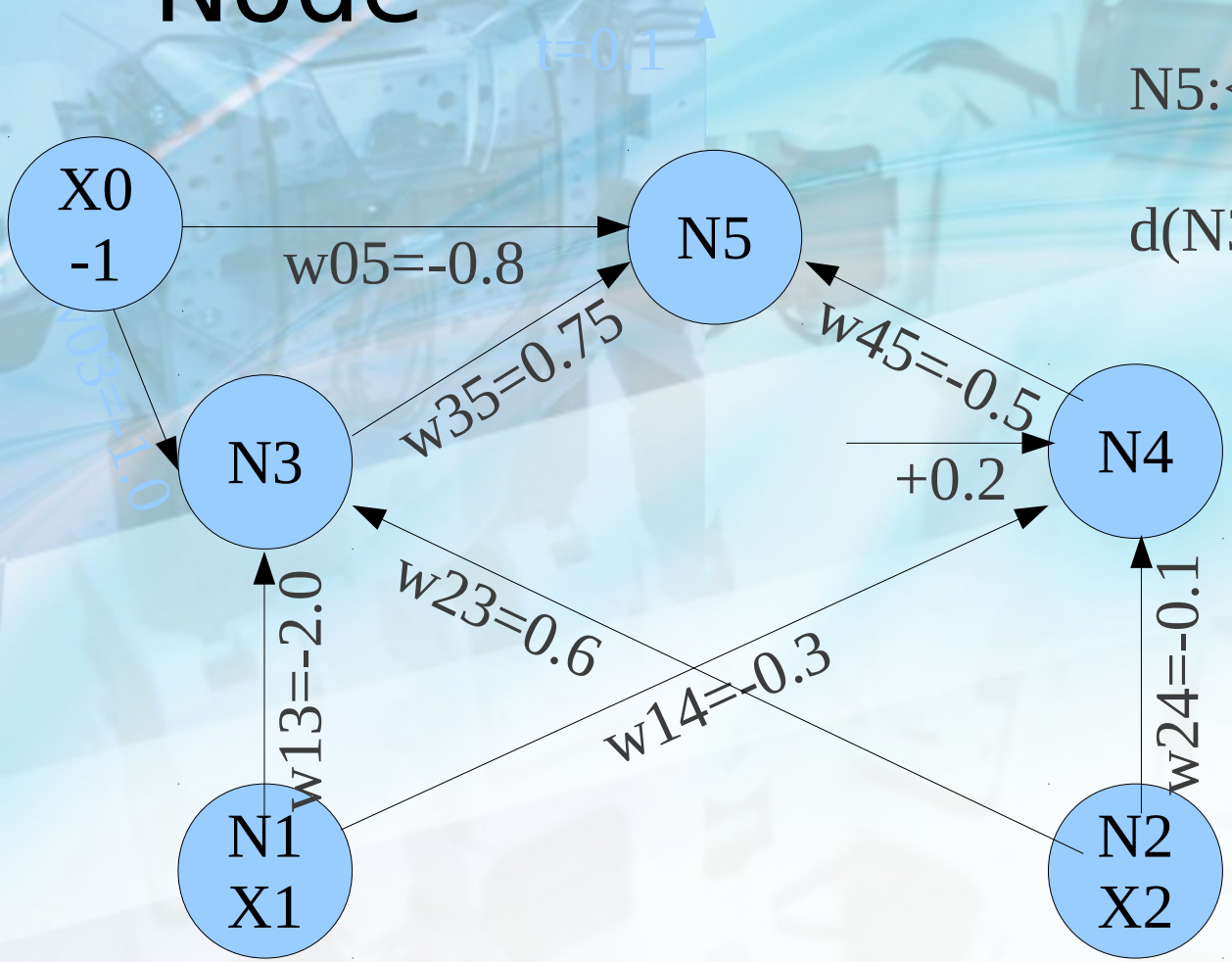
Backpropagation Example: Calculate delta_k for Output



N5:<1,0>, 0.1, alpha=0.1

d(N5) = y_k(1-y_k)(t_k-y_k)
= 0.69(1-0.69)(0.1-0.69)
= -0.13

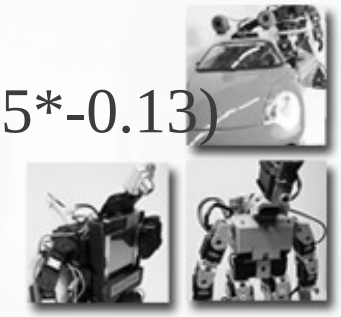
Backpropagation Example: Calculate delta_k for Hidden Node



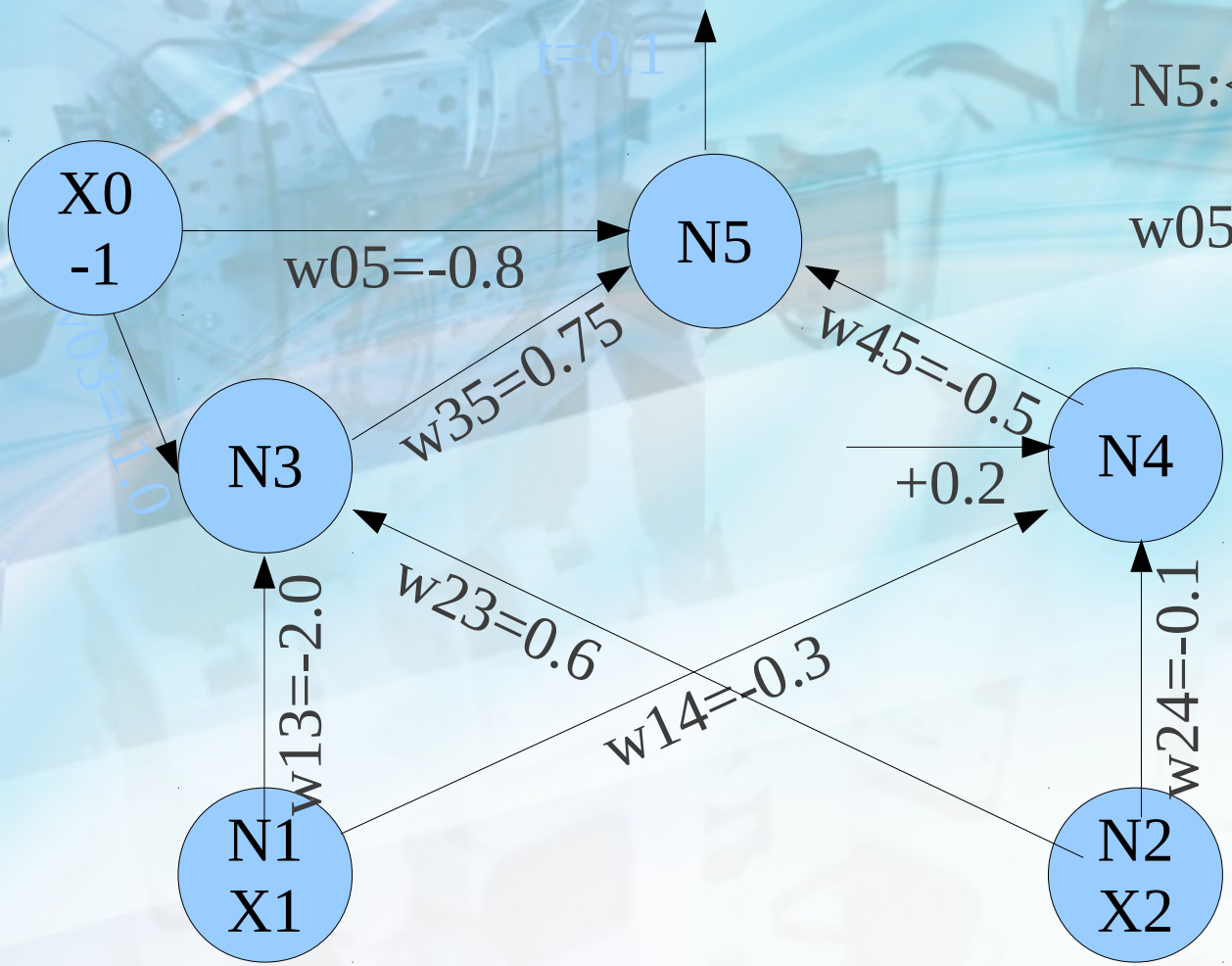
N5:<1,0>, 0.1, alpha=0.1

d(N3) = y_k(1-y_k)sum(w_h,k*d_k)
= 0.27(1-0.27)(0.75*-0.13)
= -0.02

d(N4)=0.37(1-0.37)(-0.5*-0.13)
=0.01



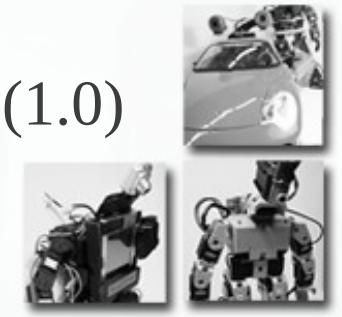
Backpropagation Example: Weight Update



N5:<1,0>, 0.1, alpha=0.1

w05=w05+alpha*delta_5*x_0
= -0.8+0.1*(-0.13)*(-1.0)
= -0.79

w13=-2.0+0.1*(-0.02)*(1.0)
= -2.002

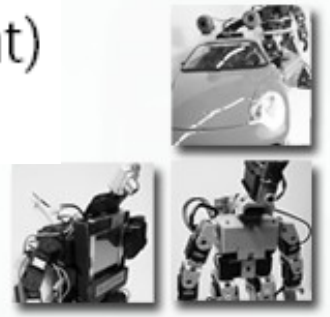


Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  -in practice often works well (can be invoked multiple times with different initial weights)
- Often include weight *momentum* term

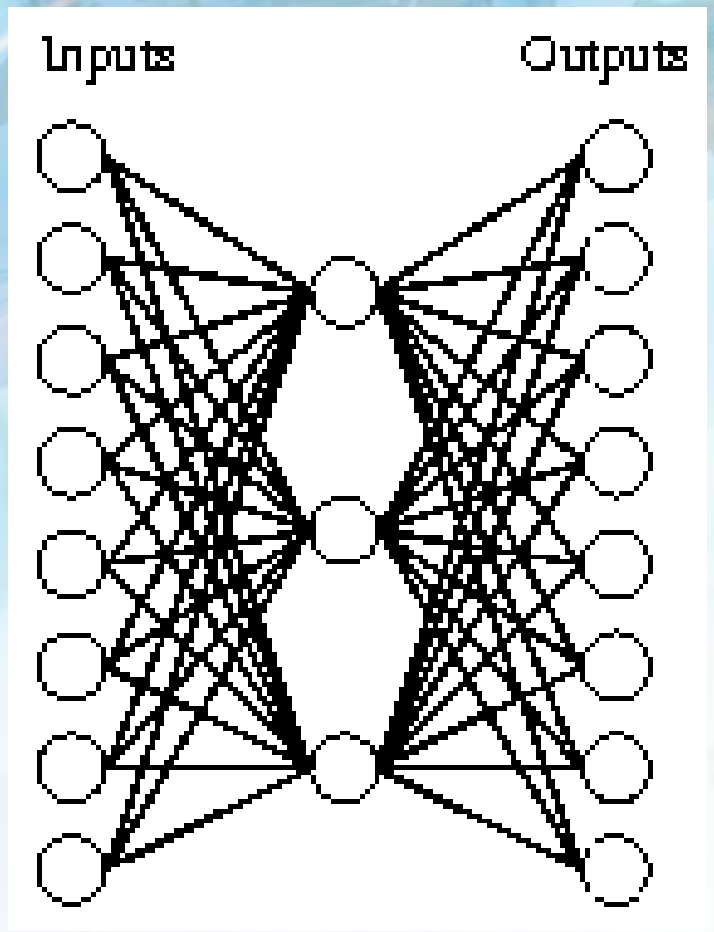
$$\Delta w_{i,j}(n) = \eta\, \delta_j\, x_{i,j} + \alpha\, \Delta w_{i,j}(n-1)$$

- Minimizes error training examples
  - Will it generalize well to unseen instances (over-fitting)?
- Training can be slow typical 1000-10000 iterations
  (use Levenberg-Marquardt instead of gradient descent)
- Using network after training is fast



8-3-8 Binary Encoder

Representation



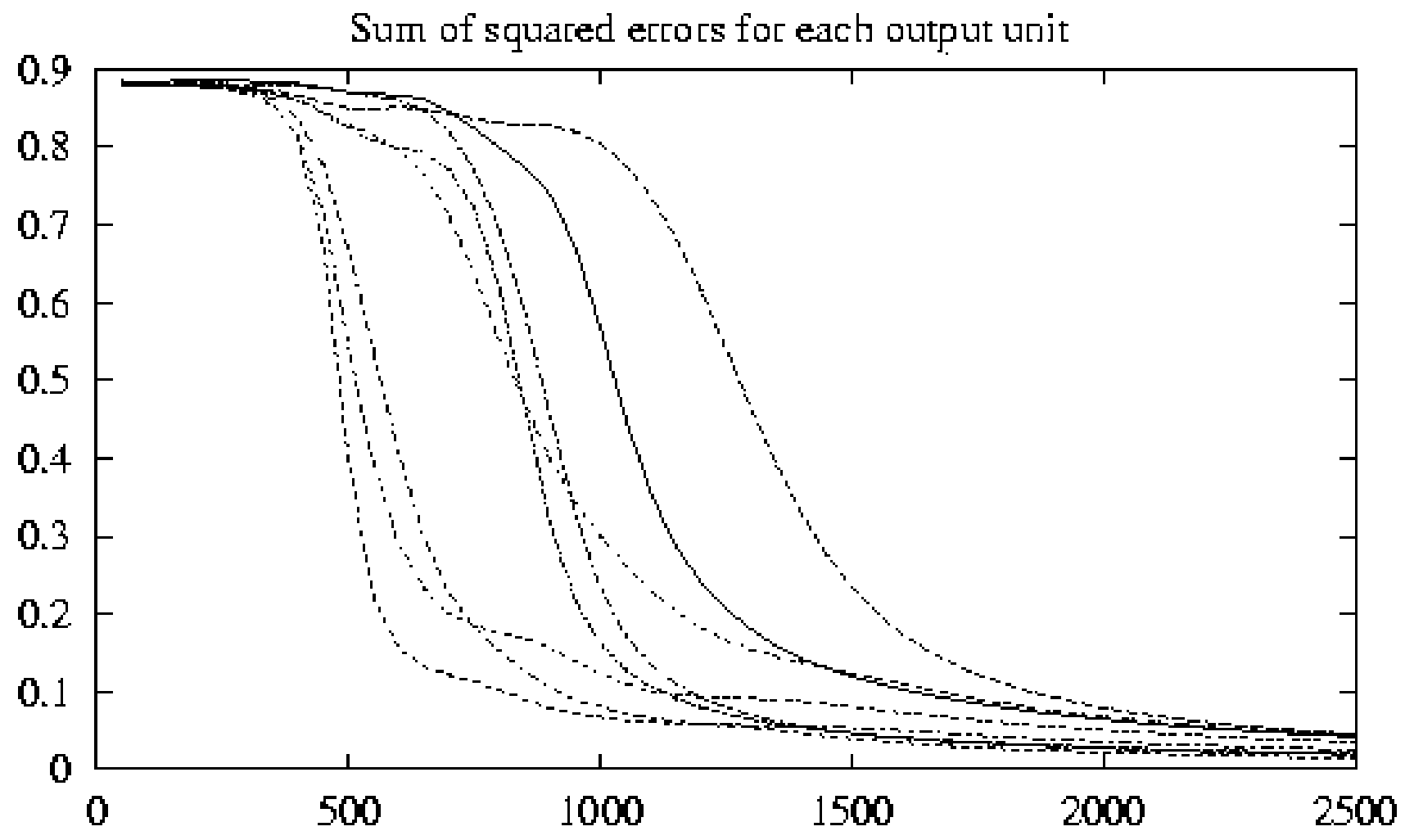
Target Function

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

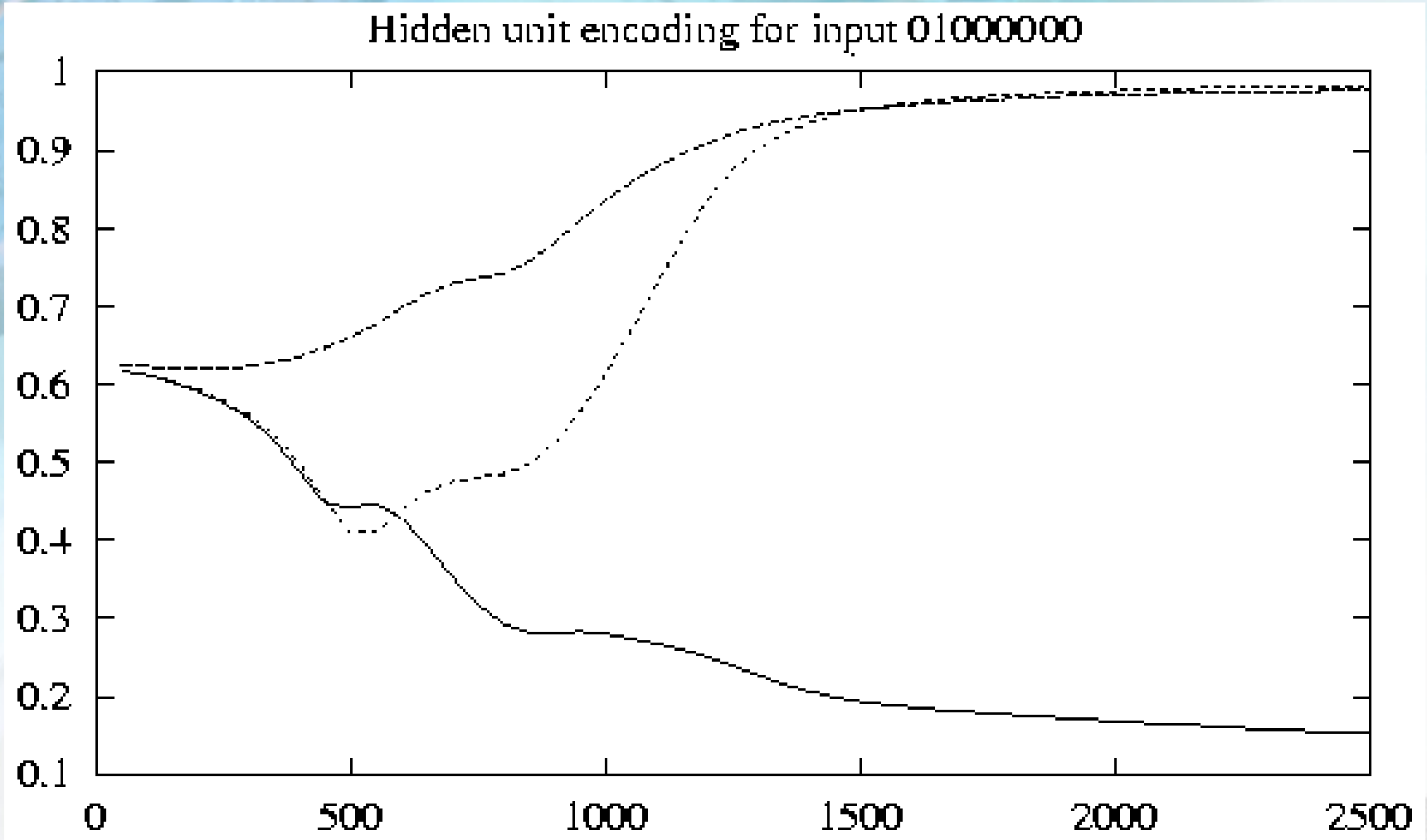
Learned Hidden Layer for 8-3-8 Encoder

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

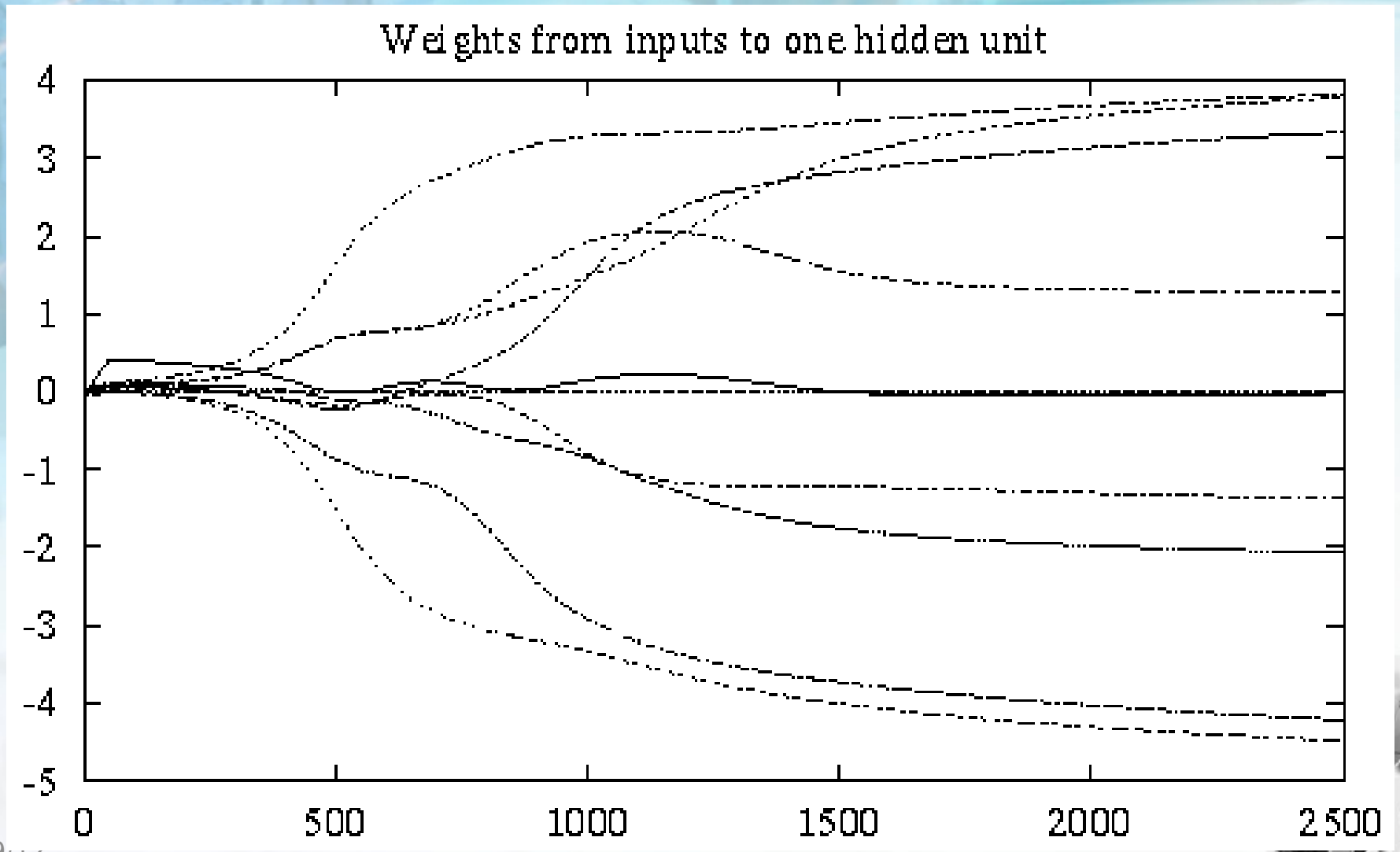
Sum of Squared Errors for Output Units



Hidden Unit Encoding for Input 0100000000



Weights from Inputs to one Hidden Unit



Convergence of Backpropagation

Gradient descent to some local minimum perhaps not global minimum

- Add momentum term: $\Delta w_{ki}(n)$
  - $\Delta w_{ki}(n) = \alpha \, \delta_k(n) \, x_i(n) + \lambda \Delta w_{ki}(n-1)$
    with $\lambda \in [0,1]$
- Stochastic gradient descent
- Train multiple nets with different initial weights

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non -linear functions possible as training progresses

Optimization Methods

- There are other more efficient (faster convergence) optimization methods than gradient descent
  - Newton's method uses a quadratic approximation (2^{nd} order Taylor expansion)
  - $F(\mathbf{x}+\Delta\mathbf{x}) = F(\mathbf{x}) + \nabla F(x)\,\Delta x + \Delta x\,\nabla^2 F(x)\,\Delta x + \ldots$
  - Conjugate gradients
  - Levenberg-Marquardt algorithm

Expressive Capabilities of ANNs

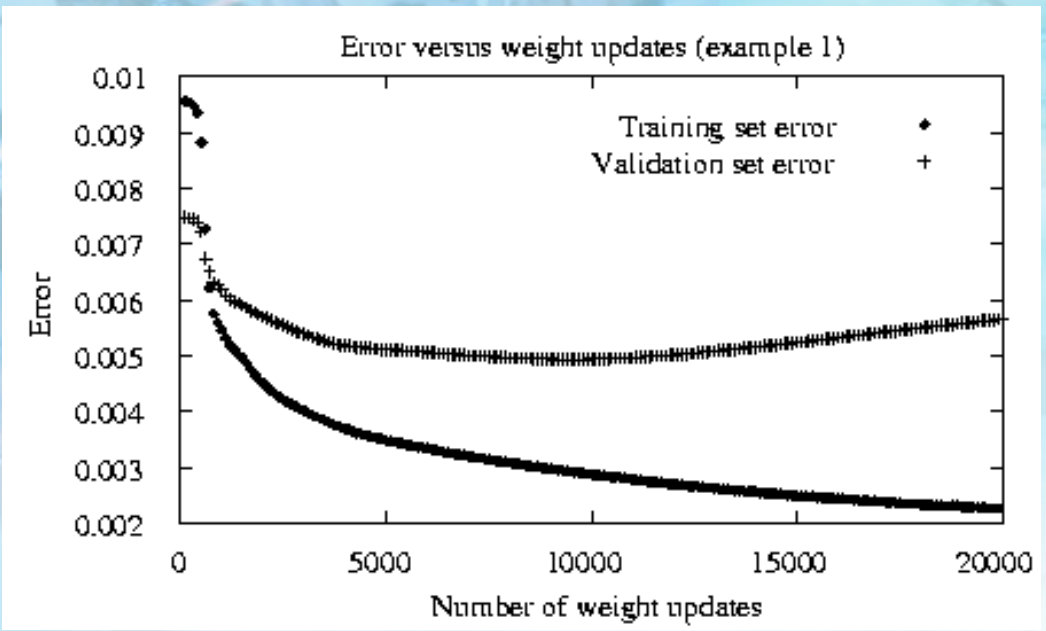
Boolean functions

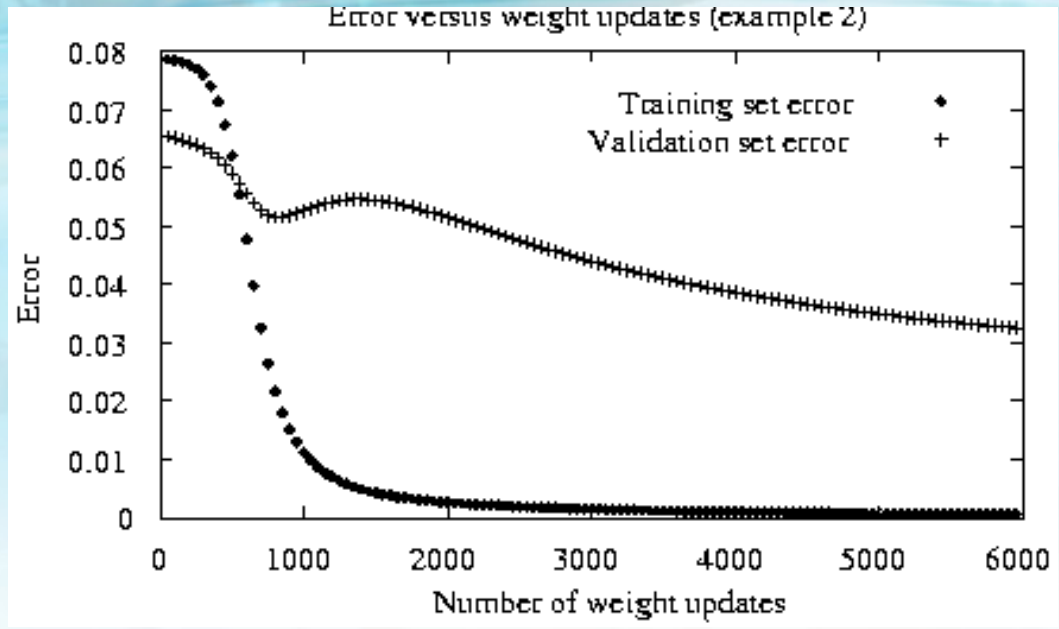
- Every boolean function can be represented by network with single hidden layer
- But might require exponential (in number of inputs) hidden units

Continuous functions

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989, Hornik 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988]

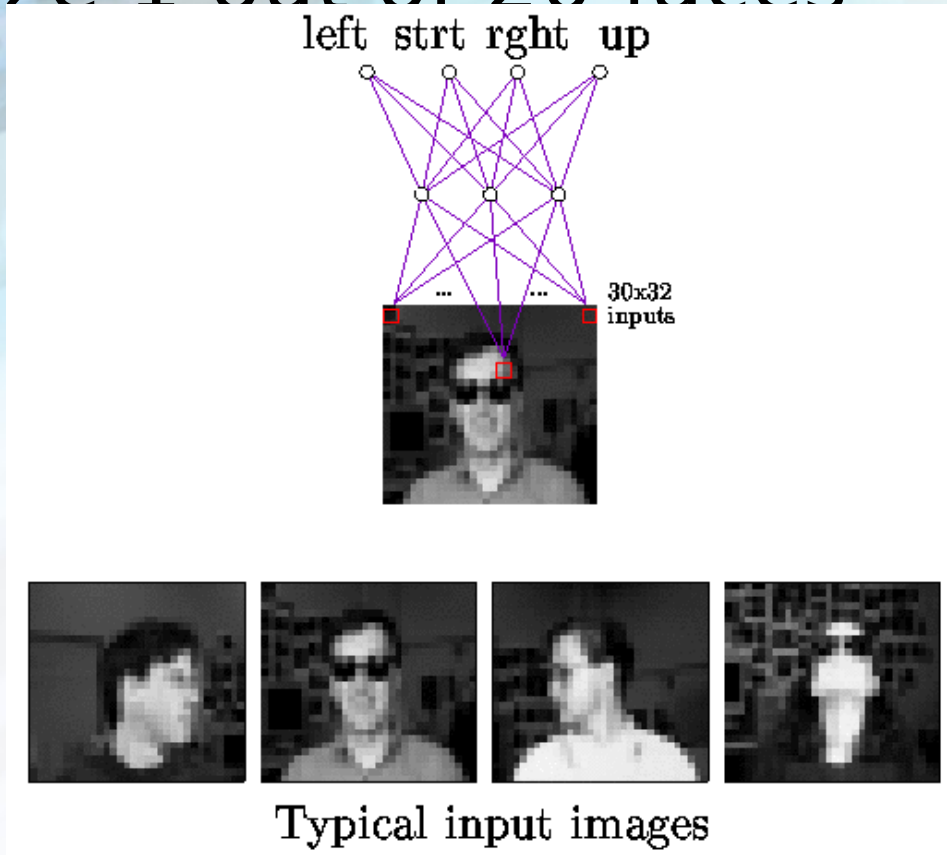
Overfitting in ANN





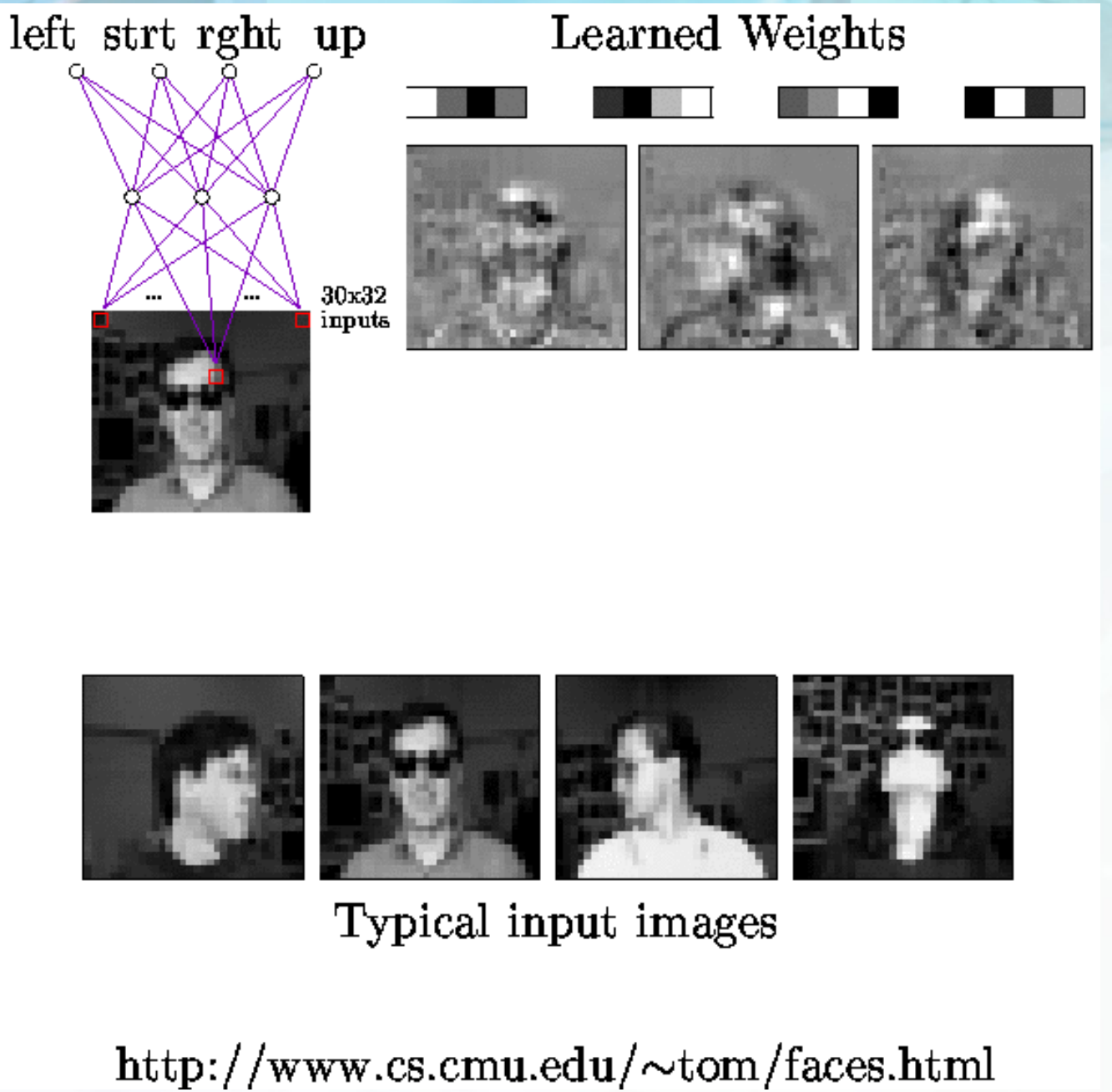
Example: Face Recognition

- 90% accurate learning head pose
- Recognize 1 out of 20 faces



Typical input images

Example: Face Recognition Learned Hidden Units Weights



Alternative Error Functions

Penalize large weights:

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

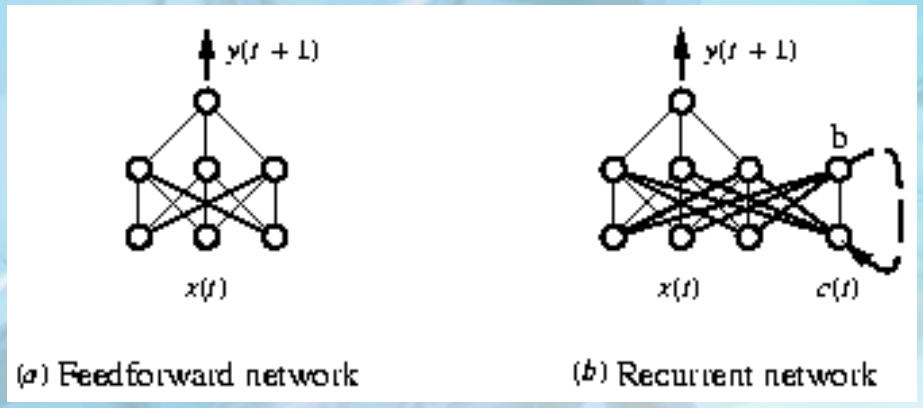
Train on target slopes as well as values:

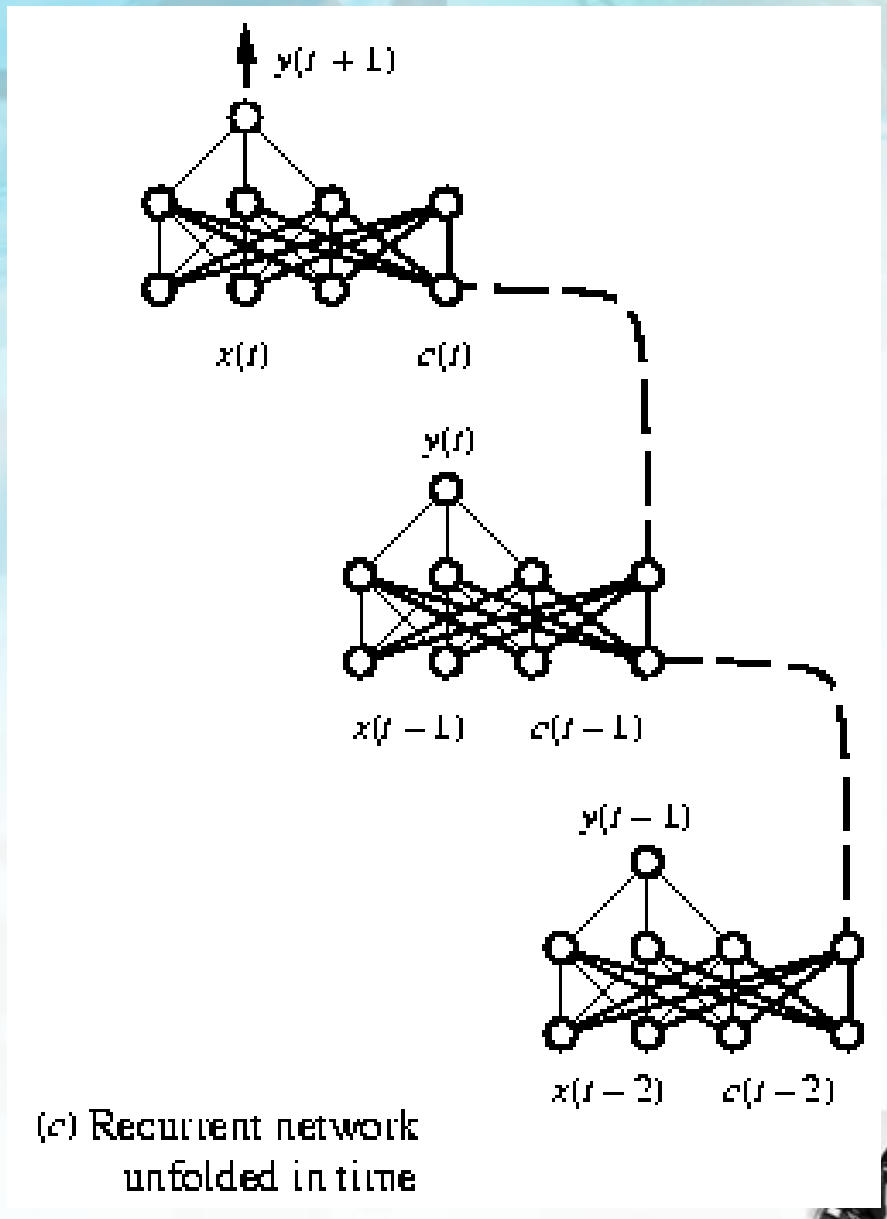
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

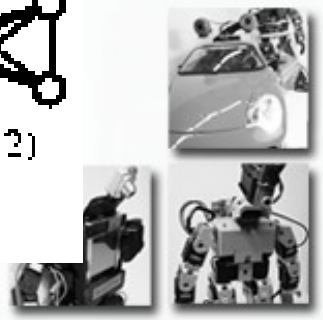
Tie together weights:

- e.g., in phoneme recognition network

Learning of Functions with States Recurrent Networks







Summary

- Biological inspired ANNs. Threshold units
- Multi-layer networks
- Backpropagation Algorithms
- Hidden layer
- Example: Face Recognition
- Extensions to ANNs
  - Neuron models
  - Error functions

Background

- Frank Hoffman. http://www.nada.kth.se/kurser/kth/2D1431/02/index.html
- Neural Networks A Comprehensive Foundation, Simon Haykin, Prentice-Hall, 1999
- Networks for Pattern Recognition , C.M. Bishop, Oxford University Press, 1996
- Neural Network Design , M. Hagan et al, PWS, 1995.
- Perceptrons: An Introduction to Computational Geometry, Minsky, Papert, 1969.