Artificial Neural Networks

鮑興國 Ph.D.

National Taiwan University of
Science and Technology

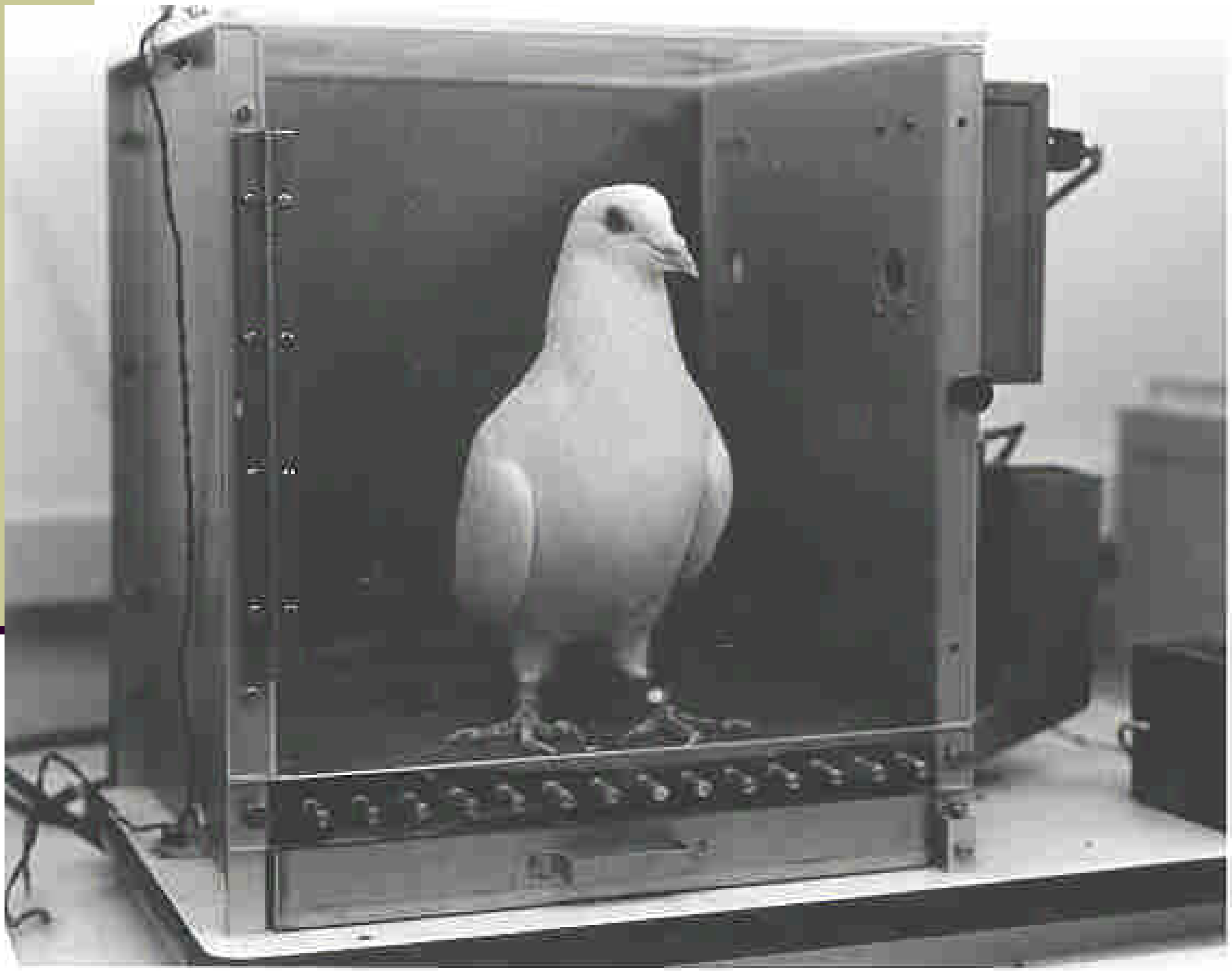
Outline

- Perceptrons
- Gradient descent
- Multilayer Feedforward networks
- Backpropagation
- Hidden layer representations
- Examples
- Advanced topics

What is an Artificial Neural Network?

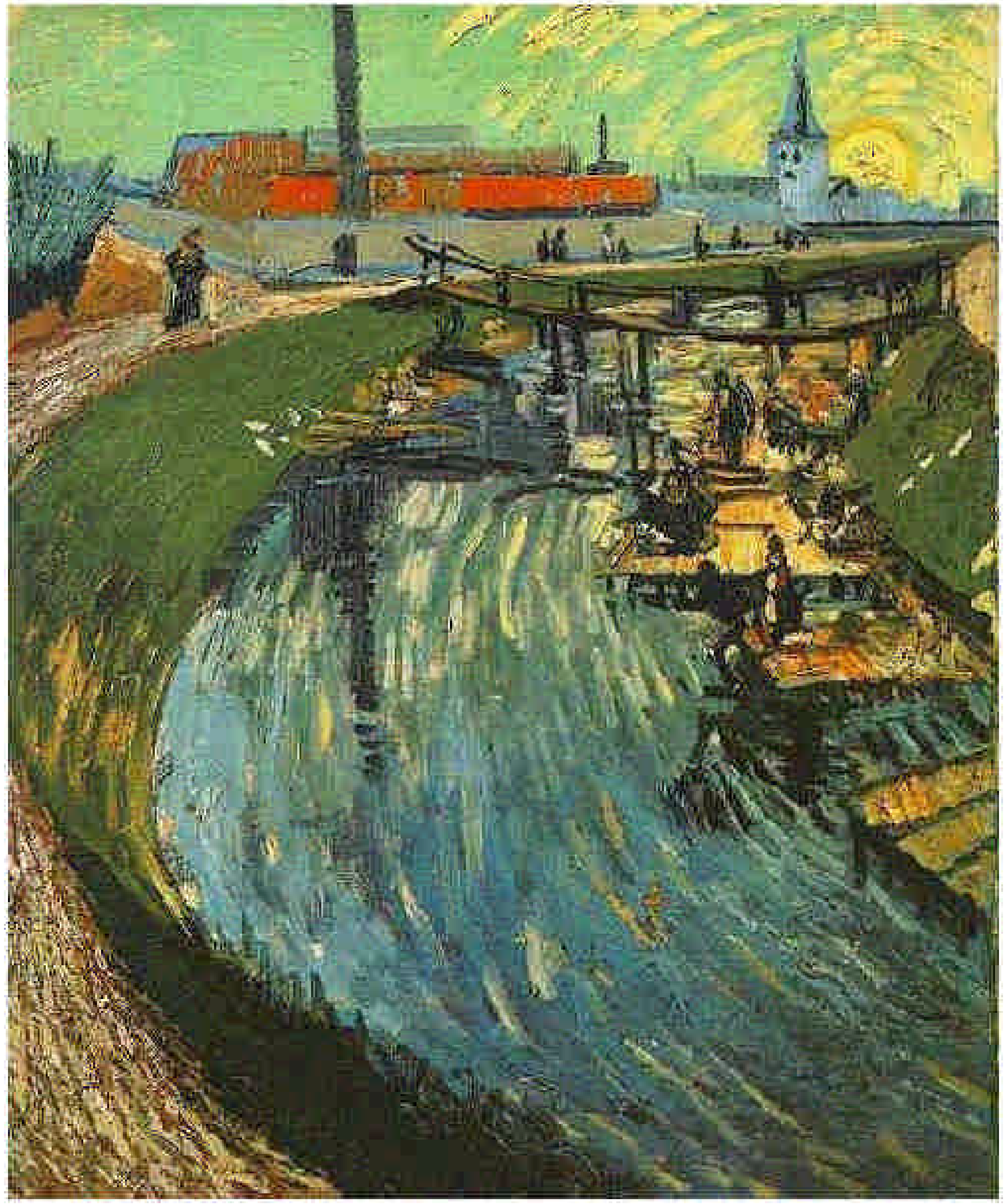
- It is a formalism for representing functions inspired from biological learning systems
- The network is composed of parallel computing units which each computes a simple function
- Some useful computations taking place in Feedforward Multilayer Neural Networks are
  - Summation
  - Multiplication
  - Threshold (e.g., $1/(1 + e^{-x})$, the sigmoidal threshold function). Other functions are also possible

Pigeons as art experts



- Pigeons as art experts (Watanabe *et al.* 1995)
- Experiment:

1. Pigeon in Skinner box
2. Present paintings of two different artists (e.g. Chagall / Van Gogh)
3. Reward for pecking when presented a particular artist (e.g. Van Gogh)



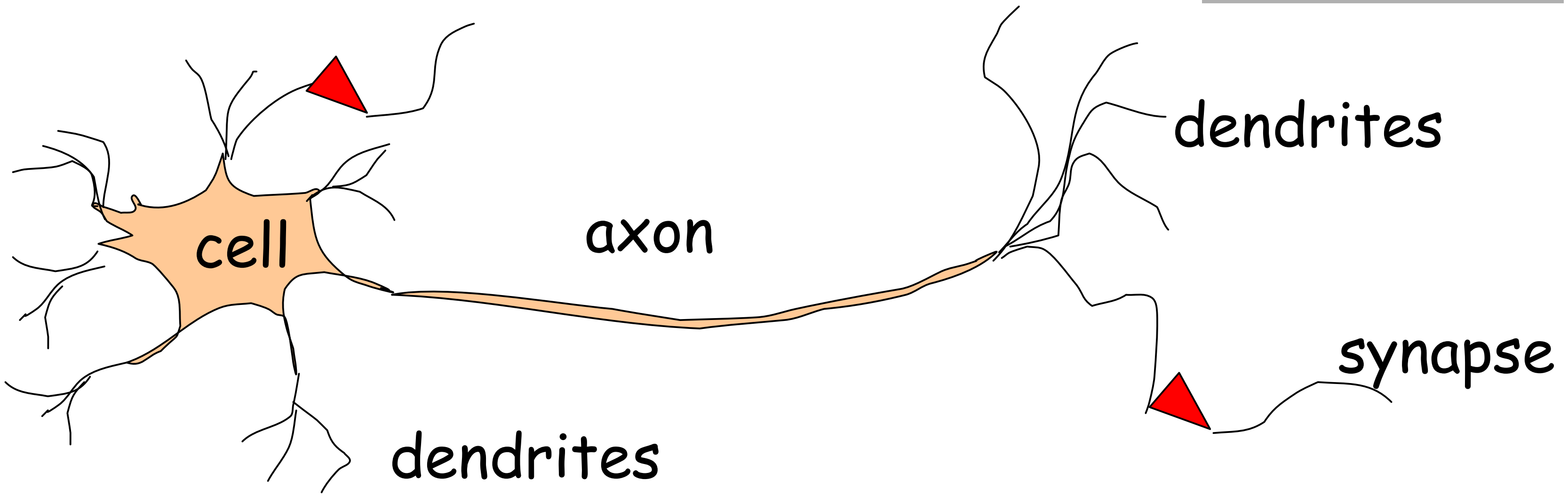
Pigeons as art experts (cont.)

- Pigeons were able to discriminate between Van Gogh and Chagall with 95% accuracy (when presented with pictures they had been trained on)

- Discrimination still 85% successful for previously unseen paintings of the artists

- Pigeons do not simply memorise the pictures
- They generalize from the already seen to make predictions

Biological Motivation



- Biological Learning Systems are built of very complex webs of interconnected neurons
- Information-processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons
- ANNs attempt to capture this mode of computation

Biological Neural Systems

- Neuron switching time : > 10^{-3} secs
  - Computer takes 10^{-10} secs
- Number of neurons in the human brain: ~10^{11}
- Connections (synapses) per neuron: ~10^{4}-10^{5}
- Face recognition : ~0.1 secs
  - 100 inference steps? Brain must be parallel!
- High degree of parallel computation
- Distributed representations

Properties of Artificial Neural Nets (ANNs)

- Many simple neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed processing
- Learning by tuning the connection weights
- ANNs are motivated by biological neural systems; but not as complex as biological systems
  - For instance, individual units in ANN output a single constant value instead of a complex time series of spikes

A Brief History of Neural Networks (Pomerleau)

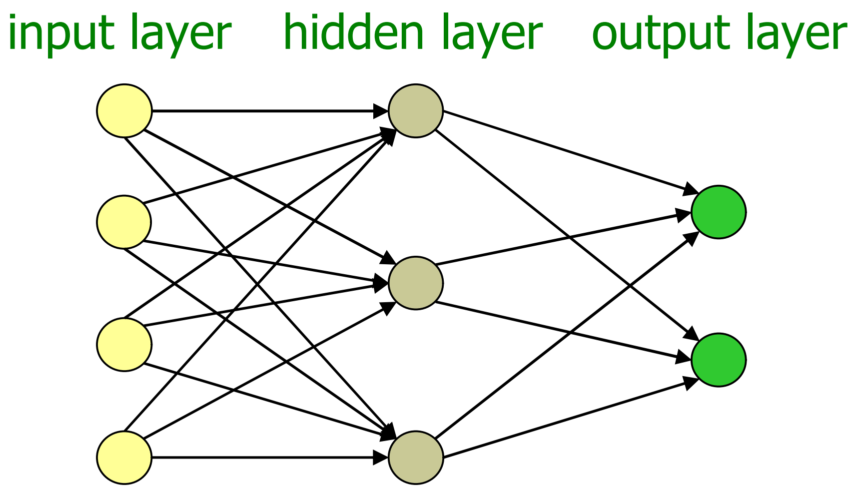
- 1943: McCulloch and Pitts proposed a model of a neuron → Perceptron (Mitchell, section 4.4)
- 1960s: Widrow and Hoff explored Perceptron networks (which they called “Adelines”) and the delta rule.
- 1962: Rosenblatt proved the convergence of the perceptron training rule.
- 1969: Minsky and Papert showed that the Perceptron cannot deal with nonlinearly-separable data sets --- even those that represent simple function such as X-OR.
- 1975: Werbos’ ph.D. thesis at Harvard (beyond regression) defines backpropagation.
- 1985: PDP book published that ushers in modern era of neural networks.
- 1990’s: Neural networks enter mainstream applications.

Appropriate Problem Domains for Neural Network Learning

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Form of target function is unknown
- Humans do not need to interpret the results (black box model)
- Training examples may contain errors (ANN are robust to errors)
- Long training times acceptable

Prototypical ANN Approach

- Mostly, network structure is fixed. Therefore, learning = weight adjustment
  - deciding the structure is an art!
- Units interconnected in layers
  - directed, acyclic graph (DAG)
  - skip-layer connection is possible
  - the network can be sparse, with not all possible connections within a layer being present (convolutional networks)



Types of ANNs

- Feedforward: Links are unidirectional, and there are no cycles, i.e., the network is a directed acyclic graph (DAG). Units are arranged in layers, and each unit is linked only to units in the next layer. There is no internal state other than the weights
- Recurrent: Links can form arbitrary topologies. Cycles can implement memory. Behavior can become *unstable*, *oscillatory*, or *chaotic*

ALVINN: training and performance

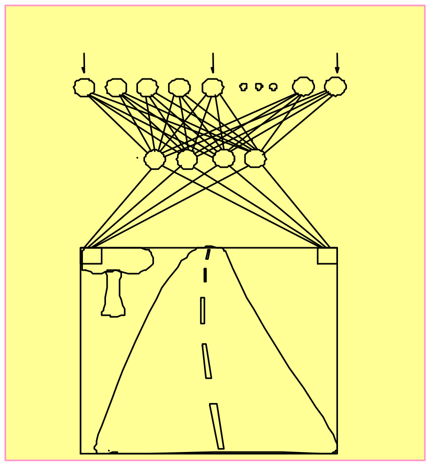
Drives 70 mph on a public highway, by ~ 5 mins training

Camera image

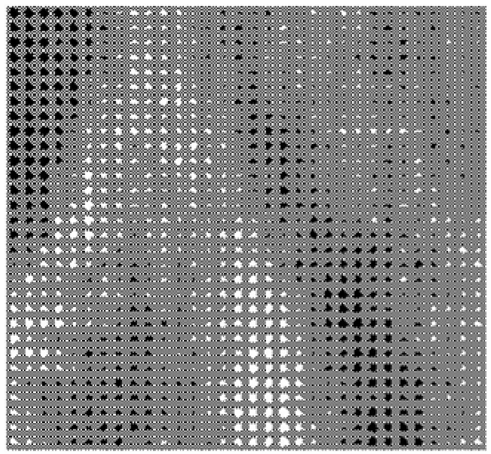
30 outputs for steering

4 hidden units

30x32 pixels as inputs



The weights from a hidden unit to 30 output units



30x32 weights into one out of four hidden unit. A white box indicates a positive weight and a black box a negative weight

ALVINN: training and performance

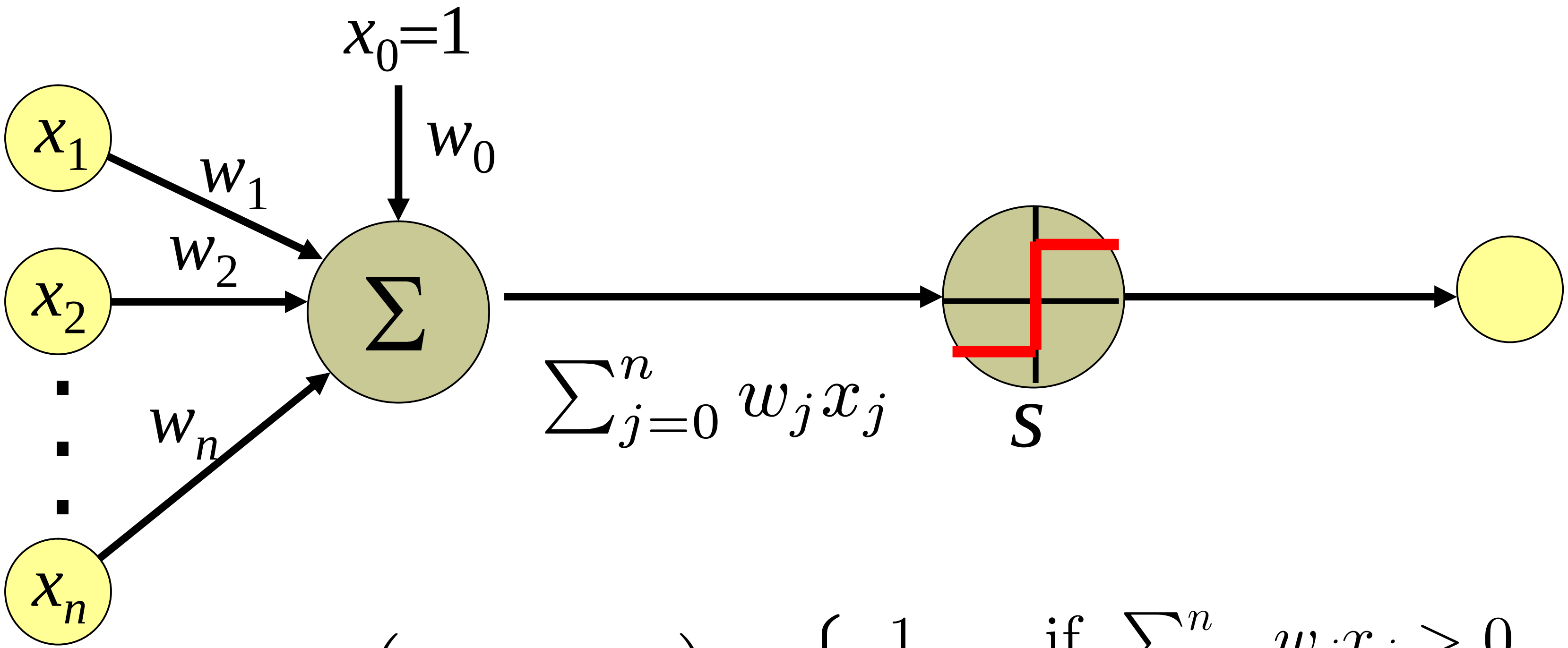
- Trained with computer-generated road images
- Involved 1200 different combinations of scenes, curvatures, lighting conditions and distortion levels
- Entire driver implemented on an on-board computer and a modified Chevy van!
- Performed comparably to the best traditional vision-based navigation systems evaluated under similar conditions
- Training was done in half-an-hour!
    - Was training done on board?
- For comparison -- Algorithm-based drivers take months for algorithm development

Perceptrons

- Structure & function
  - inputs, weights, threshold
  - hypotheses in weight vector space
- Representational power
  - defines a hyperplane decision surface
  - linearly separable problems
  - most boolean functions
  - m of n functions
    - Output “1” if m of n inputs are “1”s

Perceptron

- Linear threshold unit (LTU)

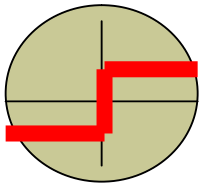


$$s\left(\sum_{j=0}^{n} w_j x_j\right) = \begin{cases} 1 & \text{if } \sum_{j=0}^{n} w_j x_j > 0 \\ -1 & \text{otherwise} \end{cases}$$

Purpose of the Activation Function *s*

- We want the unit to be “active” (near +1) when the “right” inputs are given
- We want the unit to be “inactive” (near -1) when the “wrong” inputs are given.
- It’s preferable for *s* to be nonlinear. Otherwise, the entire neural network collapses into a simple linear function.

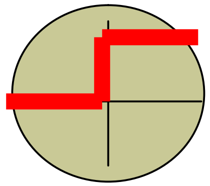
Possibilities for function *s*



Sign function

sign(x) = +1, if $x > 0$

-1, if $x \leq 0$

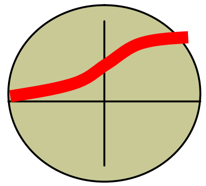


Step function

step(x) = 1, if x > threshold

0, if $x \leq$ threshold

(in picture above, threshold = 0)

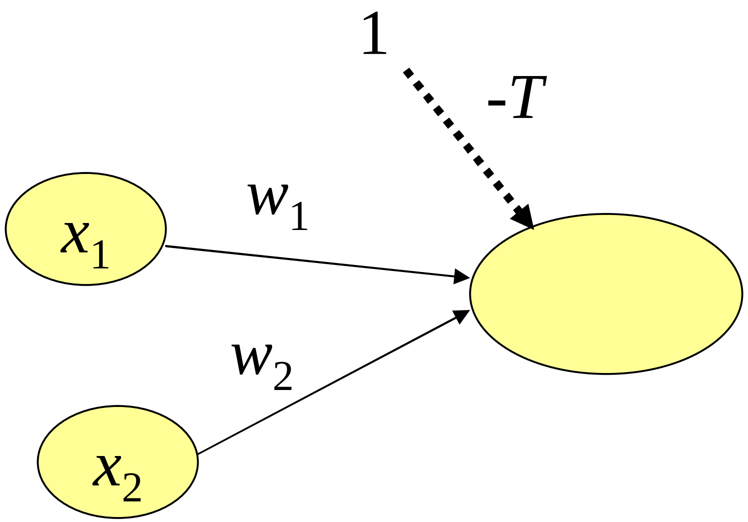


Sigmoid (logistic) function

sigmoid(x) = $1/(1+e^{-x})$

Adding an extra input with activation $x_0 = 1$ and weight $w_0 = -T$ (called the ***bias weight***) is equivalent to having a threshold at T. This way we can always assume a 0 threshold.

Using a Bias Weight to Standardize the Threshold

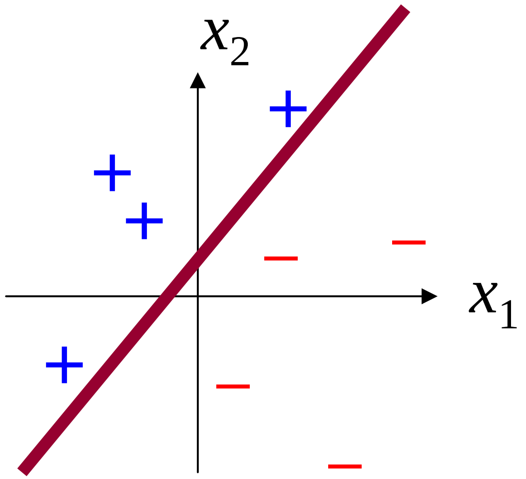


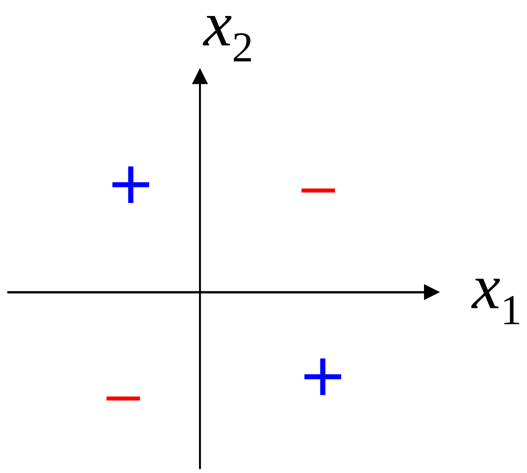
$$w_1x_1 + w_2x_2 < \boldsymbol{T}$$

$$\Downarrow$$

$$w_1x_1 + w_2x_2 - T < \mathbf{0}$$

Decision Surface of a Perceptron

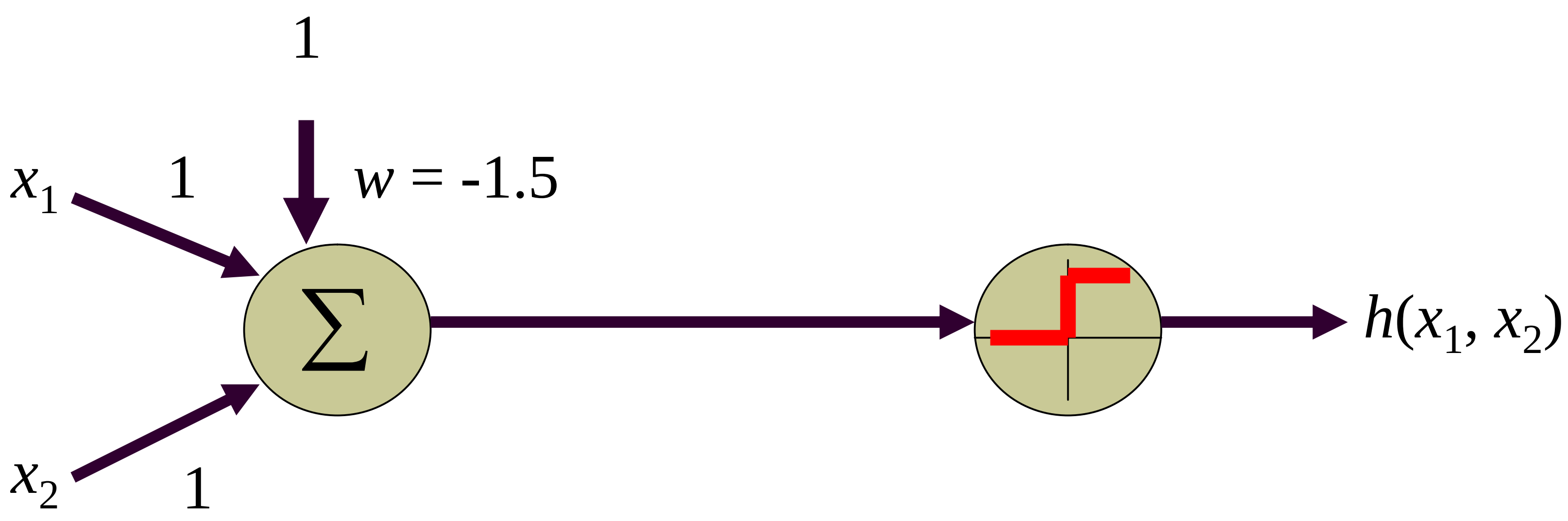




- Perceptron is able to represent some useful functions and (x_1, x_2): choose weights $w_0 = -1.5$, $w_1=1$, $w_2=1$
- But functions that are not linearly separable (e.g. XOR) are not representable

Implementing AND

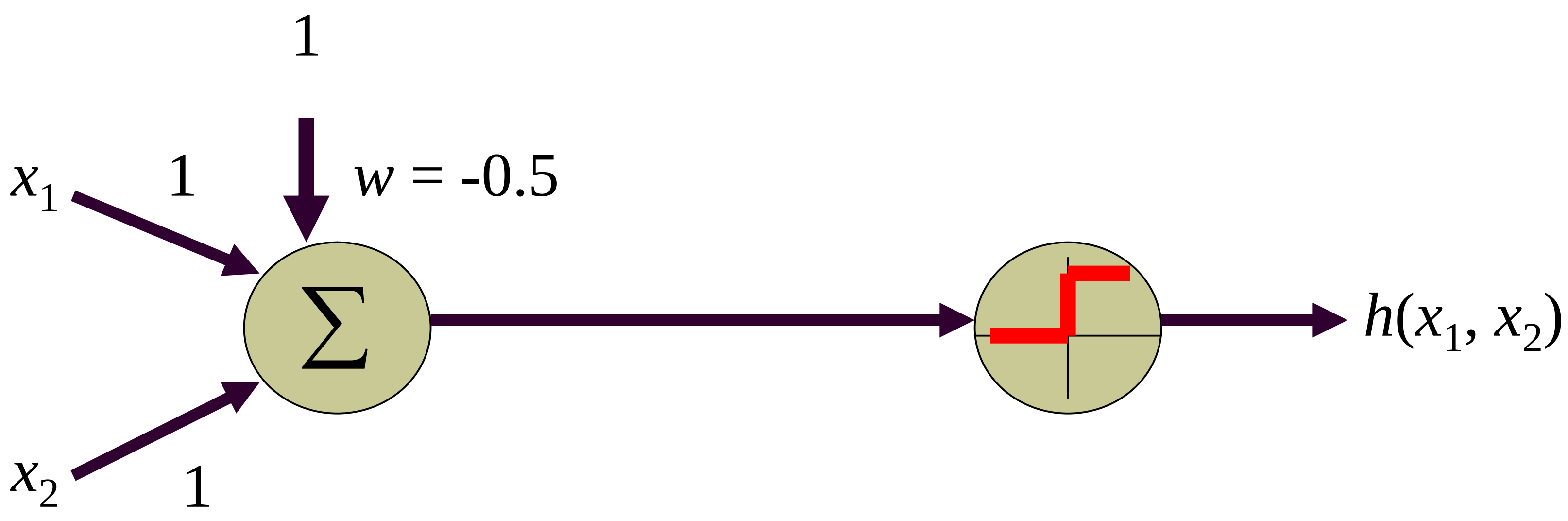
Assume Boolean (0/1) input values…



$$h(x_1, x_2) = 1 \quad \text{if } -1.5 + x_1 + x_2 > 0$$
$$= 0 \quad \text{otherwise}$$

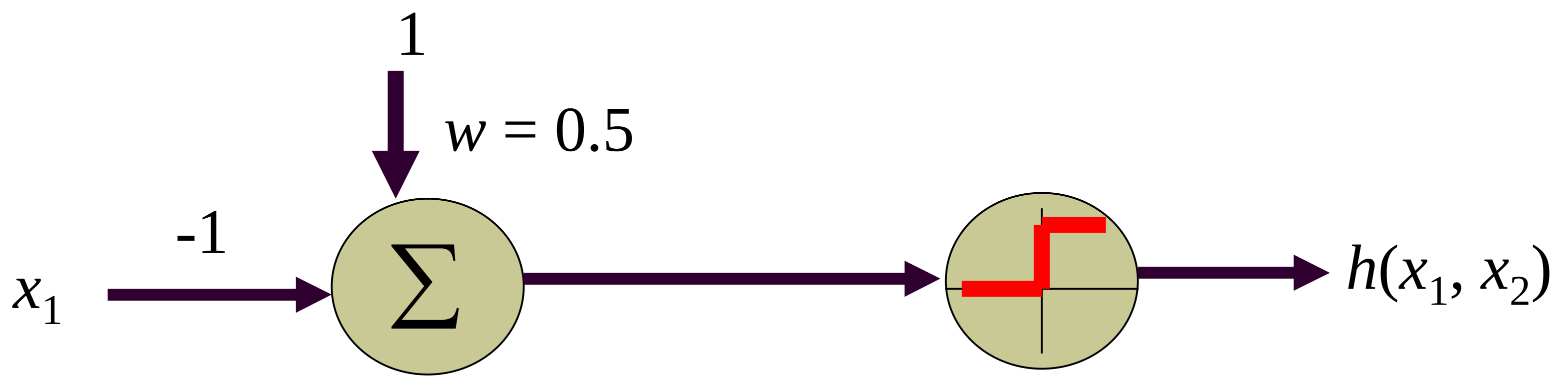
Implementing OR

Assume Boolean (0/1) input values…



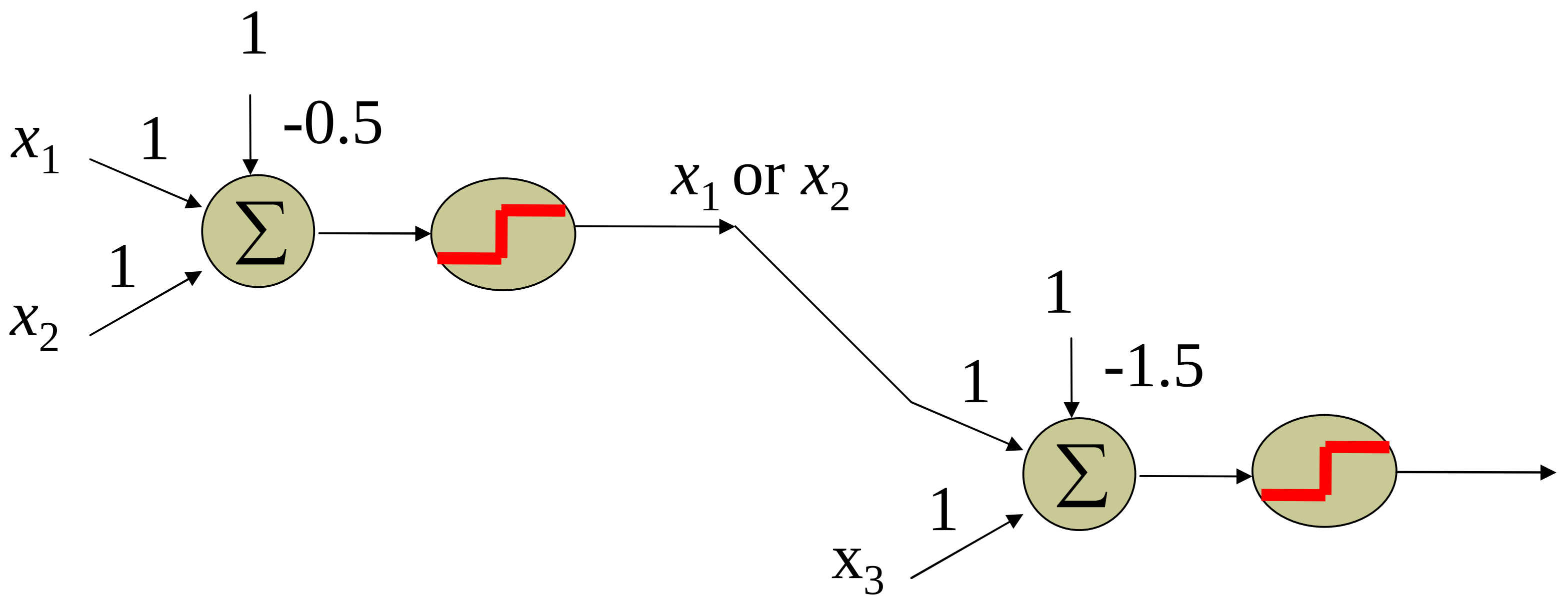
$$h(x_1,x_2) = 1 \quad \text{if } -0.5 + x_1 + x_2 > 0$$
$$= 0 \quad \text{otherwise}$$

Implementing NOT



$$h(x_1) = 1 \quad \text{if } 0.5 - x_1 > 0$$
$$= 0 \quad \text{otherwise}$$

Implementing more complex Boolean functions



(x_1 or x_2) and x_3

Perceptron Learning Rule

$w_j \leftarrow w_j + \Delta w_j$

$\Delta w_j = \eta\,(y - h(\mathbf{x}))\,x_j$

y is the target output for the current training example

$h(\mathbf{x})$ is the perceptron output

η is a small constant (e.g. 0.1) called *learning rate*

- Start with some random weights (usually small values)
- If the output is correct ($y = h(\mathbf{x})$) the weights w_j are not changed
- If the output is incorrect ($y \neq h(\mathbf{x})$) the weights w_j are changed such that the output of the perceptron for the new weights is *closer* to y.
- The algorithm converges to the correct classification
  - if the training data is linearly separable
  - and η is sufficiently small

Perceptron Learning Rule

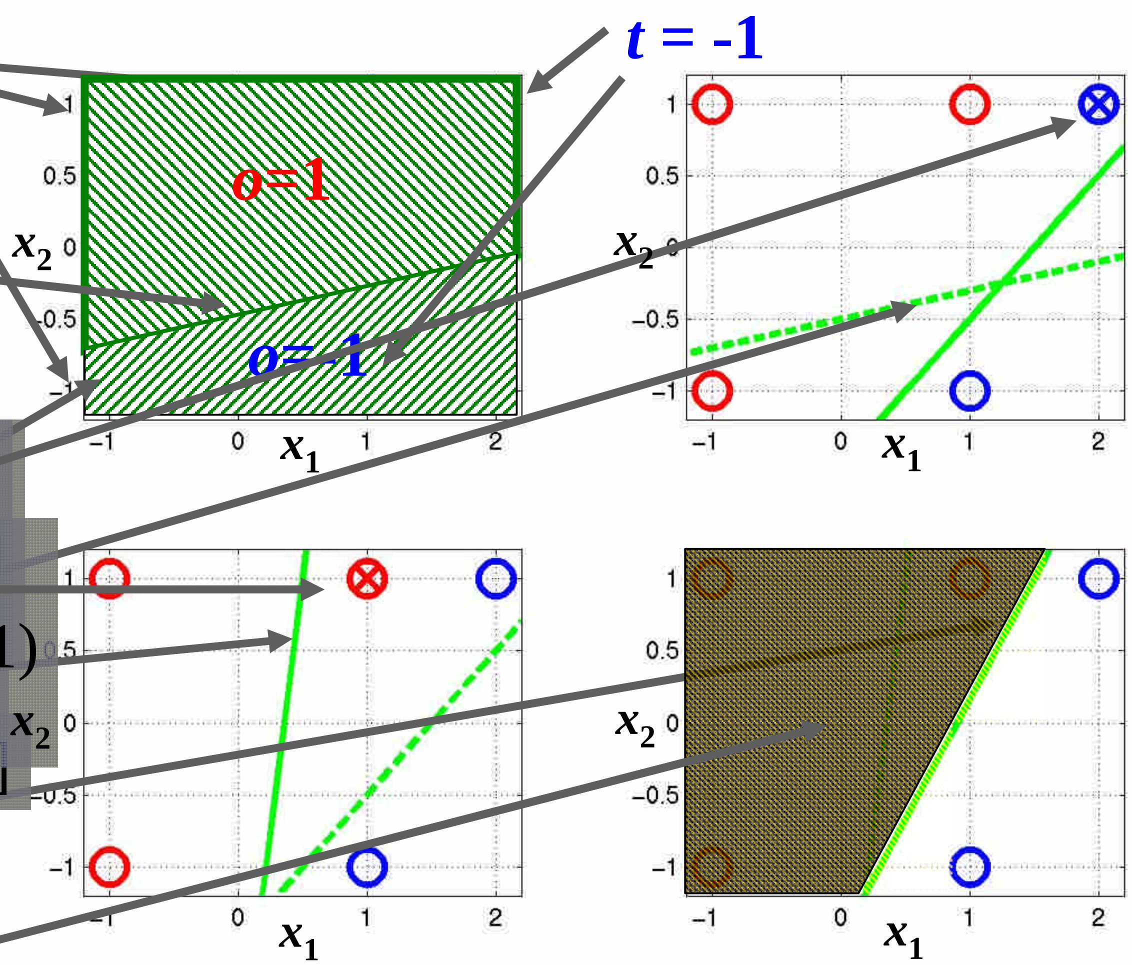
$w = [0.25\ –0.1\ 0.5]$
$x_2 = 0.2\ x_1 – 0.5$

$(x, y)=([2,1], –1)$
$h = \text{sgn}(0.45-0.6+0.3)$
$=1$
$\Delta w = [0.2\ –0.2\ –0.2]$

$(x, y)=([1,1], 1)$
$h = \text{sgn}(0.25–0.7+0.1)$
$= –1$
$\Delta w = [–0.2\ –0.4\ –0.2]$

$\Delta w = [0.2\ 0.2\ 0.2]$

$-0.5x_1+0.3x_2+0.45>0 \Rightarrow h = 1$



Perceptron Convergence Theorem

- If the training data are linear separable, the perceptron learning algorithm is guaranteed to find an exact solution in a finite number of steps (by many – Rosenblatt (1962), Block, Nilsson, Minsky and Pappert, Duda and Hart, etc.)
- Proof:
- Many different solutions can be found with the different initialization of the parameters or the order of presentation of data points

Gradient Descent Learning Rule

- Perceptron learning rule fails to converge if examples are not linearly separable
- Consider linear unit without threshold and continuous output h (not just $-1, 1$)
  - $h = w_0 + w_1 x_1 + \ldots + w_n x_n$
- Train the w_j's such that they minimize the squared error
  - $E[w_1, \ldots, w_n] = ½ \sum_{i \in D} (y_i - h(x_i))^2$

  where D is the set of training examples

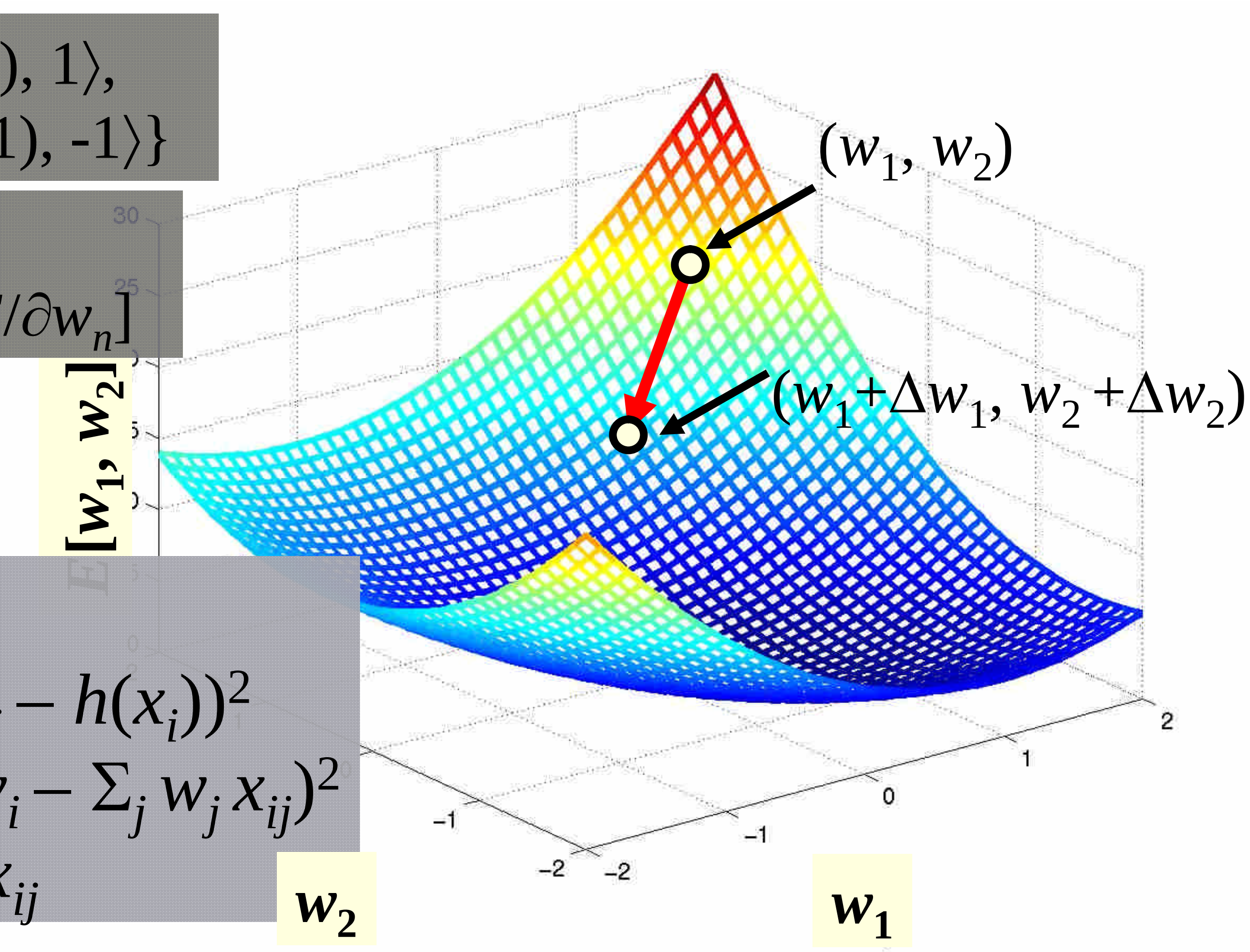
Gradient Descent

$D = \{\langle(1,1), 1\rangle, \langle(-1,-1), 1\rangle,$
$\quad \langle(1,-1), -1\rangle, \langle(-1,1), -1\rangle\}$

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

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

$$\begin{aligned}\Delta w_j &= -\eta\ \partial E/\partial w_j \\ &= -\eta\ \partial/\partial w_j\ \tfrac{1}{2}\Sigma_i(y_i - h(x_i))^2 \\ &= -\eta\ \partial/\partial w_j\ \tfrac{1}{2}\ \Sigma_i(y_i - \Sigma_j\ w_j\ x_{ij})^2 \\ &= \ \eta\ \Sigma_i(y_i - h(x_i))\ x_{ij}\end{aligned}$$



Gradient Descent

- Train the w_j's such that they minimize the squared error
  - $E[w_1,\ldots, w_n] = ½\ \Sigma_{i\in D}\ (y_i - h(x_i))^2$

Gradient:

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

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

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

$= -\eta\ \partial/\partial w_j\ ½\Sigma_i(y_i - h(x_i))^2$

$= -\eta\ \partial/\partial w_j\ ½\Sigma_i(y_i - \Sigma_j\ w_j\ x_{ij})^2$

$= -\eta\ \Sigma_i(y_i - h(x_i))(-x_{ij})$

Gradient Descent

Gradient-Descent(*training_examples*, η)

Each training example is a pair of the form $\langle(x_1, \ldots, x_n), y\rangle$ where $(x_1,\ldots, x_n)$ is the vector of input values, and y is the target output value, η is the learning rate (e.g. 0.1)

- Initialize each w_j to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_j to zero
  - For each $\langle(x_1, \ldots, x_n), y\rangle$ in *training_examples* Do
    - Input the instance $(x_1, \ldots, x_n)$ to the linear unit and compute the output h
    - For each linear unit weight w_j Do
      - $\Delta w_j = \Delta w_j + \eta\,(y - h(\mathbf{x}))\,x_j$
  - For each linear unit weight w_j Do
    - $w_j = w_j + \Delta w_j$
- Termination condition – error falls under a given threshold

Perceptron Learning (Thresholded Version)

1. Initialize weights and threshold: Set weights w_j to small random values
2. Present Input and Desired Output: Set the inputs to the example values x_j and let the desired output be y
3. Calculate Actual Output

$$h = sgn(\vec{w} \cdot \vec{x})$$

4. Adapt Weights: If actual output is different from desired output, then

$$w_j \Leftarrow w_j + \eta(y - h(\mathbf{x}))x_j$$

   where $0 < \eta < 1$ is the learning rate

5. Repeat from Step 2 until done

Gradient Descent Learning (Unthresholded Version)

1. Initialize weights and threshold: Set weights w_j to small random values
2. Present Input and Desired Output: Set the inputs to the example values x_j and let the desired output be y
3. Calculate Unthresholded Output

$$h = \vec{w} \cdot \vec{x}$$

4. Adapt Weights: If actual output is different from desired output, then

$$w_j \Leftarrow w_j + \eta \sum_{i \in D} (y_i - h(\mathbf{x}_i)) x_{ij}$$

   where $0 < \eta < 1$ is the learning rate
5. Repeat from Step 2 until done

Incremental Stochastic Gradient Descent

- Batch mode : gradient descent

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

  $E_D[w] = ½ \sum_i (y_i - h(\mathbf{x}_i))^2$

- Incremental mode: gradient descent

  $w=w - \eta\, \nabla E_i[w]$ over individual training examples i

  $E_i[w] = ½ (y_i - h(\mathbf{x}_i))^2$

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

Comparison Perceptron and Gradient Descent Rule

Perceptron learning rule guaranteed to succeed (converge in finite steps) if

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

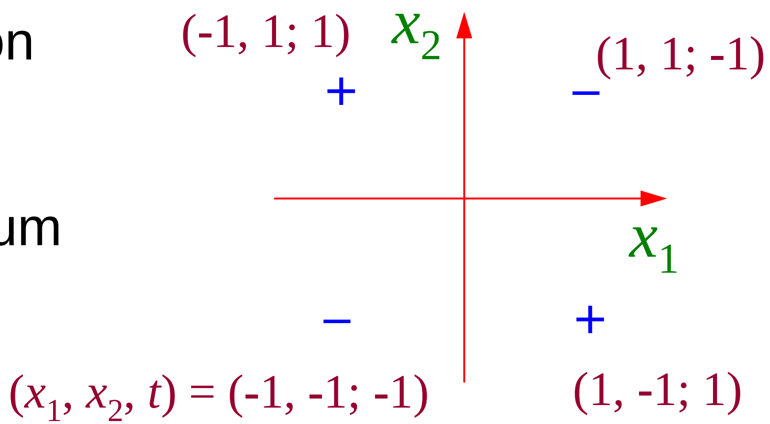
Gradient descent learning rules uses gradient descent

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

XOR

$$
\begin{aligned}
h(\vec{x}) &= \vec{w} \cdot \vec{x} \\
E(\vec{w}) &= \frac{1}{2} \sum\nolimits_{i \in D} (y_i - h(\mathbf{x}_i))^2 \\
&= \frac{1}{2} \Big[(-1 - w_0 - w_1 - w_2)^2 + (1 - w_0 + w_1 - w_2)^2 \\
&\quad + (-1 - w_0 + w_1 + x_2)^2 + (1 - w_0 - w_1 + w_2)^2 \Big] \\
&= 2(1 + w_0^2 + w_1^2 + w_2^2)
\end{aligned}
$$

- The error will reach
  the minimum 2 when $w_0 = w_1 = w_2 = 0$
- For perceptron learning, the iteration will not stop!
- For gradient descent learning, process will converge to the minimum even the dataset is not linearly-separable!



Limitations of Threshold and Perceptron Units

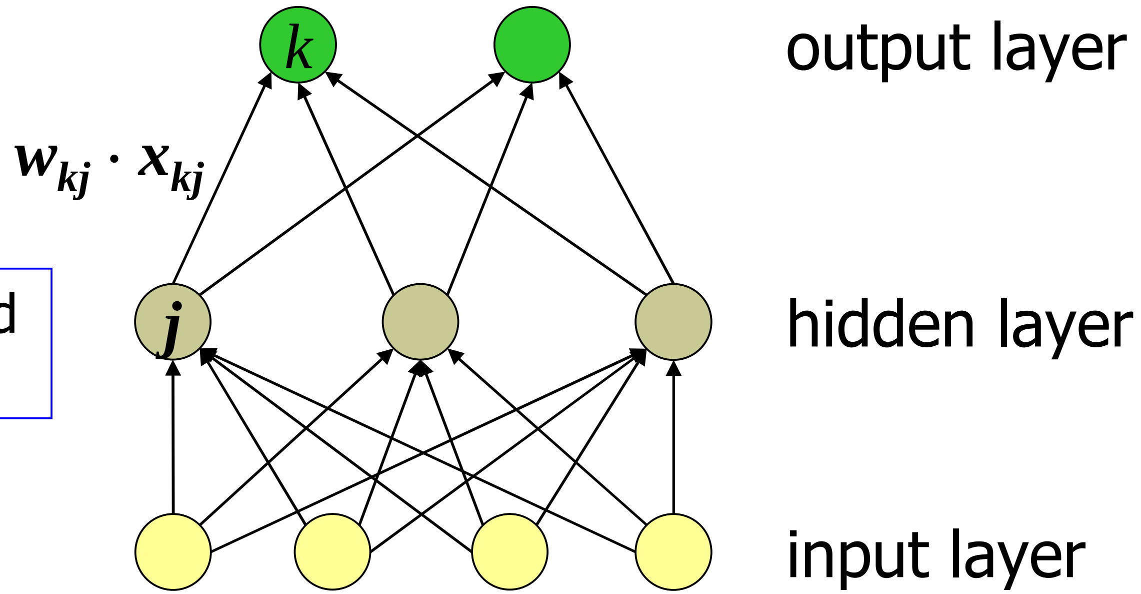
Limitations of Threshold and Perceptron Units

- Perceptrons can only learn linearly separable classes
- Perceptrons cycle if classes are not linearly separable
- Threshold units converge always to MSE hypothesis
- Network of perceptrons – how to train?
- Network of threshold units – not necessary! (why?)

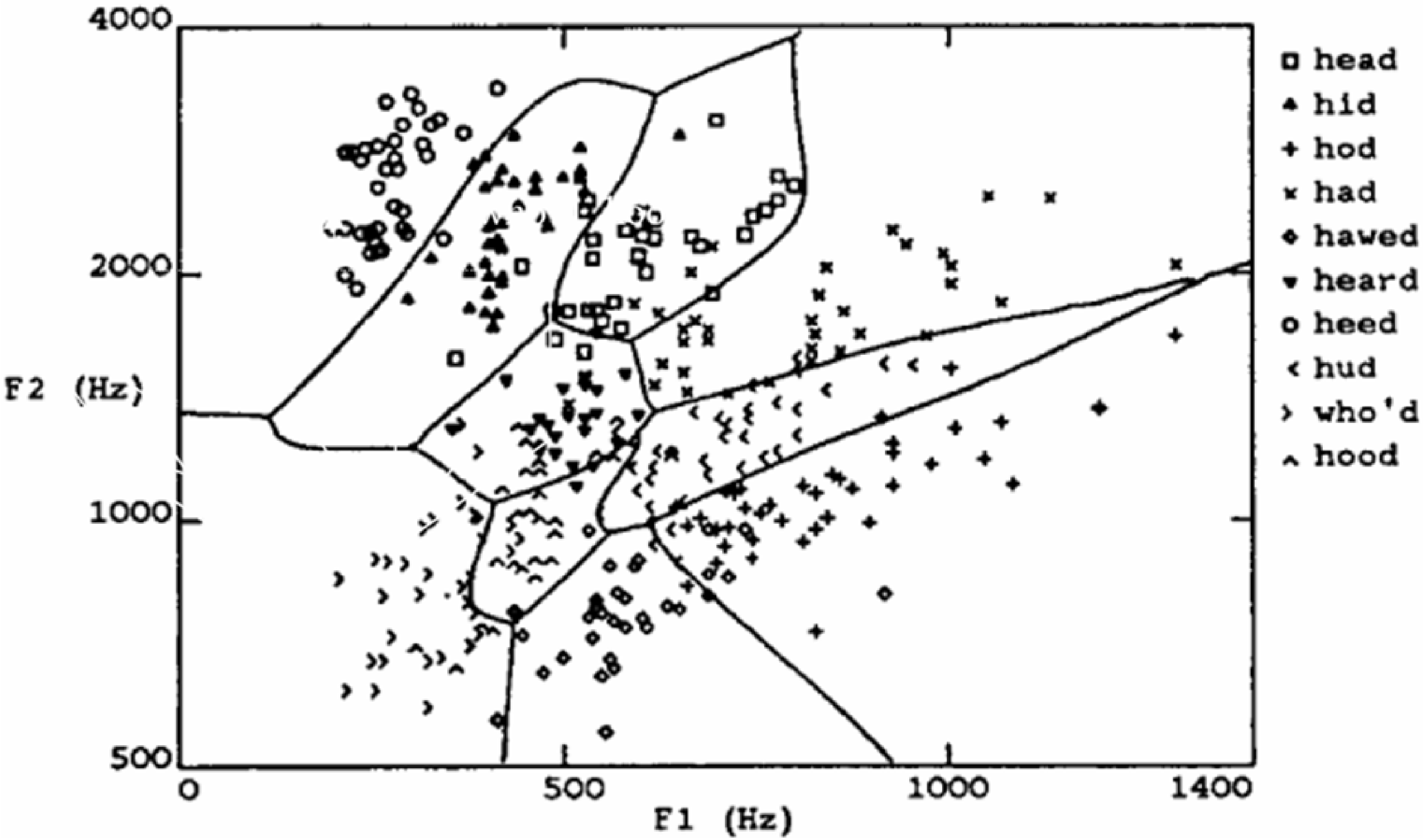
Multilayer Networks

- Single perceptrons can only express linear decision surfaces
- On the other hand, multilayer networks are capable of expressing a rich variety of nonlinear decision surfaces

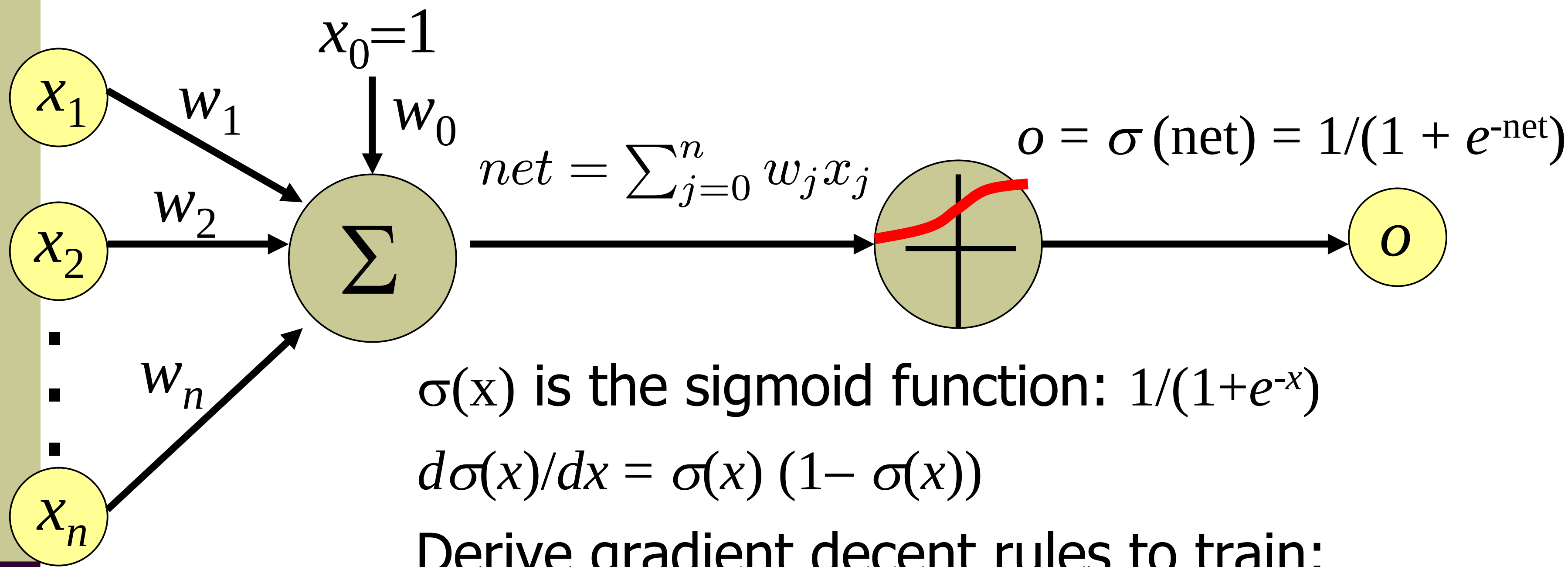
conventionally, it's called a two-layer network



A Speech Recognition Task



Sigmoid Threshold Unit



$\sigma(\mathrm{x})$ 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

  $\partial E/\partial w_j = -\sum_i (y_i - h(\mathbf{x}_i))\, h(\mathbf{x}_i)\,(1 - h(\mathbf{x}_i))\, x_{ij}$

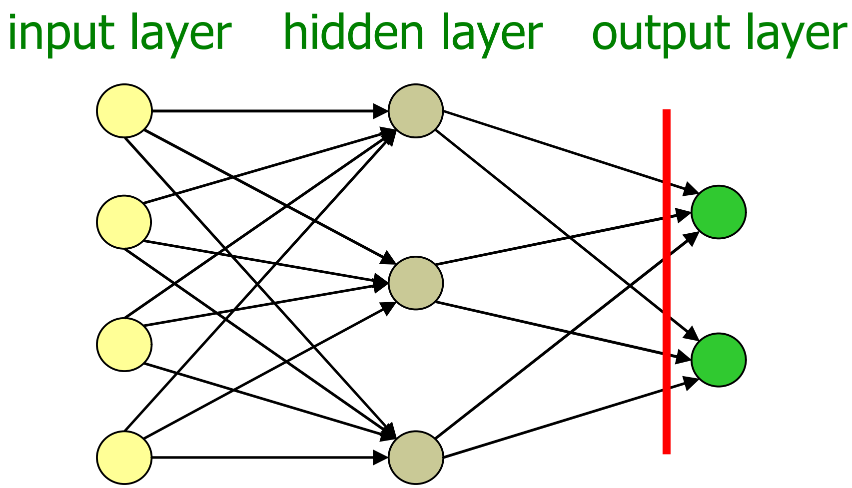
- Multilayer networks of sigmoid units
  backpropagation:

Designation of Output Units

- Regression: identity function
- Binary classification: e.g., sigmoid function
- Multiclass classfication: softmax function

$$h(\mathbf{x}, \mathbf{w}) = \frac{\exp(net_k)}{\sum_\ell \exp(net_\ell)}$$

- *K* binary classification problem: *K* sigmoid function for each problem



BACKPROPAGATION Algorithm

Initialize each w_j to some small random value

Until the termination condition is met, Do

For each training example $\langle(x_1,\ldots, x_n), y\rangle$ Do

Input the instance $(x_1,\ldots, x_n)$ to the network and compute the network outputs h_k for every output unit k

For each output unit k

$$\delta_k = h_k(1-h_k)(y_k-h_k)$$

For each hidden unit j

$$\delta_j = h_j(1-h_j)\ \Sigma_k\, w_{kj}\ \delta_k$$

For each network weight w_{kj} Do

$w_{kj} = w_{kj} + \Delta w_{kj}$ where

$$\Delta w_{kj} = \eta\ \delta_k\ x_{kj}$$

Derivation of the **BACKPROPAGATION** Rule I

$$E_i(\vec{w}) \equiv \frac{1}{2}\sum_{k \in \mathrm{outputs}}(y_k - h_k)^2$$

$$\Delta w_{kj} = -\eta \frac{\partial E_i}{\partial w_{kj}}$$

- x_{kj}: the jth input to unit k
- w_{kj}: the weight associated with the jth input to unit k
- $\mathrm{net}_k = \Sigma_j \, w_{kj} \, x_{kj}$ (the weighted sum of inputs for unit k)
- h_k: the output computed by unit k
- y_k: the target output for unit k
- σ: the sigmoid function
- outputs: the set of units in the final layer of the network
- Downstream(k): the set of units whose immediate inputs include the output of unit k

Derivation of the **BACKPROPAGATION** Rule II

$$\begin{aligned}
\frac{\partial E_i}{\partial w_{kj}} &= \frac{\partial E_i}{\partial \mathrm{net}_k}\frac{\partial \mathrm{net}_k}{\partial w_{kj}} \\
&= \frac{\partial E_i}{\partial \mathrm{net}_k} x_{kj}
\end{aligned}$$

Training rule for output unit weights:

$$\begin{aligned}
\frac{\partial E_i}{\partial \mathrm{net}_k} &= \frac{\partial E_i}{\partial h_k}\frac{\partial h_k}{\partial \mathrm{net}_k} \\
\frac{\partial E_i}{\partial h_k} &= \frac{\partial}{\partial h_k}\frac{1}{2}\sum_{\ell \in \mathrm{outputs}} (y_\ell - h_\ell)^2 \\
\frac{\partial E_i}{\partial h_k} &= \frac{\partial}{\partial h_k}\frac{1}{2}(y_k - h_k)^2 \\
&= \frac{1}{2}2(y_k - h_k)\frac{\partial (y_k - h_k)}{\partial h_k} \\
&= -(y_k - h_k) \\
\frac{\partial h_k}{\partial \mathrm{net}_k} &= \frac{\partial \sigma(\mathrm{net}_k)}{\partial \mathrm{net}_k} \\
&= h_k(1 - h_k) \\
\frac{\partial E_i}{\partial \mathrm{net}_k} &= -(y_k - h_k)h_k(1 - h_k) \\
\Delta w_{kj} &= -\eta\frac{\partial E_i}{\partial w_{kj}} = \eta(y_k - h_k)h_k(1 - h_k)x_{kj}
\end{aligned}$$

Derivation of the **BACKPROPAGATION** Rule III

$$
\begin{aligned}
\frac{\partial E_i}{\partial \mathrm{net}_k} &= \sum_{\ell \in \mathrm{Downstream}(k)} \frac{\partial E_i}{\partial \mathrm{net}_\ell} \frac{\partial \mathrm{net}_\ell}{\partial \mathrm{net}_k} \\
&= \sum_{\ell \in \mathrm{Downstream}(k)} -\delta_\ell \frac{\partial \mathrm{net}_\ell}{\partial \mathrm{net}_k} \\
&= \sum_{\ell \in \mathrm{Downstream}(k)} -\delta_\ell \frac{\partial \mathrm{net}_\ell}{\partial h_k} \frac{\partial h_k}{\partial \mathrm{net}_k} \\
&= \sum_{\ell \in \mathrm{Downstream}(k)} -\delta_\ell w_{\ell k} \frac{\partial h_k}{\partial \mathrm{net}_k} \\
&= \sum_{\ell \in \mathrm{Downstream}(k)} -\delta_\ell w_{\ell k} h_k (1 - h_k) \\
\delta_k &= h_k (1 - h_k) \sum_{\ell \in \mathrm{Downstream}(k)} \delta_\ell w_{\ell k} \\
\Delta w_{kj} &= \eta \delta_k x_{kj}
\end{aligned}
$$

Training rule for hidden unit weights

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

$$\Delta w_{kj}(n) = \eta\, \delta_k\, x_{kj} + \alpha\, \Delta w_{kj}\,(n-1)$$

- 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

Learning in Arbitrary Acyclic Networks

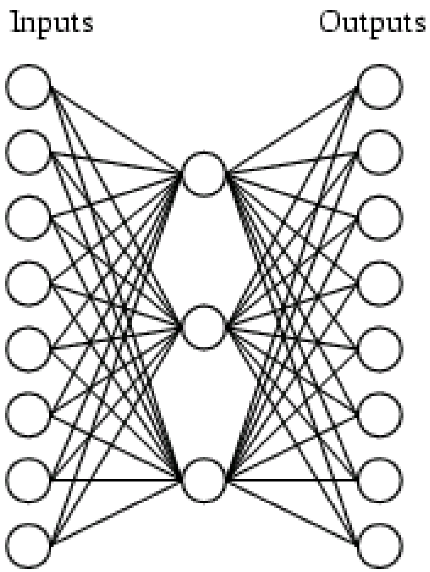
- For networks of more than two layers
  - The δ_r value for a unit r in layer m is computed from the d values at the next deeper layer $m+1$ according to

$$\delta_r = o_r(1 - o_r) \sum_{s \in \mathrm{layer}\ m+1} w_{sr}\delta_s$$

- For networks where nodes are not arranged in uniform layers
  - The δ_r value for any internal unit

$$\delta_r = o_r(1 - o_r) \sum_{s \in \mathrm{Downstream}\ m+1} w_{sr}\delta_s$$

Learning Hidden Layer Representations



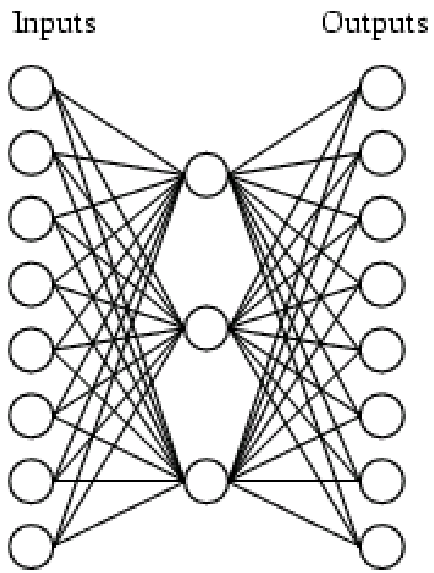
A target function:

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned??

Learning Hidden Layer Representations

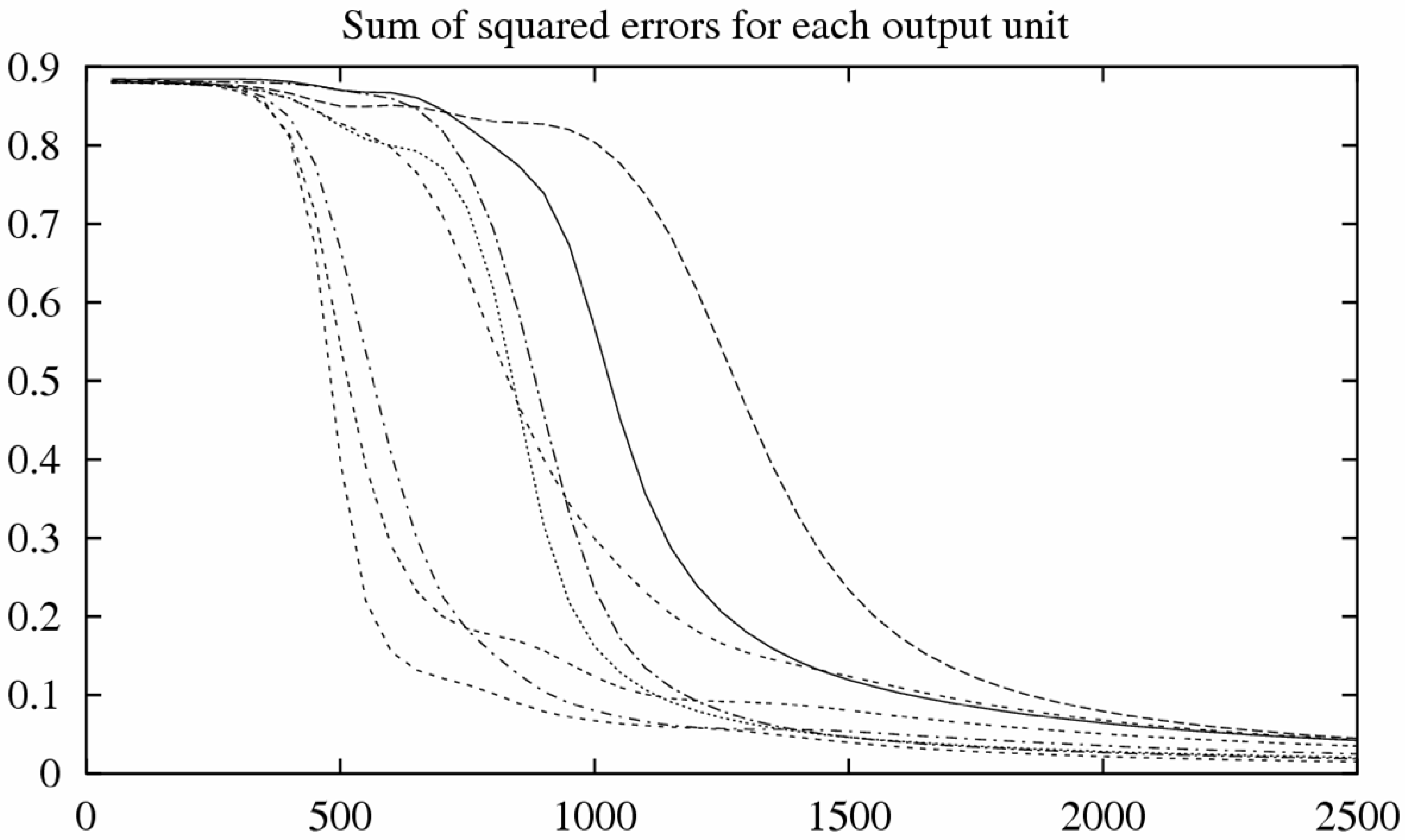
A network:



Learned hidden layer representation:

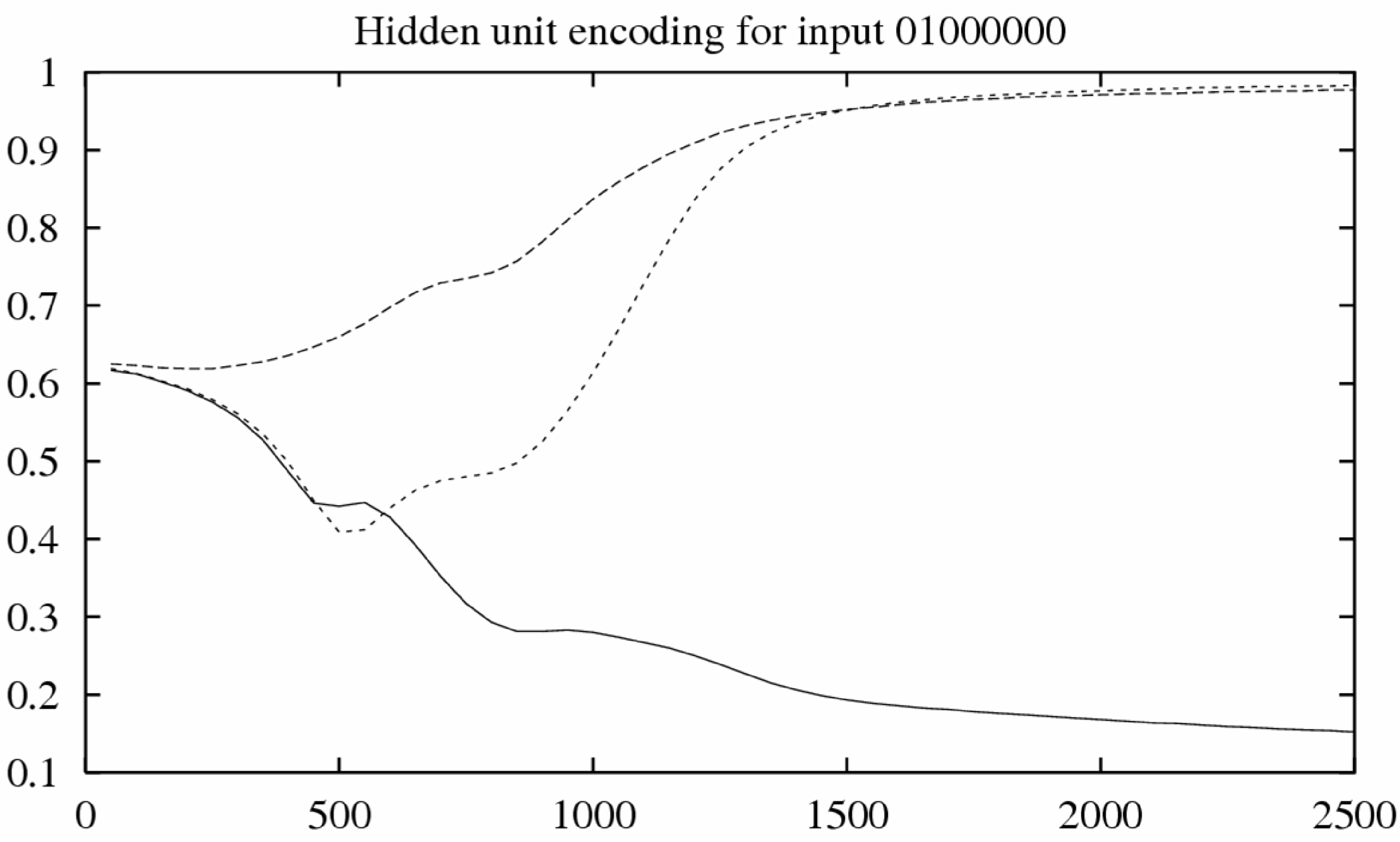
Input		Hidden Values		Output
10000000	→	.89 .04 .08	→	10000000
01000000	→	.15 .99 .99	→	01000000
00100000	→	.01 .97 .27	→	00100000
00010000	→	.99 .97 .71	→	00010000
00001000	→	.03 .05 .02	→	00001000
00000100	→	.01 .11 .88	→	00000100
00000010	→	.80 .01 .98	→	00000010
00000001	→	.60 .94 .01	→	00000001

Training



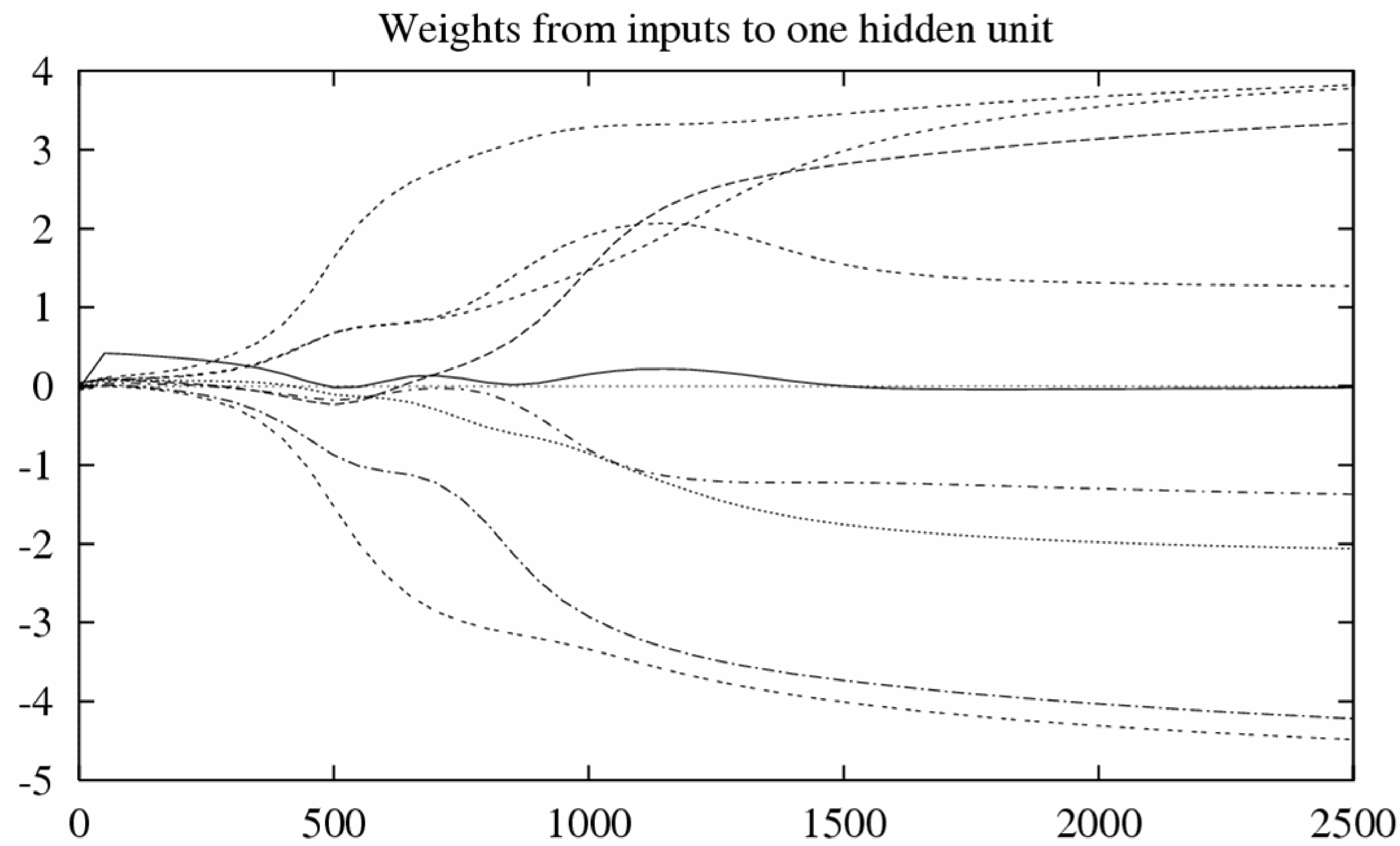
Sum of squared errors for each output unit
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500

Training



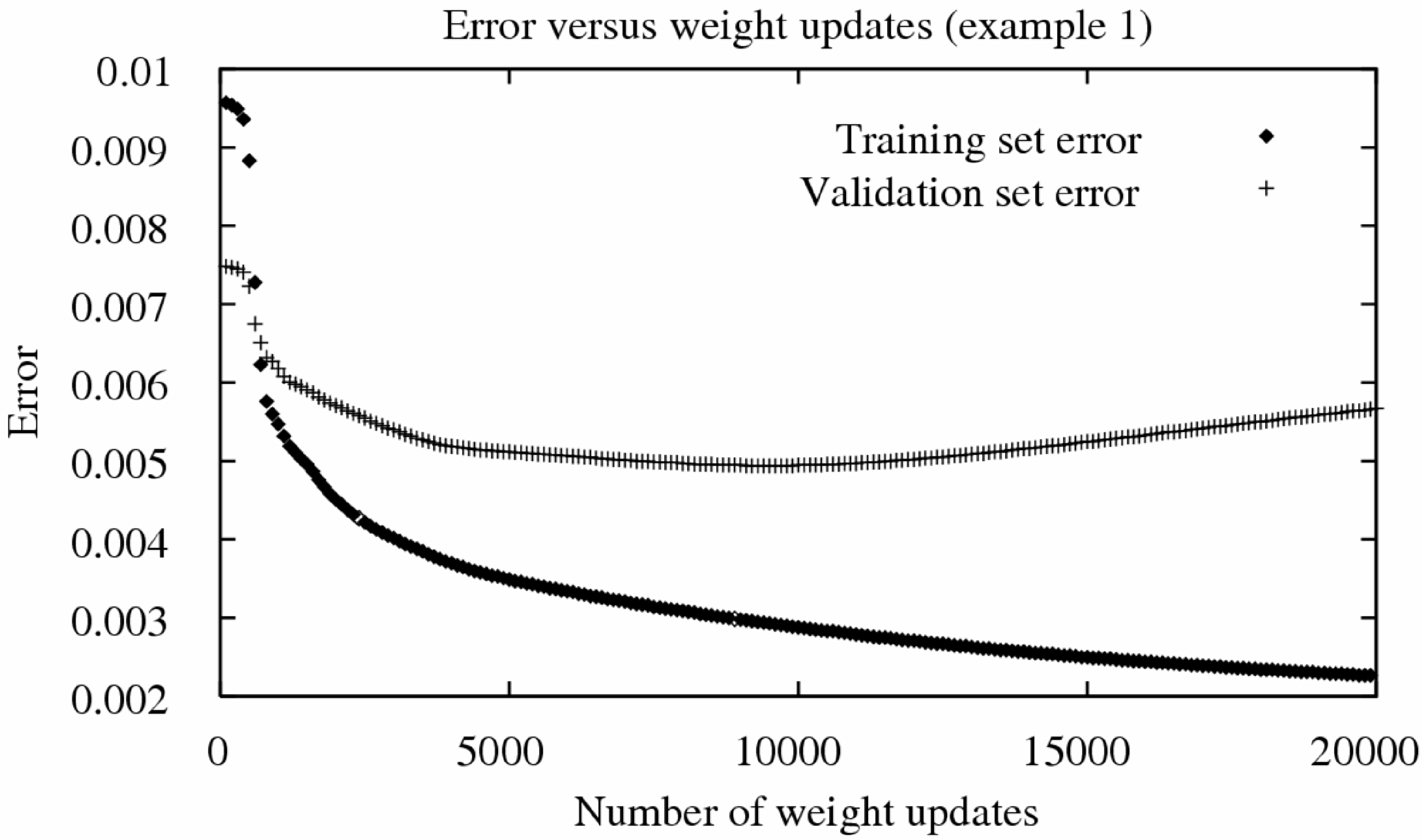
Hidden unit encoding for input 01000000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
500
1000
1500
2000
2500

Training

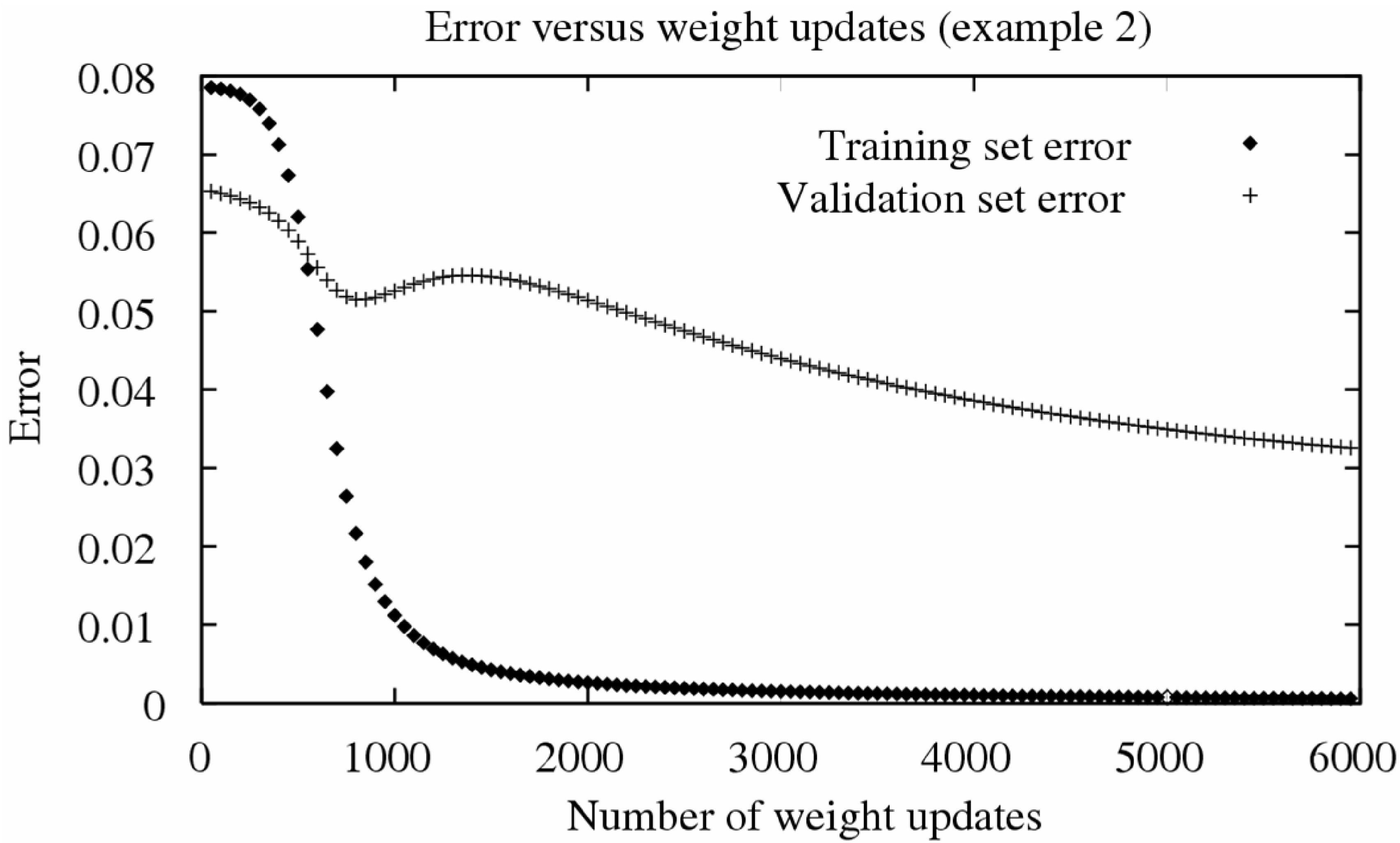


Weights from inputs to one hidden unit
4
3
2
1
0
-1
-2
-3
-4
-5
0
500
1000
1500
2000
2500

Overfitting: case I



Overfitting: case II



Convergence of Backprop

Gradient descent to some local minimum

- Perhaps not global minimum (because the function is nonlinear!)

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly nonlinear functions possible as training progresses
- Close enough to the global min. if only a local minimum

Avoid the Local Minimum

- Add momentum (through smooth area)
- Stochastic gradient descent
- Train multiple nets with different initial weights
  - Choose the best one by validation
  - Using the result from “committee”

Avoid ANN Overfitting

1. Weight decay

- Decrease each weight by a small factor during each iteration
- Plays the role of a penalty term
- [Keep weight values small]

2. Use a different validation set

- Use the number of iterations that leads to the lowest error on the validation set

Expressive Capabilities of ANN

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]

Literature & Resources

- Textbook:
  - “Neural Networks for Pattern Recognition”, Ch. 5, C. M. Bishop, 1996
  - “Machine Learning”, Ch. 4, T. M. Mitchell, 1997
- Software:
  - Neural Networks for Face Recognition
    http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html
  - SNNS Stuttgart Neural Networks Simulator
    http://www-ra.informatik.uni-tuebingen.de/SNNS
  - Neural Networks at your fingertips
    http://www.stats.gla.ac.uk/~ernest/files/NeuralAppl.html