Artificial Neural Networks

鮑興國 Ph.D.

National Taiwan University of
Science and Technology

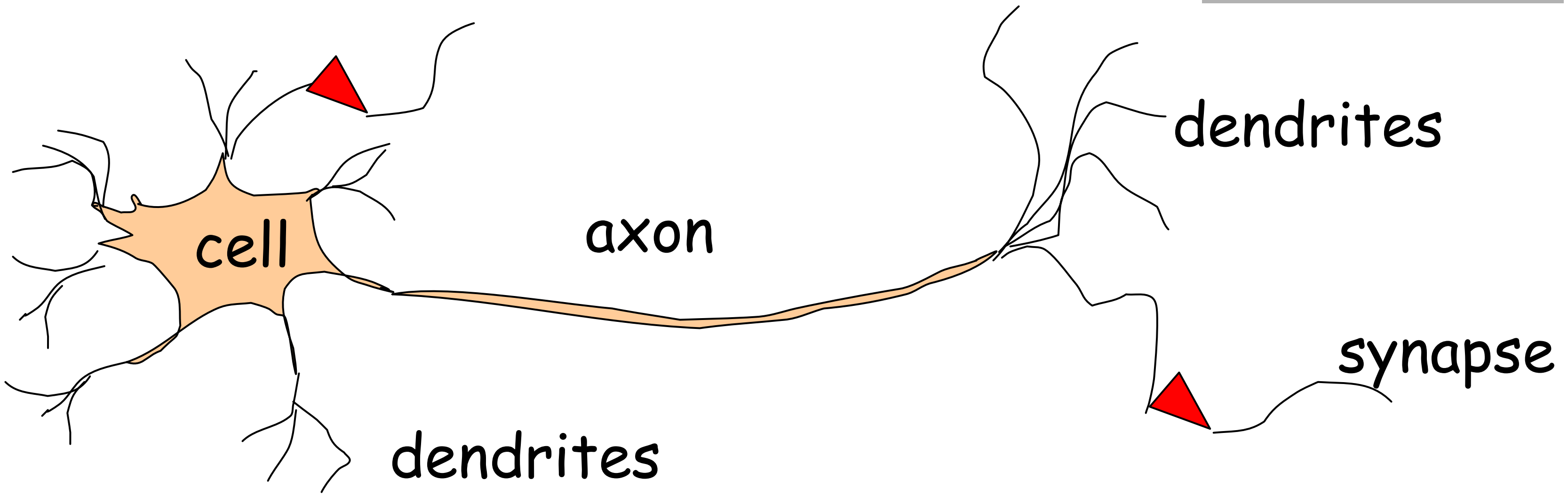
Outline

- Perceptrons
- Gradient descent
- Multi-layer 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

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: $\sim 10^{11}$
- Connections (synapses) per neuron: $\sim 10^{4}$-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

- Units interconnected in layers
  - directed, acyclic graph (DAG)
- Network structure is fixed
  - learning = weight adjustment
  - BACKPROPAGATION algorithm

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

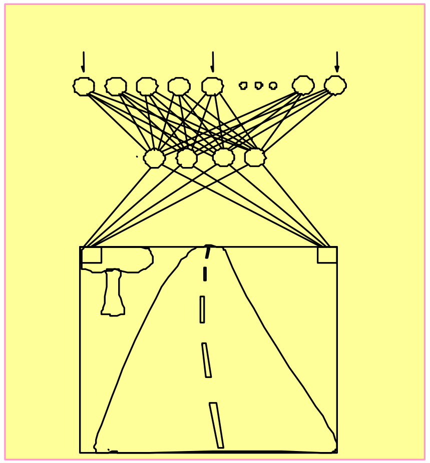
Drives 70 mph on a public highway, by ~ 5 mins training

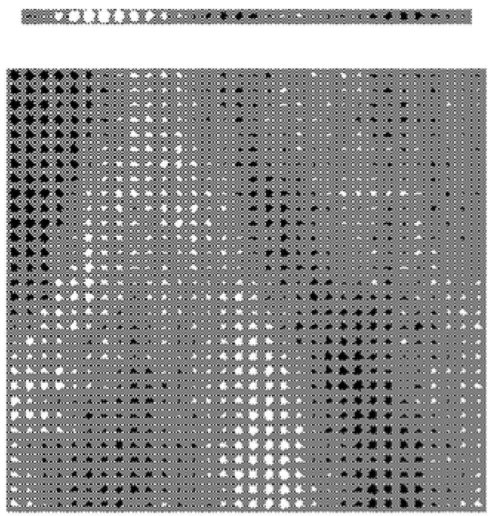
Camera image

30 outputs for steering

4 hidden units

30x32 pixels as inputs





The weights from a hidden unit to 30 output units

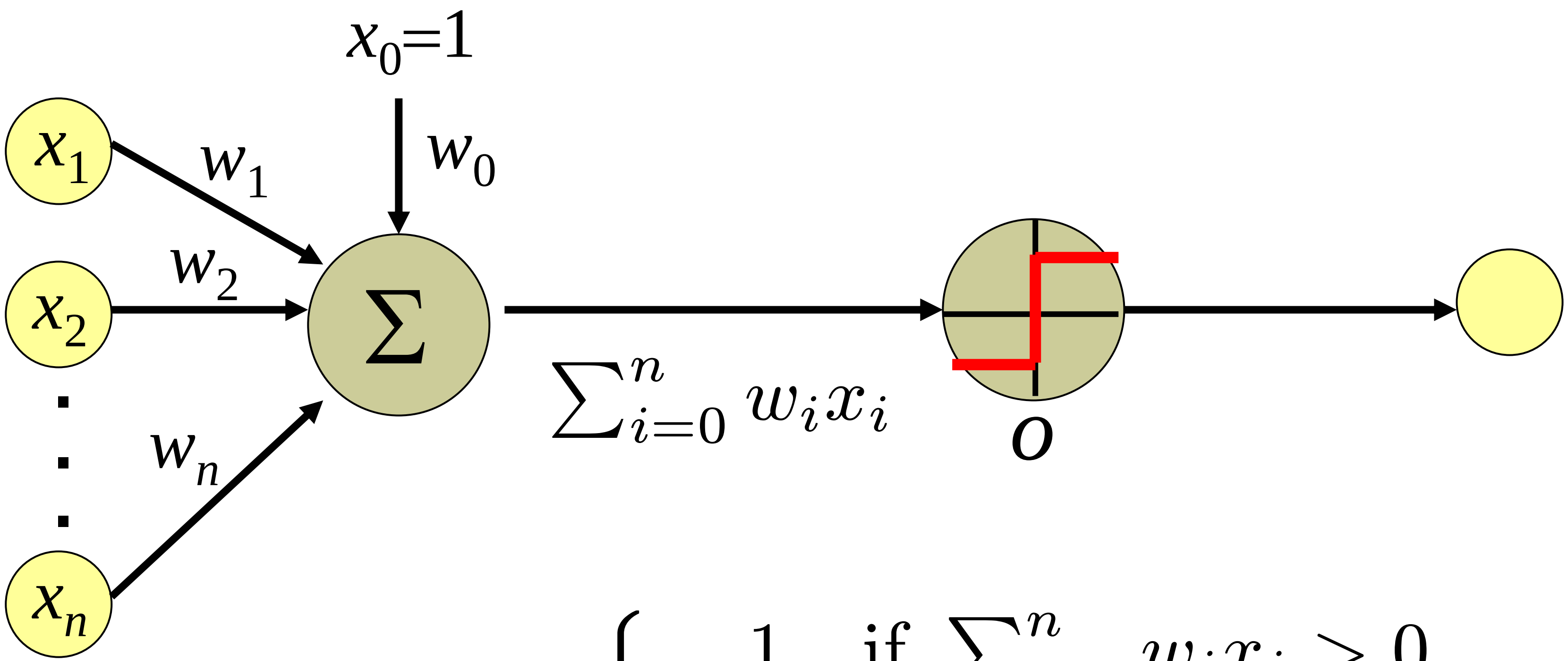
30x32 weights into one out of four hidden unit. A white box indicates a positive weight and a black box a negative weight

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)

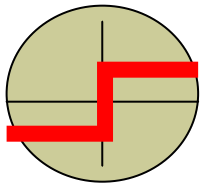


$$o = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$$

Purpose of the Activation Function *o*

- 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 *o* to be nonlinear. Otherwise, the entire neural network collapses into a simple linear function.

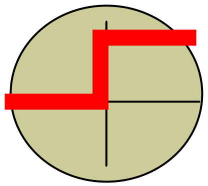
Possibilities for function o



Sign function

$\mathbf{sign}(x) = +1$, if $x > 0$

-1, if $x \le 0$

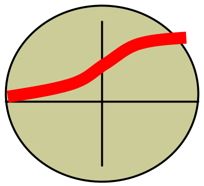


Step function

$\mathbf{step}(x) = 1$, if $x >$ threshold

0, if $x \le$ threshold

(in picture above, threshold = 0)

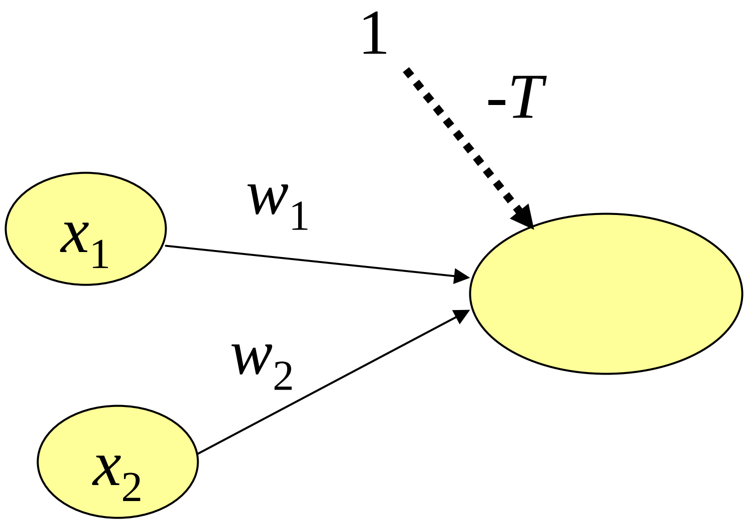


Sigmoid (logistic) function

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

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

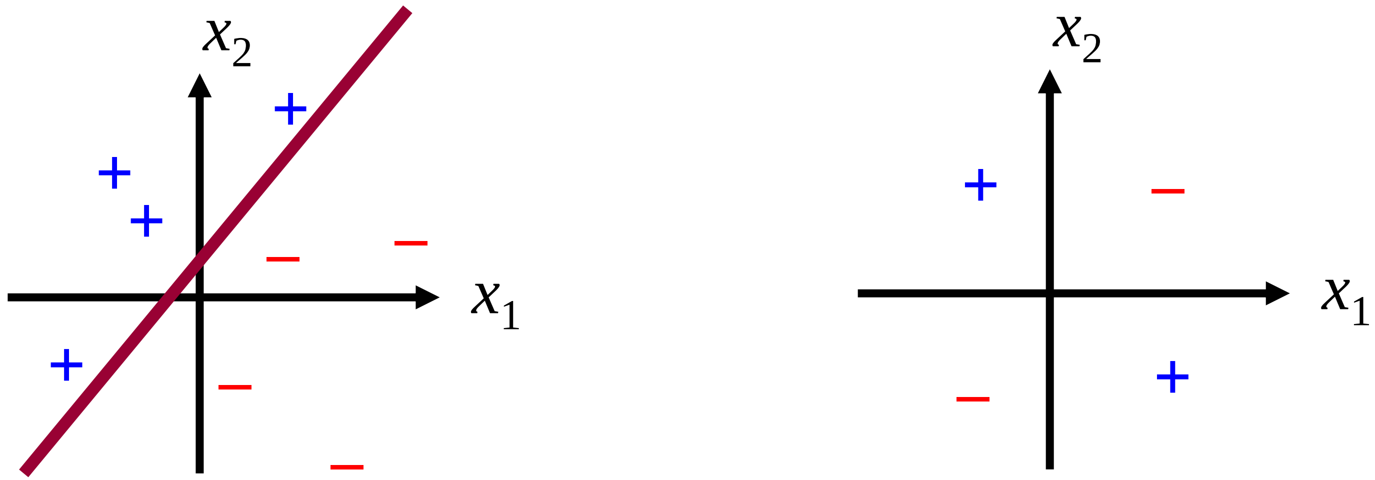
Using a Bias Weight to Standardize the Threshold



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

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

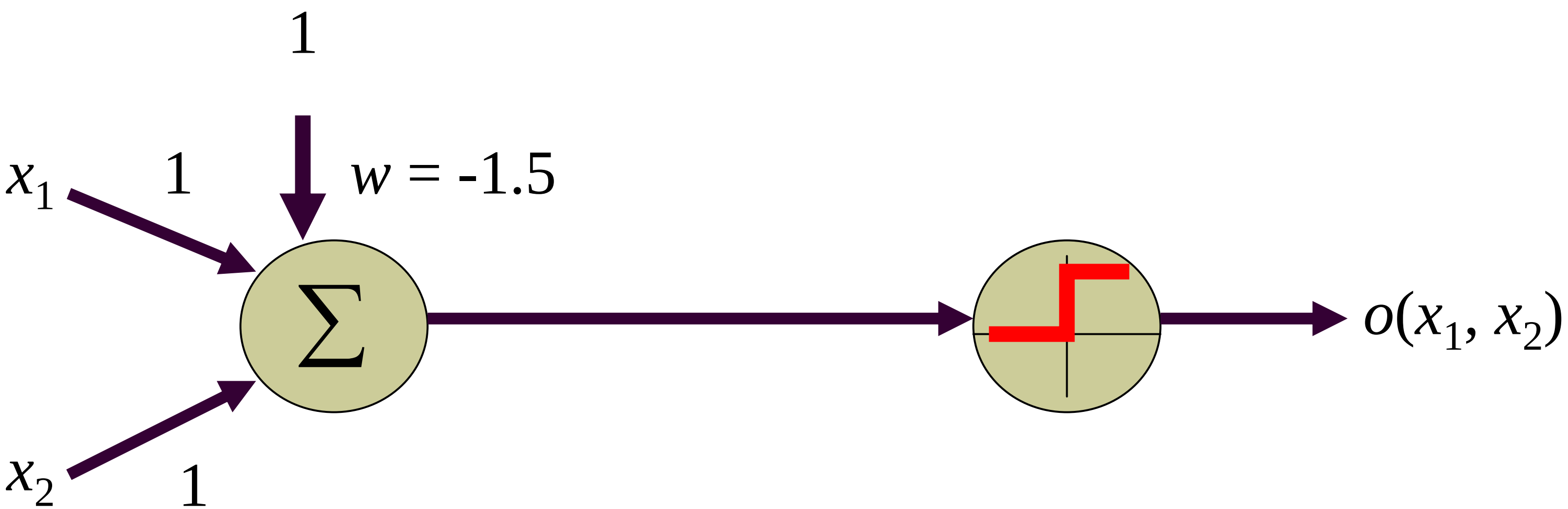
Decision Surface of a Perceptron



- 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

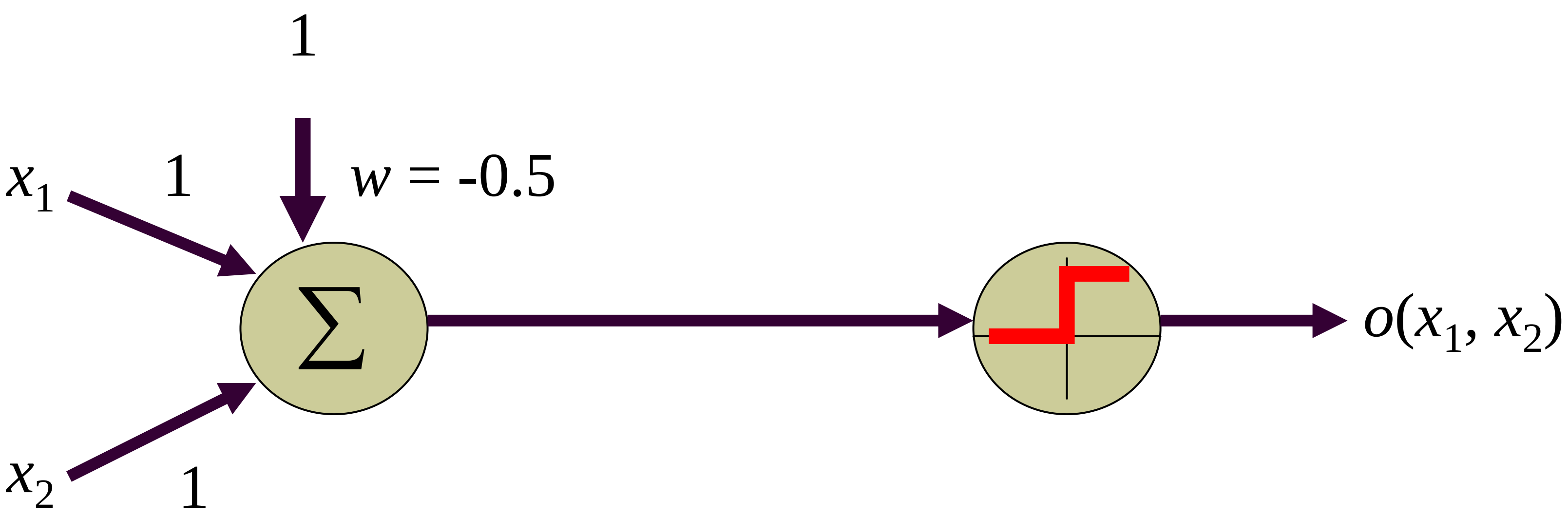
Assume Boolean (0/1) input values...



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

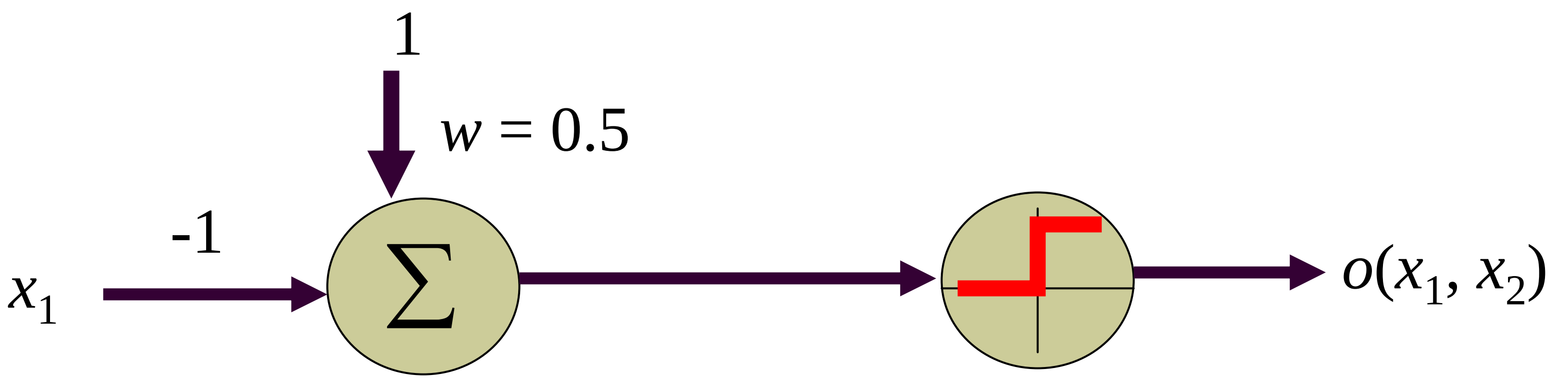
Implementing OR

Assume Boolean (0/1) input values...



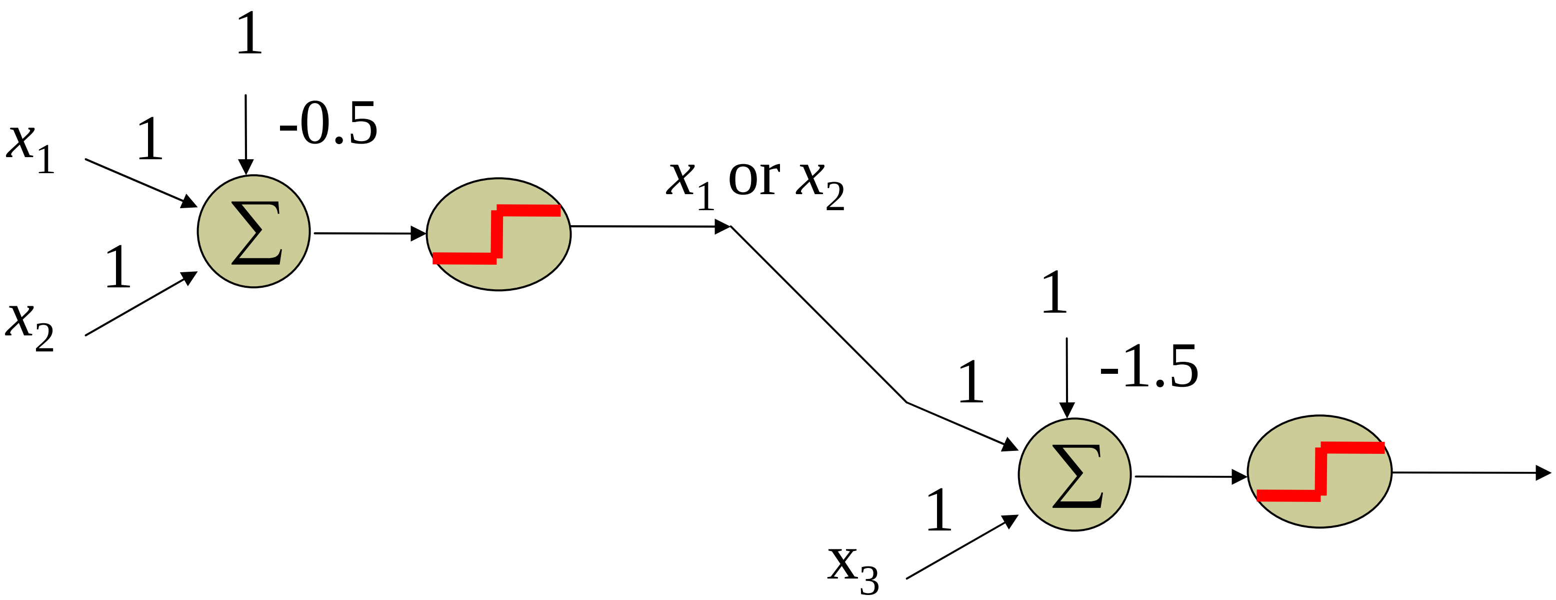
$$o(x_1,x_2) = 1 \quad \text{if } -0.5 + x_1 + x_2 > 0$$
$$= 0 \quad \text{otherwise}$$

Implementing NOT



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

Implementing more complex Boolean functions



(x_1 or x_2) and x_3

Perceptron Learning Rule

$w_i \leftarrow w_i + \Delta w_i$

$\Delta w_i = \eta \, (t - o) \, x_i$

t is the target output for the current training example

o is the perceptron output

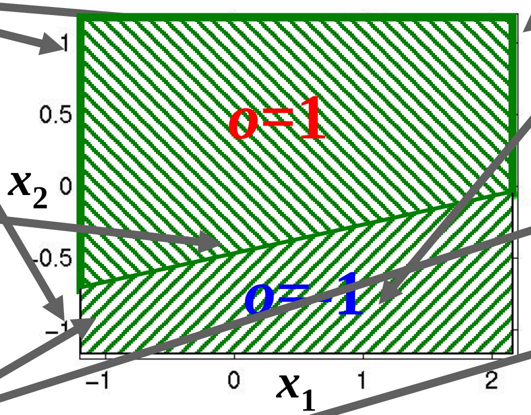
η is a small constant (e.g. 0.1) called *learning rate*

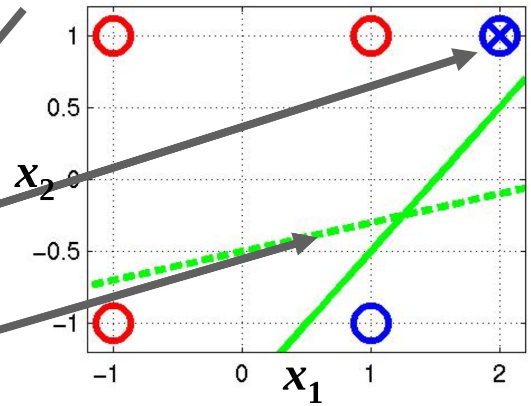
- Start with some random weights (usually small values)
- If the output is correct ($t = o$) the weights w_i are not changed
- If the output is incorrect ($t \neq o$) the weights w_i are changed such that the output of the perceptron for the new weights is *closer* to t.
- The algorithm converges to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

Perceptron Learning Rule

$t = 1$

$t = -1$





$w = [0.25\ –0.1\ 0.5]$

$x_2 = 0.2\ x_1 – 0.5$

$(x, t)=([-1,-1],1)$

$(x, t)=([2,1],–1)$

$o = \mathrm{sgn}(0.25+0.1–0.5)$

$o = \mathrm{sgn}(0.45–0.6+0.3)$

$=–1$

$=1$

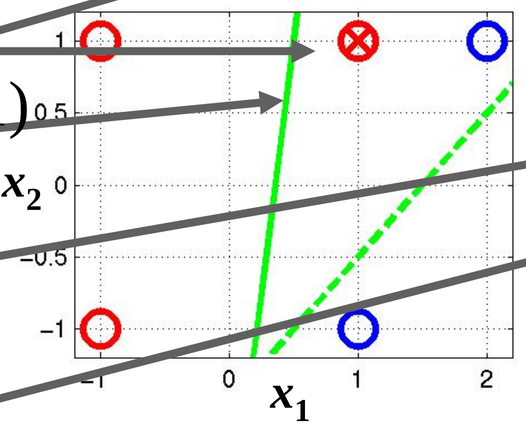
$(x, t)=([1,1],1)$

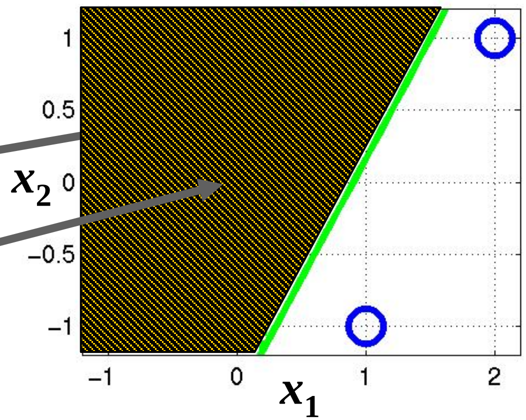
$o = \mathrm{sgn}(0.25–0.7+0.1)$

$=–1$

$\Delta w = [0.2\ –0.2\ –0.2]$

$\Delta w = [–0.2\ –0.4\ –0.2]$





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

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

Gradient Descent Learning Rule

- Perceptron learning rule fails to converge if examples are not linearly separable
- 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
  - $E[w_1, \ldots, w_n] = ½ \sum_{d \in D} (t_d - o_d)^2$

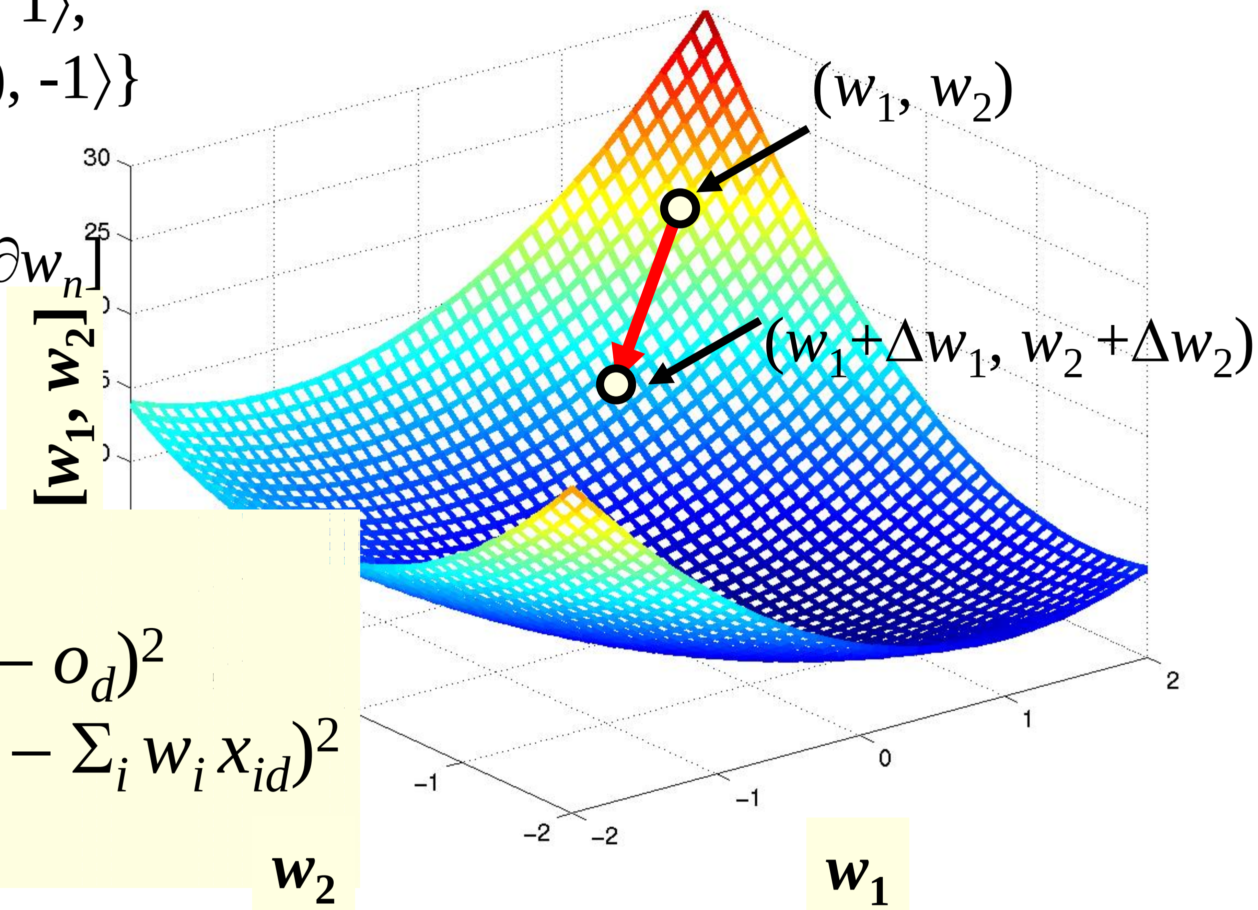
  where D is the set of training examples

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

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



$$\begin{aligned}\Delta w_i &= -\eta\, \partial E/\partial w_i \\ &= -\eta\, \partial/\partial w_i\ \tfrac{1}{2}\Sigma_d(t_d - o_d)^2 \\ &= -\eta\, \partial/\partial w_i\ \tfrac{1}{2}\, \Sigma_d(t_d - \Sigma_i\, w_i\, x_{id})^2 \\ &= \eta\, \Sigma_d(t_d - o_d)\, x_{id}\end{aligned}$$

Gradient Descent

- Train the w_i's such that they minimize the squared error
  - $E[w_1,\ldots, w_n] = ½ \, \Sigma_{d\in D} \, (t_d - o_d)^2$

Gradient:

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

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

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

$= -\eta \, \partial/\partial w_i \, ½\Sigma_d(t_d - o_d)^2$

$= -\eta \, \partial/\partial w_i \, ½\Sigma_d(t_d - \Sigma_{\mathrm{i}} \, w_i \, x_i)^2$

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

Gradient Descent

Gradient-Descent(*training_examples*, η)

Each training example is a pair of the form $\langle(x_1, \ldots, x_n), t\rangle$ where $(x_1,\ldots, x_n)$ 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 $\langle(x_1, \ldots, x_n), t\rangle$ in *training_examples* Do
    - Input the instance $(x_1, \ldots, x_n)$ 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 w_i Do
    - $w_i = w_i + \Delta w_i$
- Termination condition – error falls under a given threshold

Perceptron Learning

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

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

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

$$w_i \Leftarrow w_i + \eta(t - o)x_i$$

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

5. Repeat from Step 2 until done

Gradient Descent Learning

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

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

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

$$w_i \Leftarrow w_i + \eta \sum_{d \in D} (t_d - o_d) x_{id}$$

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

  $E_D[w] = ½ \sum_d (t_d - o_d)^2$

- Incremental mode: gradient descent

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

  $E_d[w] = ½ (t_d - o_d)^2$

- 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

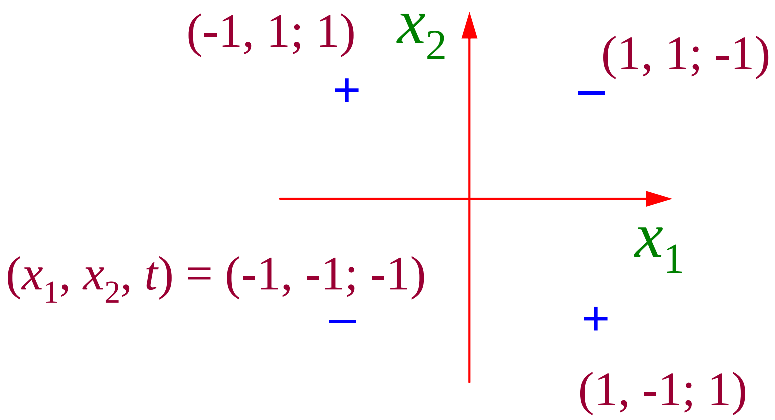
Gradient descent learning rules uses gradient descent

- 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

$$
\begin{aligned}
o(\vec{x}) &= \vec{w} \cdot \vec{x} \\
E(\vec{w}) &= \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \left[(-1 - w_1 - w_2)^2 + (1 + w_1 - w_2)^2 + (-1 + w_1 + x_2)^2 + (1 - w_1 + w_2)^2 \right] \\
&= 2(1 + w_1^2 + w_2^2)
\end{aligned}
$$

- The error will reach
  the minimum 2 when $w_1 = w_2 = 0$
- 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!



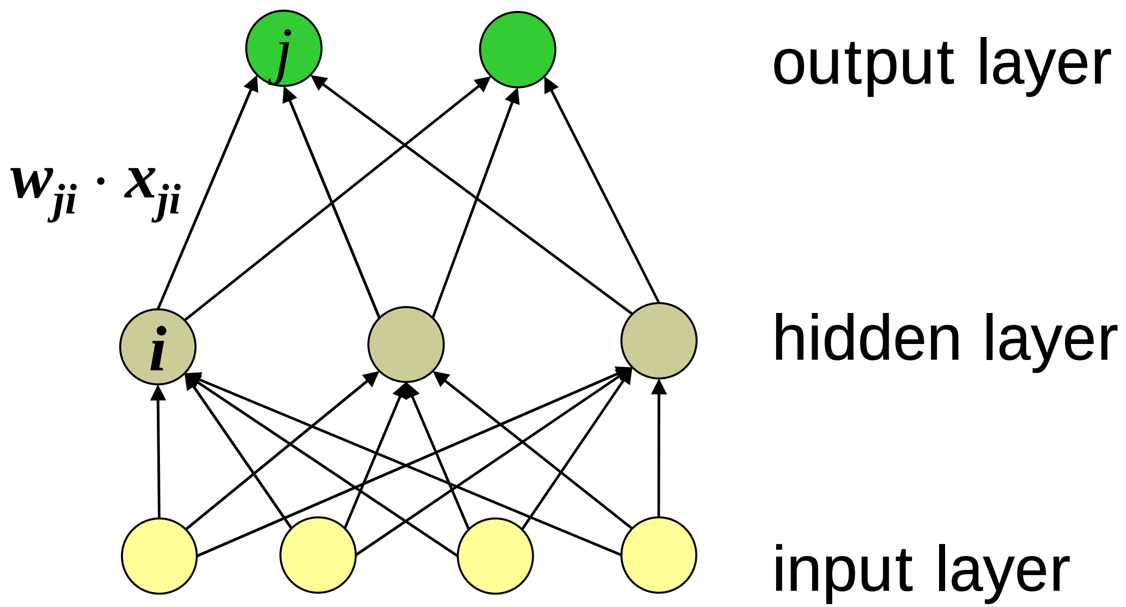
Limitations of Threshold and Perceptron Units

Limitations of Threshold and Perceptron Units

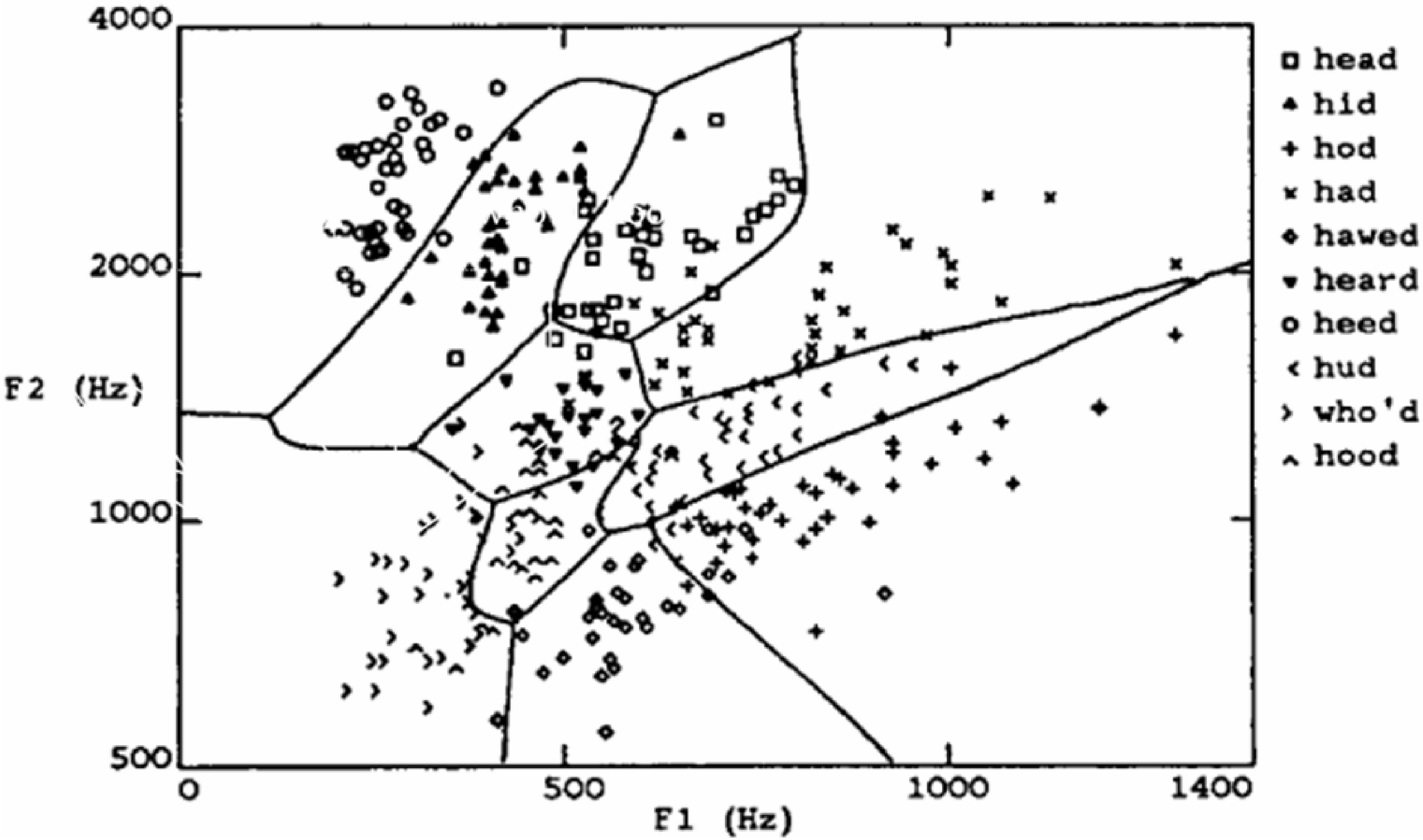
- Perceptrons can only learn linearly separable classes
- Perceptrons cycle if classes are not linearly separable
- Threshold units converge always to MSE hypothesis
- Network of perceptrons – how to train?
- Network of threshold units – not necessary! (why?)

Multi-Layer Networks

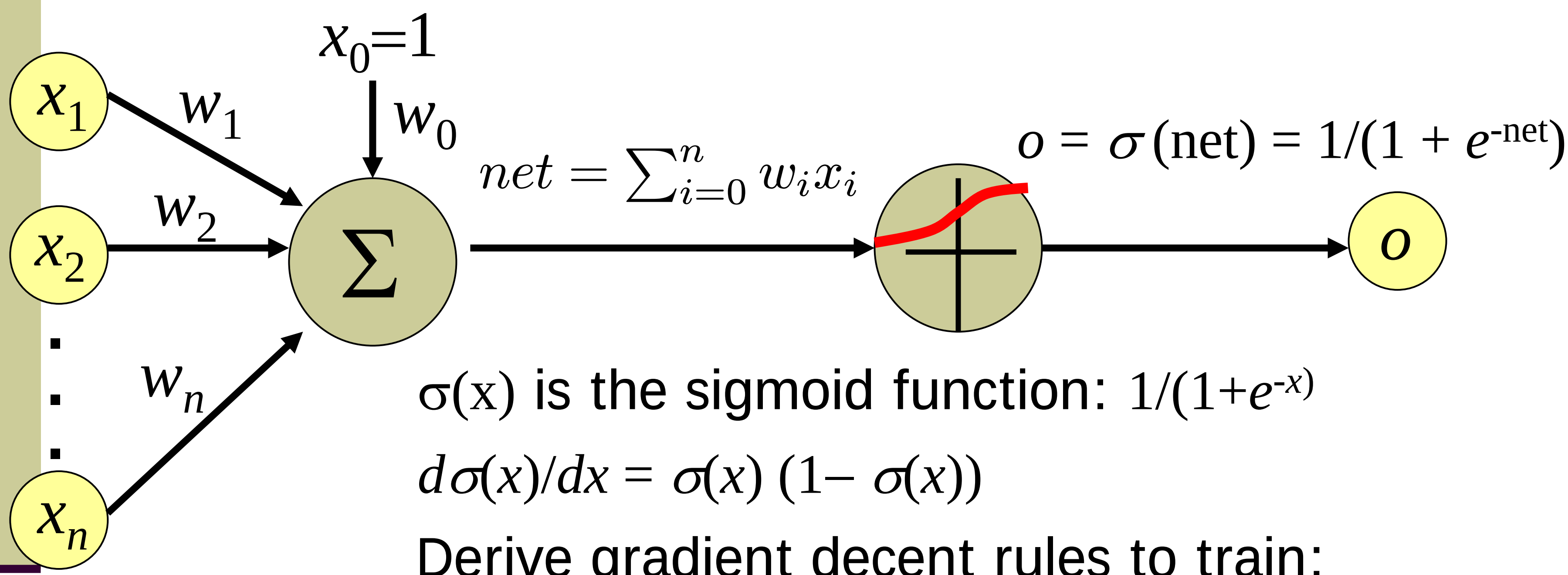
- Single perceptrons can only express linear decision surfaces
- On the other hand, multilayer networks are capable of expressing a rich variety of nonlinear decision surfaces



A Speech Recognition Task



Sigmoid Threshold 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

  $\partial E/\partial w_i = -\sum_d (t_d - o_d)\, o_d\,(1 - o_d)\, x_i$

- Multilayer networks of sigmoid units
  backpropagation:

BACKPROPAGATION Algorithm

Initialize each w_i to some small random value

Until the termination condition is met, Do

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

Input the instance $(x_1, \ldots, x_n)$ to the network and compute the network outputs o_k

For each output unit k

$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$

For each hidden unit h

$$\delta_h = o_h(1 - o_h) \Sigma_k w_{h,k} \delta_k$$

For each network weight $w_{,j}$ Do

$w_{i,j} = w_{i,j} + \Delta w_{i,j}$ where

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

Derivation of the **BACKPROPAGATION** Rule I

$$E_d(\vec{w}) \equiv \frac{1}{2}\sum_{k\in \mathrm{outputs}}(t_k - o_k)^2$$

$$\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}}$$

- x_{ji}: the ith input to unit j
- w_{ji}: the weight associated with the ith input to unit j
- net_j: $\Sigma_i\, w_{ji}\, x_{ji}$ (the weighted sum of inputs for unit j)
- o_j: the output computed by unit j
- t_j: the target output for unit j
- σ: the sigmoid function
- outputs: the set of units in the final layer of the network
- Downstream(j): the set of units whose immediate inputs include the output of unit j

Derivation of the **BACKPROPAGATION** Rule II

$$\begin{aligned}\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial \mathrm{net}_j}\frac{\partial \mathrm{net}_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial \mathrm{net}_j} x_{ji}\end{aligned}$$

Training rule for output unit weights:

$$\begin{aligned}\frac{\partial E_d}{\partial \mathrm{net}_j} &= \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net}_j} \\ \frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in \mathrm{outputs}}(t_k - o_k)^2 \\ \frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial (t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j) \\ \frac{\partial o_j}{\partial \mathrm{net}_j} &= \frac{\partial \sigma(\mathrm{net}_j)}{\partial \mathrm{net}_j} \\ &= o_j(1 - o_j) \\ \frac{\partial E_d}{\partial \mathrm{net}_j} &= -(t_j - o_j)o_j(1 - o_j) \\ \Delta w_{ji} &= -\eta\frac{\partial E_d}{\partial w_{ji}} = \eta(t_j - o_j)o_j(1 - o_j)x_{ji}\end{aligned}$$

Derivation of the **BACKPROPAGATION** Rule III

$$
\begin{aligned}
\frac{\partial E_d}{\partial \text{net}_j} &= \sum_{k \in \text{Downstream}(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} \\
&= \sum_{k \in \text{Downstream}(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} \\
&= \sum_{k \in \text{Downstream}(j)} -\delta_k \frac{\partial \text{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \\
&= \sum_{k \in \text{Downstream}(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial \text{net}_j} \\
&= \sum_{k \in \text{Downstream}(j)} -\delta_k w_{kj} o_j (1 - o_j) \\
\delta_j &= o_j (1 - o_j) \sum_{k \in \text{Downstream}(j)} \delta_k w_{kj} \\
\Delta w_{ji} &= \eta \delta_j x_{ji}
\end{aligned}
$$

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

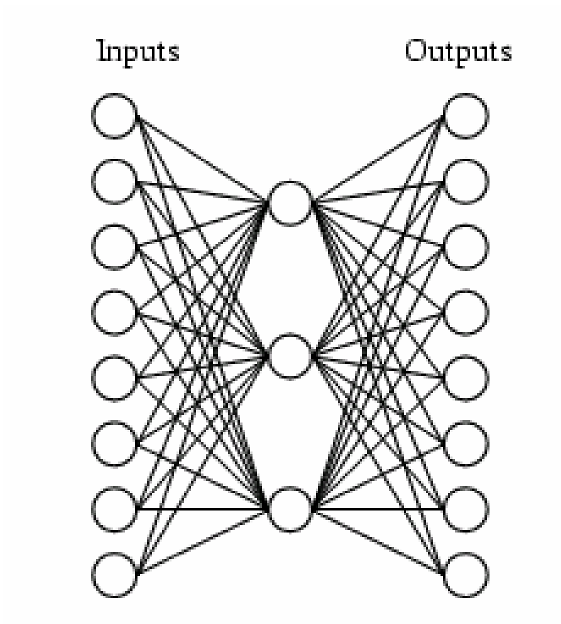
$$\Delta w_{i,j}(n) = \eta\, \delta_j\, x_{i,j} + \alpha\, \Delta w_{i,j}\,(n-1)$$

- 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

Learning Hidden Layer Representations



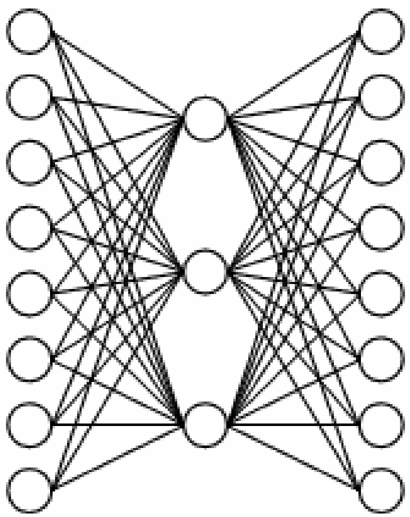
A target function:

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

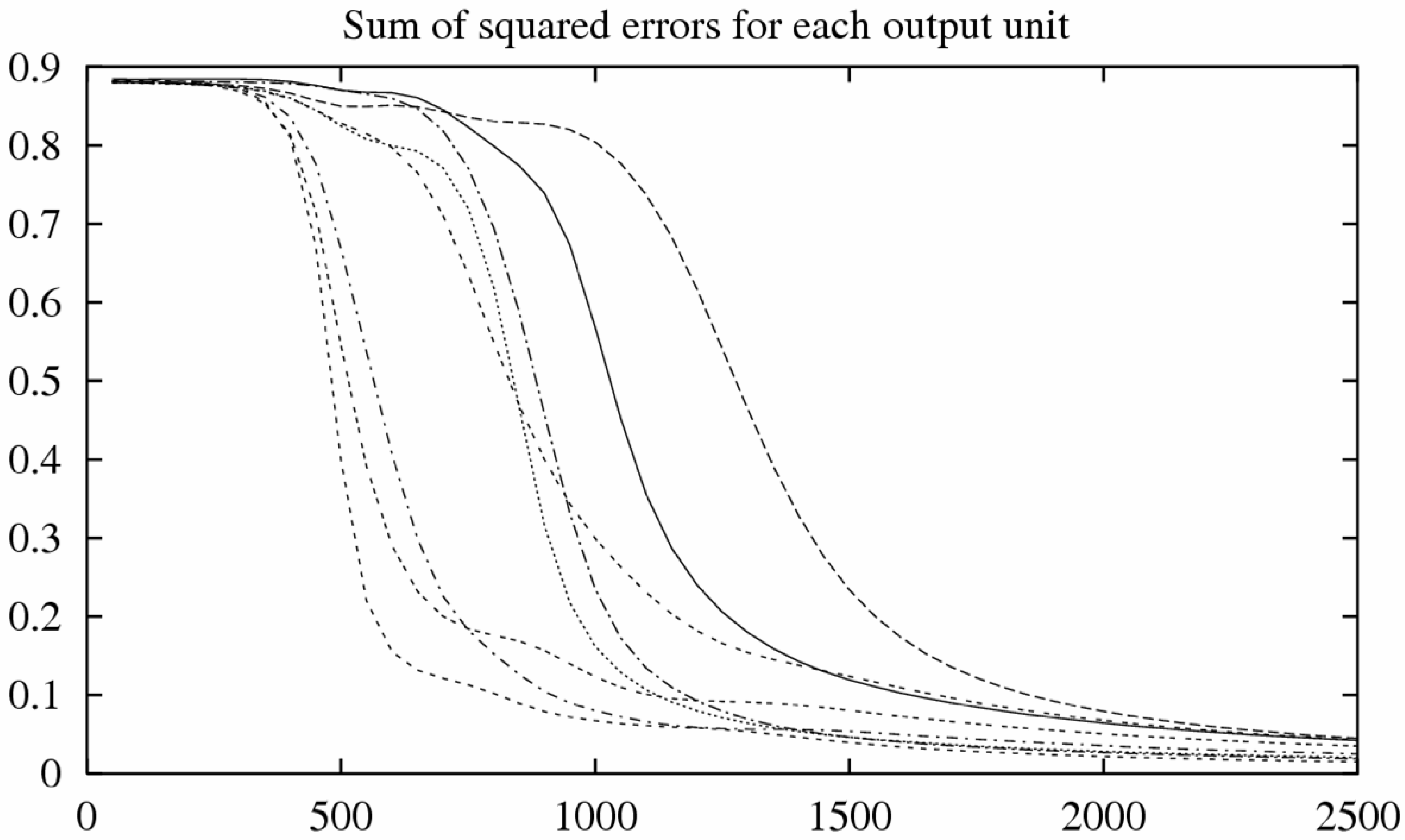
A network:



Learned hidden layer representation:

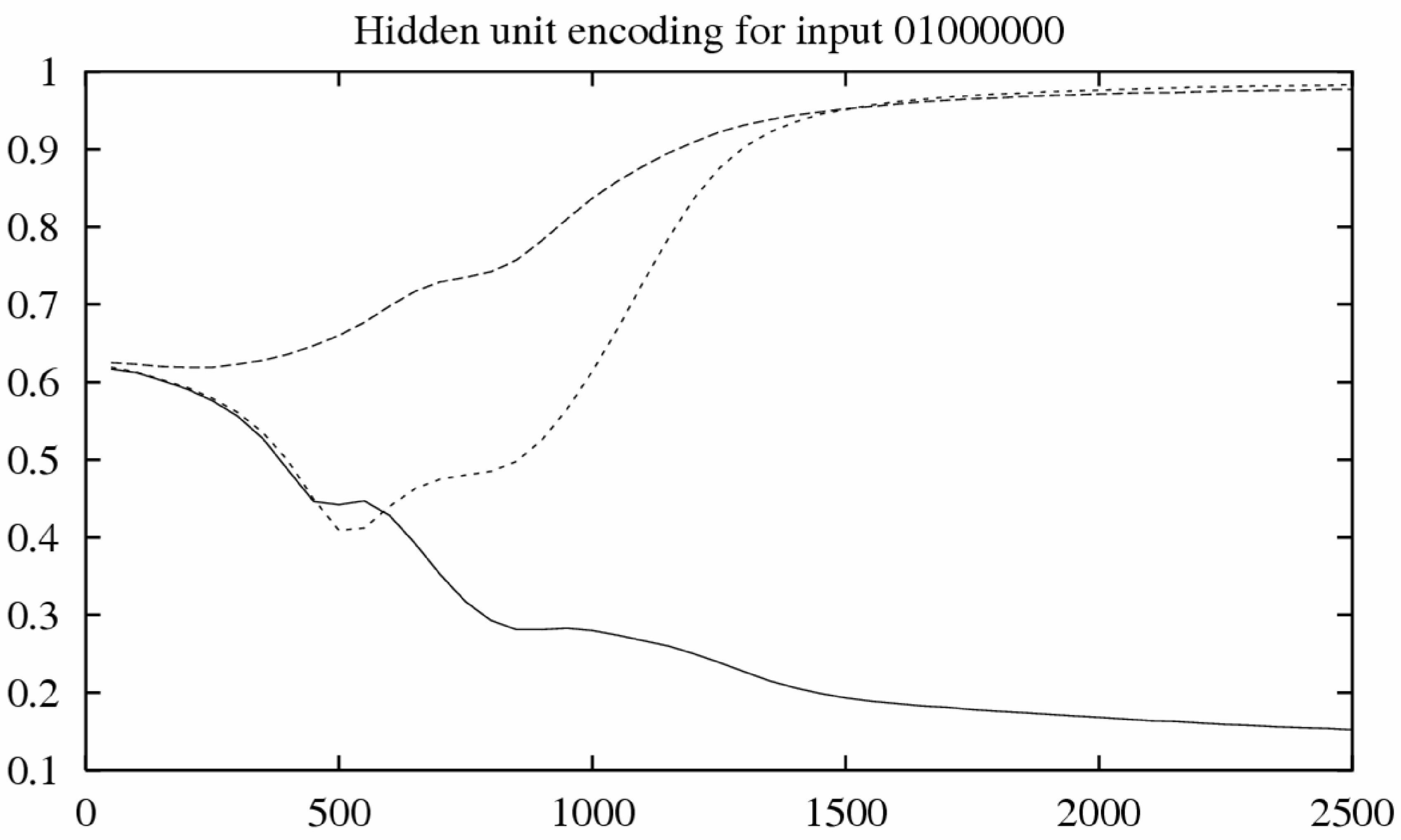
Input		Hidden Values		Output
10000000	→	.89 .04 .08	→	10000000
01000000	→	.15 .99 .99	→	01000000
00100000	→	.01 .97 .27	→	00100000
00010000	→	.99 .97 .71	→	00010000
00001000	→	.03 .05 .02	→	00001000
00000100	→	.01 .11 .88	→	00000100
00000010	→	.80 .01 .98	→	00000010
00000001	→	.60 .94 .01	→	00000001

Training

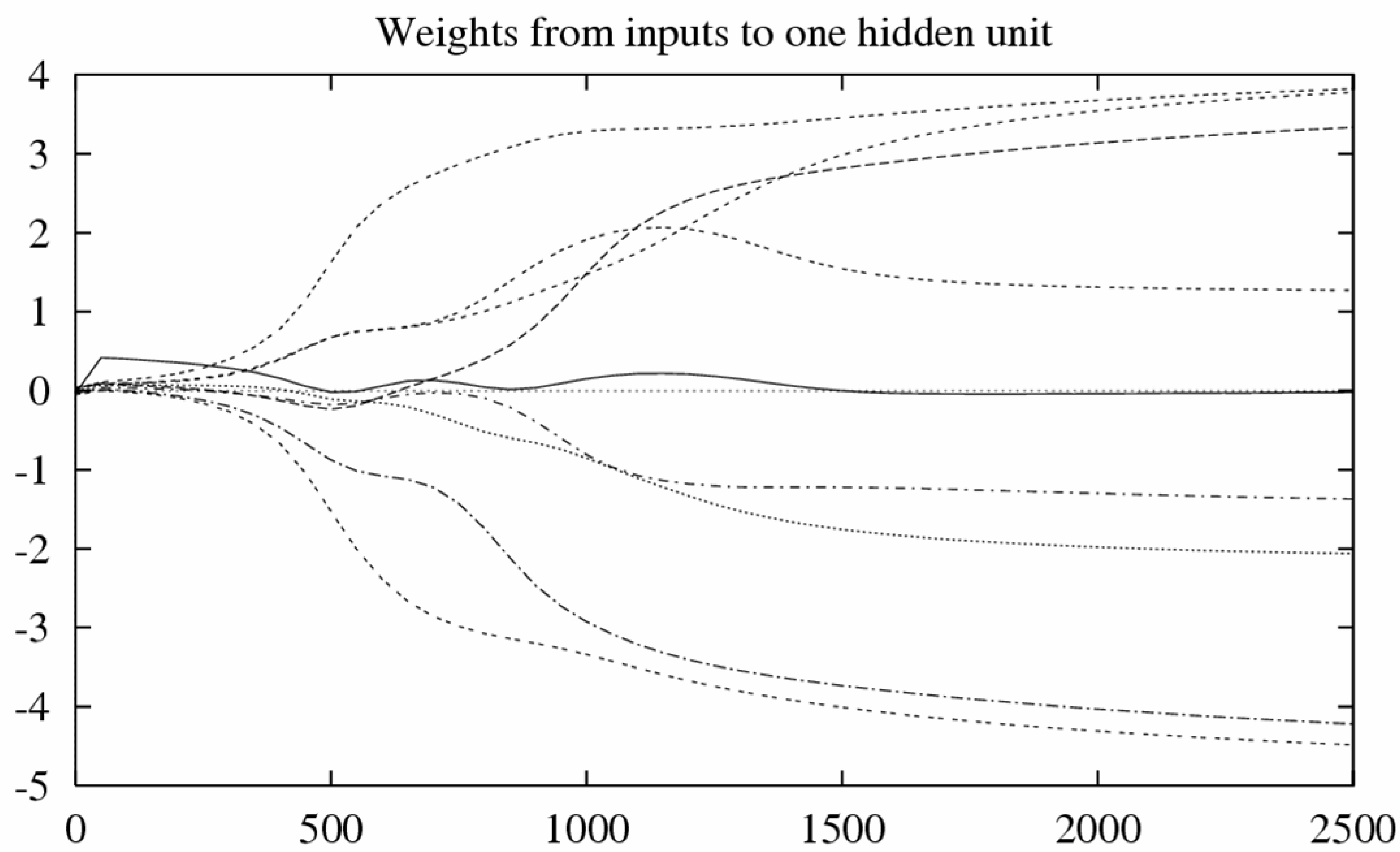


Sum of squared errors for each output unit
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500

Training

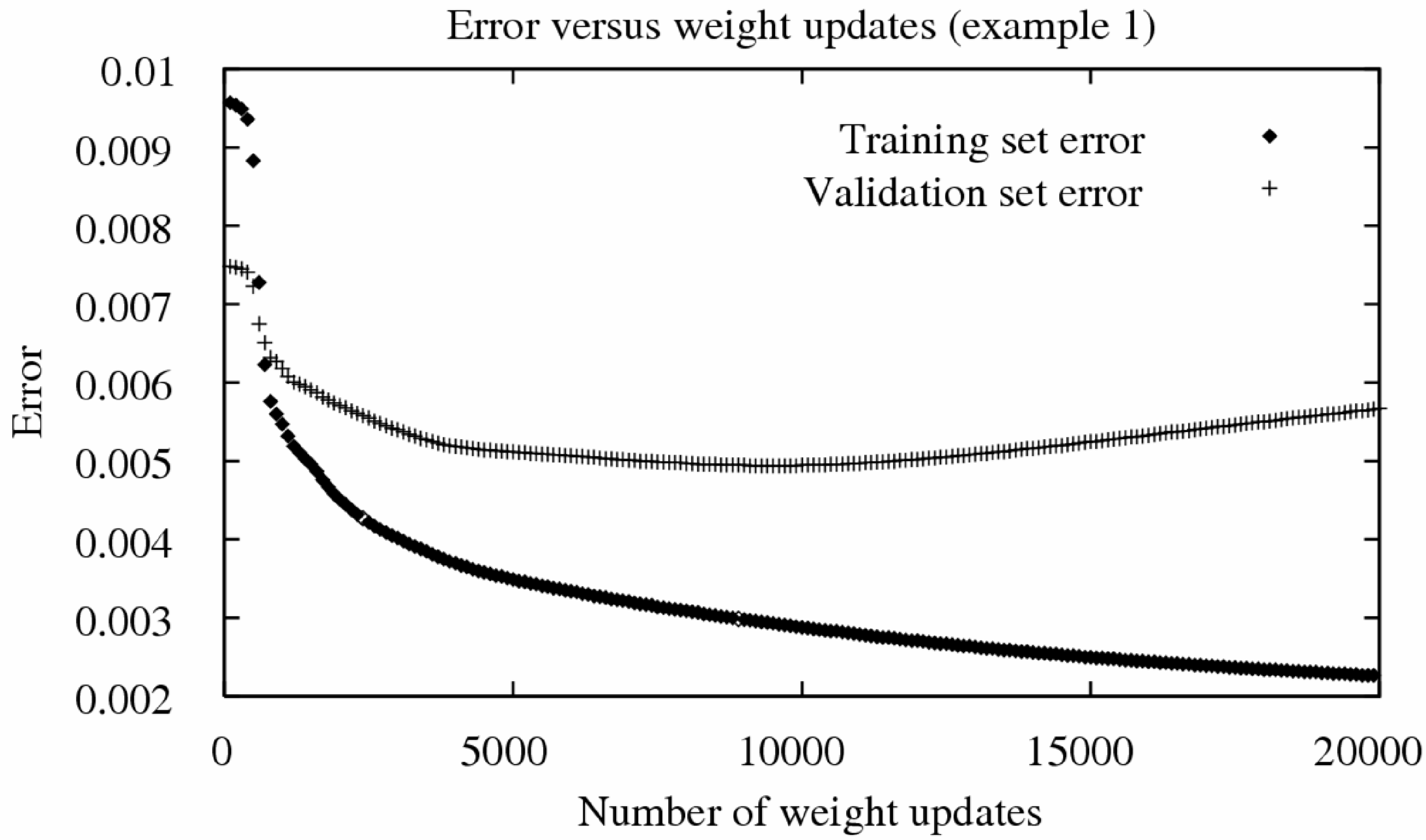

Hidden unit encoding for input 01000000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
500
1000
1500
2000
2500

Training

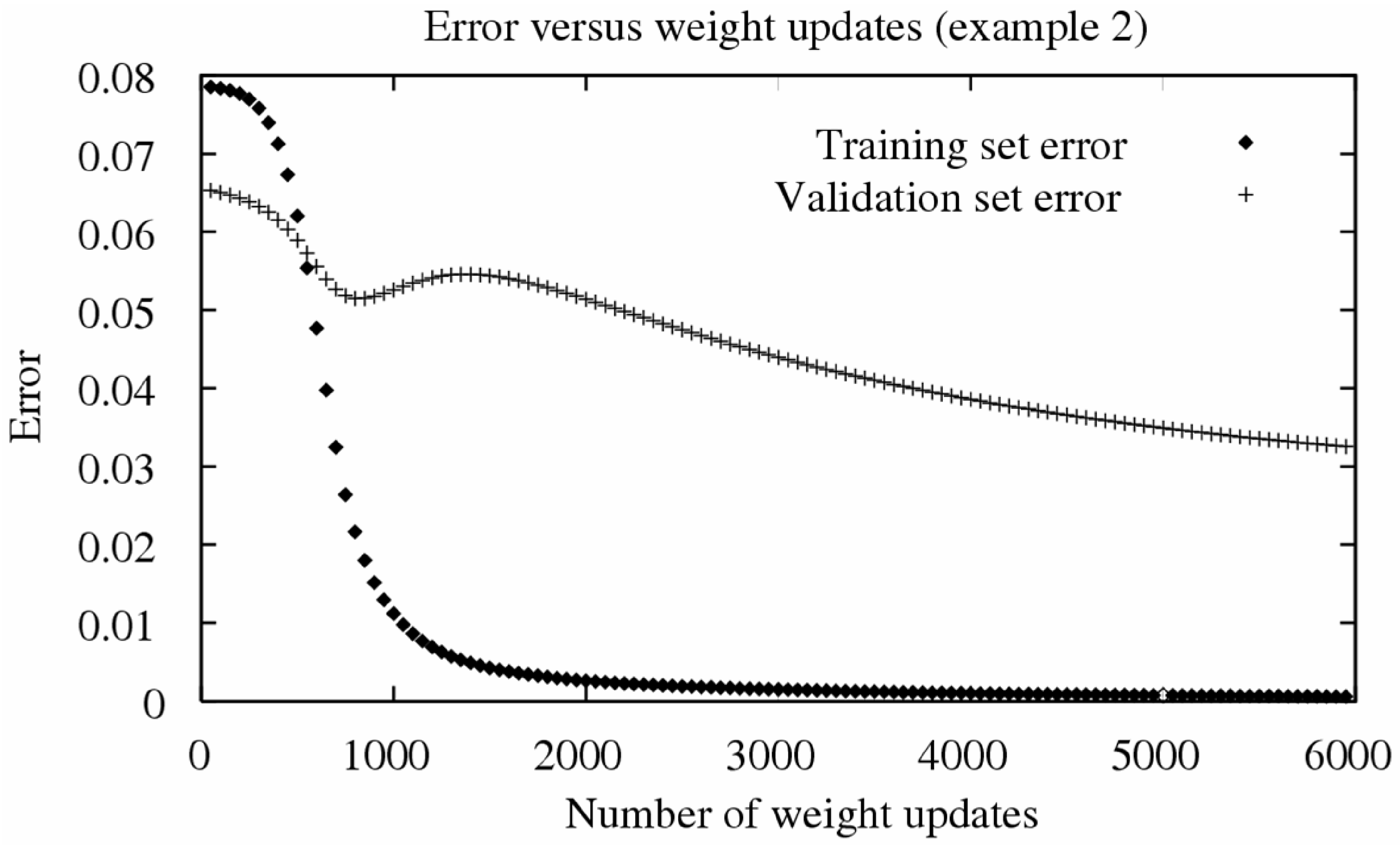


Weights from inputs to one hidden unit
4
3
2
1
0
-1
-2
-3
-4
-5
0
500
1000
1500
2000
2500

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 non-linear 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”, C. M. Bishop, 1996
  - “Machine Learning”, 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