

Artificial Neural Networks

Note: The core of the material presented here has been borrowed from the slides prepared by Pedro Domingos. Minor customization has been done to suit the specific needs of the course.

Preview

- Perceptrons
- Gradient descent
- Multilayer networks
- Backpropagation

Connectionist Models

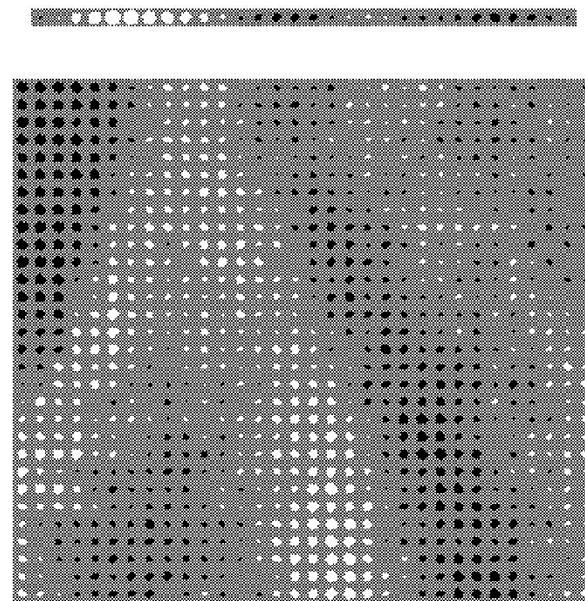
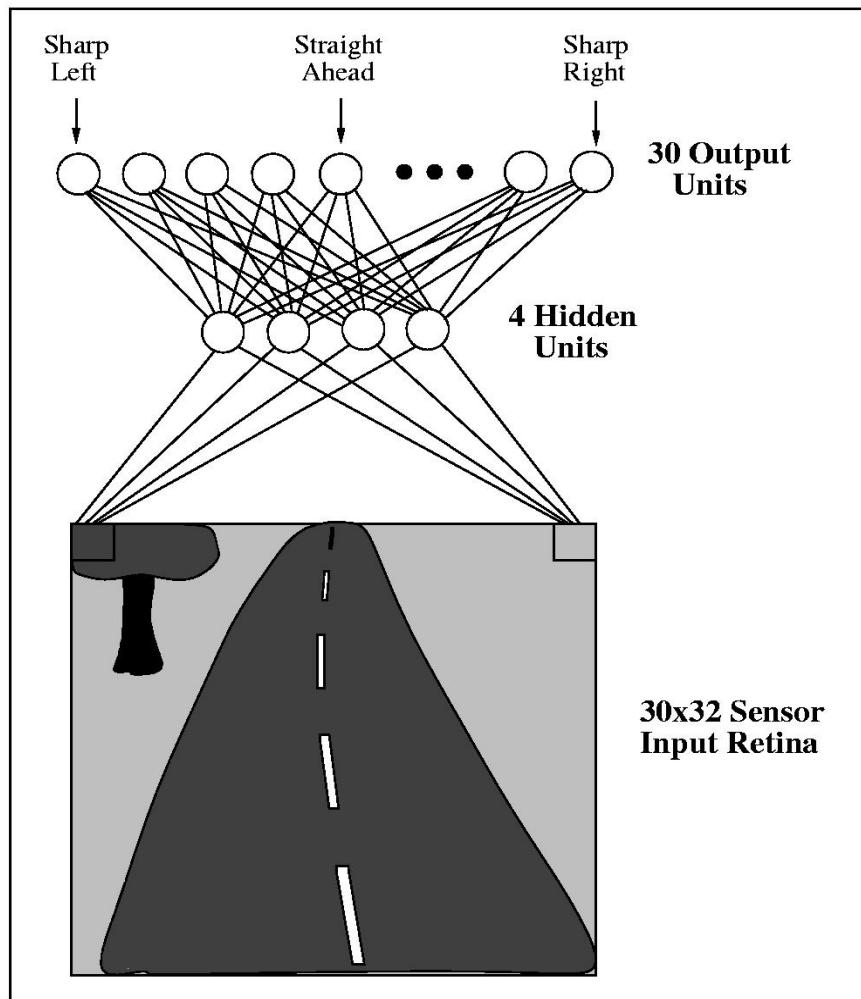
Consider humans:

- Neuron switching time $\sim .001$ second
- Number of neurons $\sim 10^{10}$
- Connections per neuron $\sim 10^{4-5}$
- Scene recognition time $\sim .1$ second
- 100 inference steps doesn't seem like enough
⇒ Much parallel computation

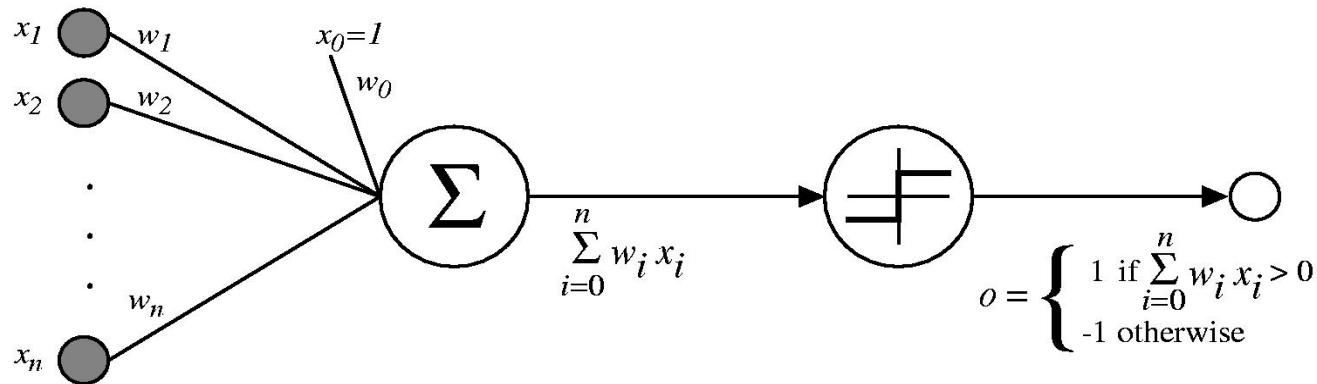
Properties of neural nets:

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically





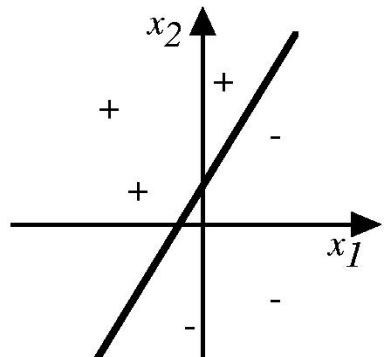
Perceptron



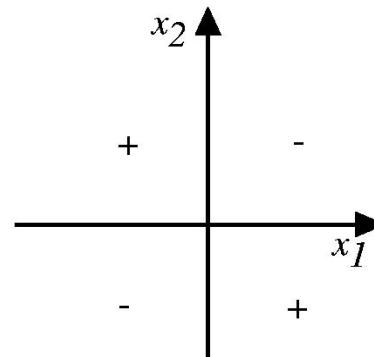
Sometimes we'll use simpler vector notation:

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

Decision Surface of a Perceptron



(a)



(b)

Represents some useful functions

- What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?

But some functions not representable

- All not linearly separable
- Therefore, we'll want networks of these...

Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

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

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., 0.1) called *learning rate*

Perceptron Training Rule

Can prove it will converge if

- Training data is linearly separable
- η sufficiently small

Gradient Descent

To understand, consider simpler *linear unit*, where

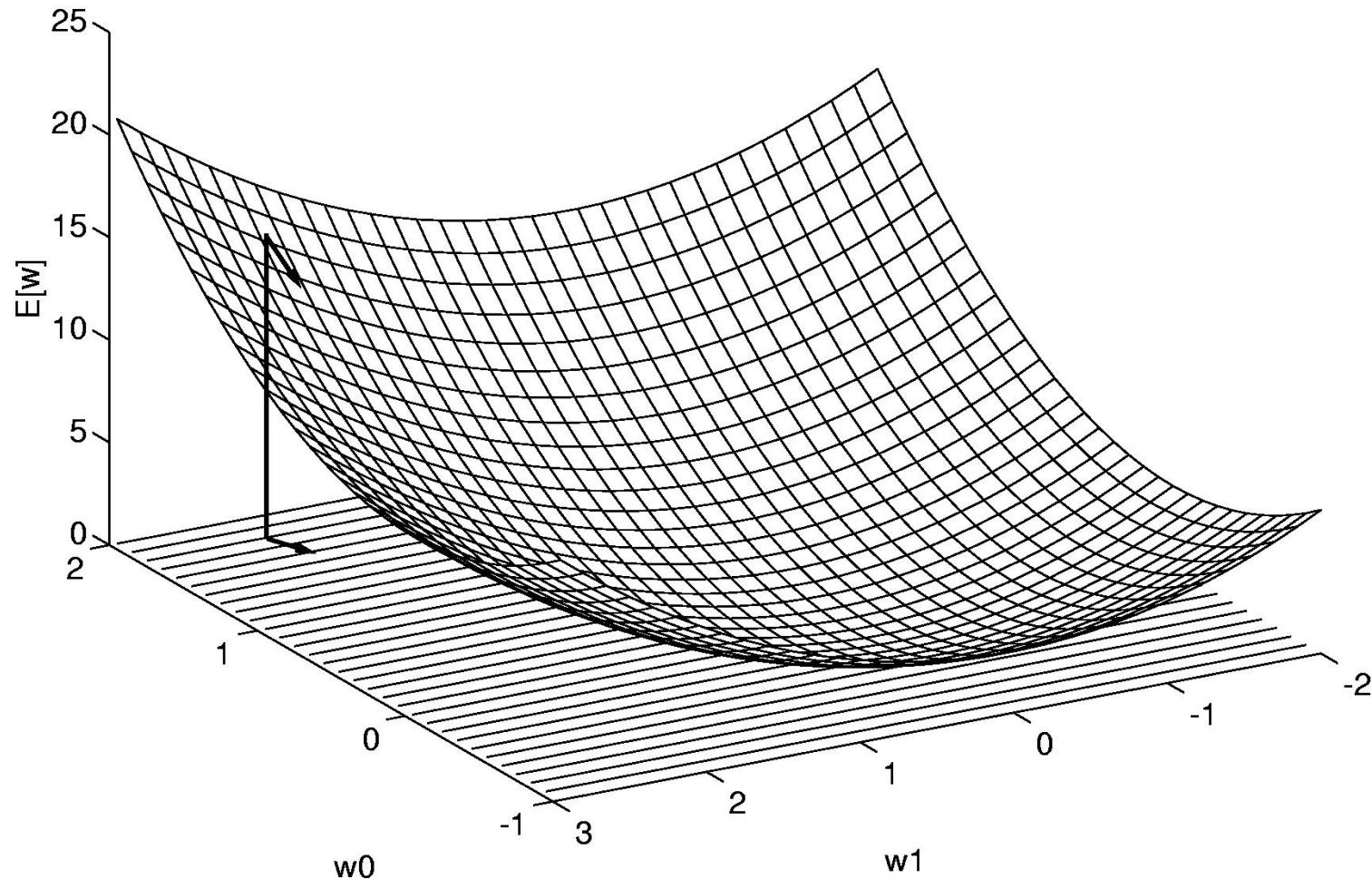
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's learn w_i 's that minimize the squared error

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

Gradient Descent



Gradient:

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

I.e.:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\ \frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d) (-x_{i,d})\end{aligned}$$

Gradient Descent

GRADIENT-DESCENT(*training-examples*, η)

Initialize each w_i to some small random value

Until the termination condition is met, Do

- Initialize each Δw_i to zero.
- For each $\langle \vec{x}, t \rangle$ in *training-examples*, Do
 - Input instance \vec{x} to unit and compute output o
 - For each linear unit weight w_i , Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

- For each linear unit weight w_i , Do

$$w_i \leftarrow w_i + \Delta w_i$$

Summary

Perceptron training rule guaranteed to succeed if

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

Linear unit training rule 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

Batch vs. Incremental Gradient Descent

Batch Mode Gradient Descent:

Do until convergence

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental Mode Gradient Descent:

Do until convergence

For each training example d in D

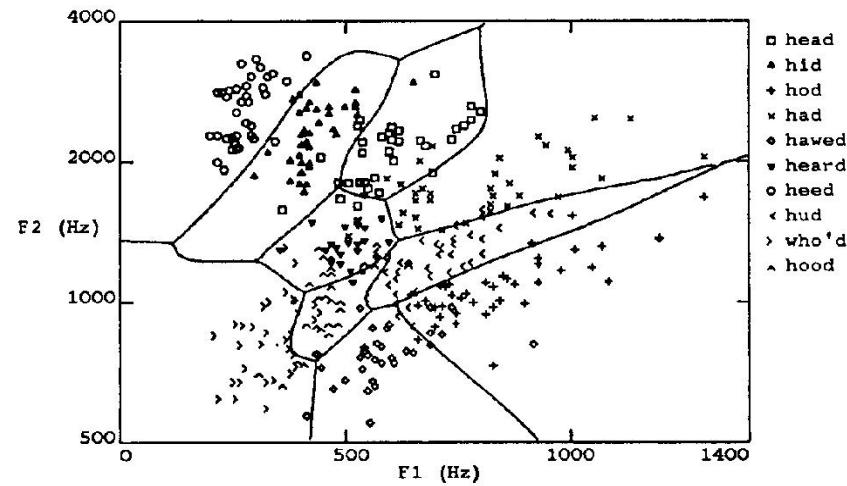
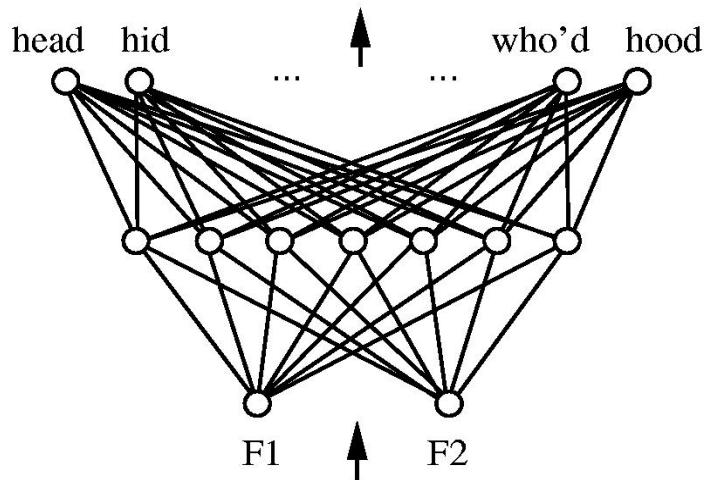
1. Compute the gradient $\nabla E_d[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

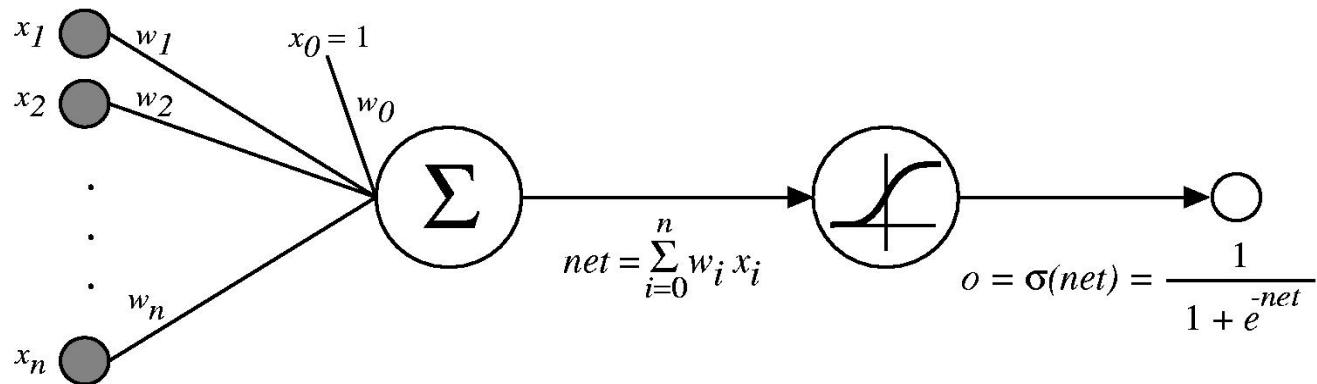
$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

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

Multilayer Networks of Sigmoid Units



Sigmoid Unit



$\sigma(x)$ is the sigmoid function

$$\frac{1}{1 + e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

Error Gradient for a Sigmoid Unit

last layer

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_d (t_d - o_d) \left(-\frac{\partial o_d}{\partial w_i} \right) \\ &= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}\end{aligned}$$

But we know:

$$\frac{\partial o_d}{\partial \text{net}_d} = \frac{\partial \sigma(\text{net}_d)}{\partial \text{net}_d} = o_d(1 - o_d)$$

$$\frac{\partial \text{net}_d}{\partial w_i} = \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

the gradient for the last layer

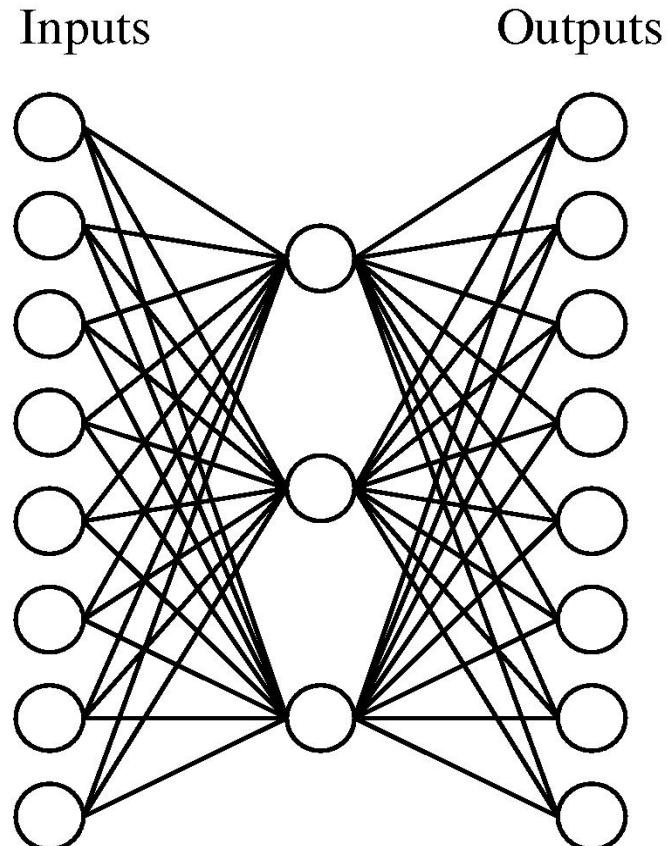
$$\frac{\partial E}{\partial w_i} = - \sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

Let: $\delta_k = -\frac{\partial E}{\partial \text{net}_k}$

all layers before the last layer

$$\begin{aligned}
\frac{\partial E}{\partial net_j} &= \sum_{k \in Outs(j)} \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k w_{kj} \frac{\partial o_k}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k w_{kj} o_j (1 - o_j) \\
\delta_j &= -\frac{\partial E}{\partial net_j} = o_j (1 - o_j) \sum_{k \in Outs(j)} \delta_k w_{kj}
\end{aligned}$$

Learning Hidden Layer Representations



A target function:

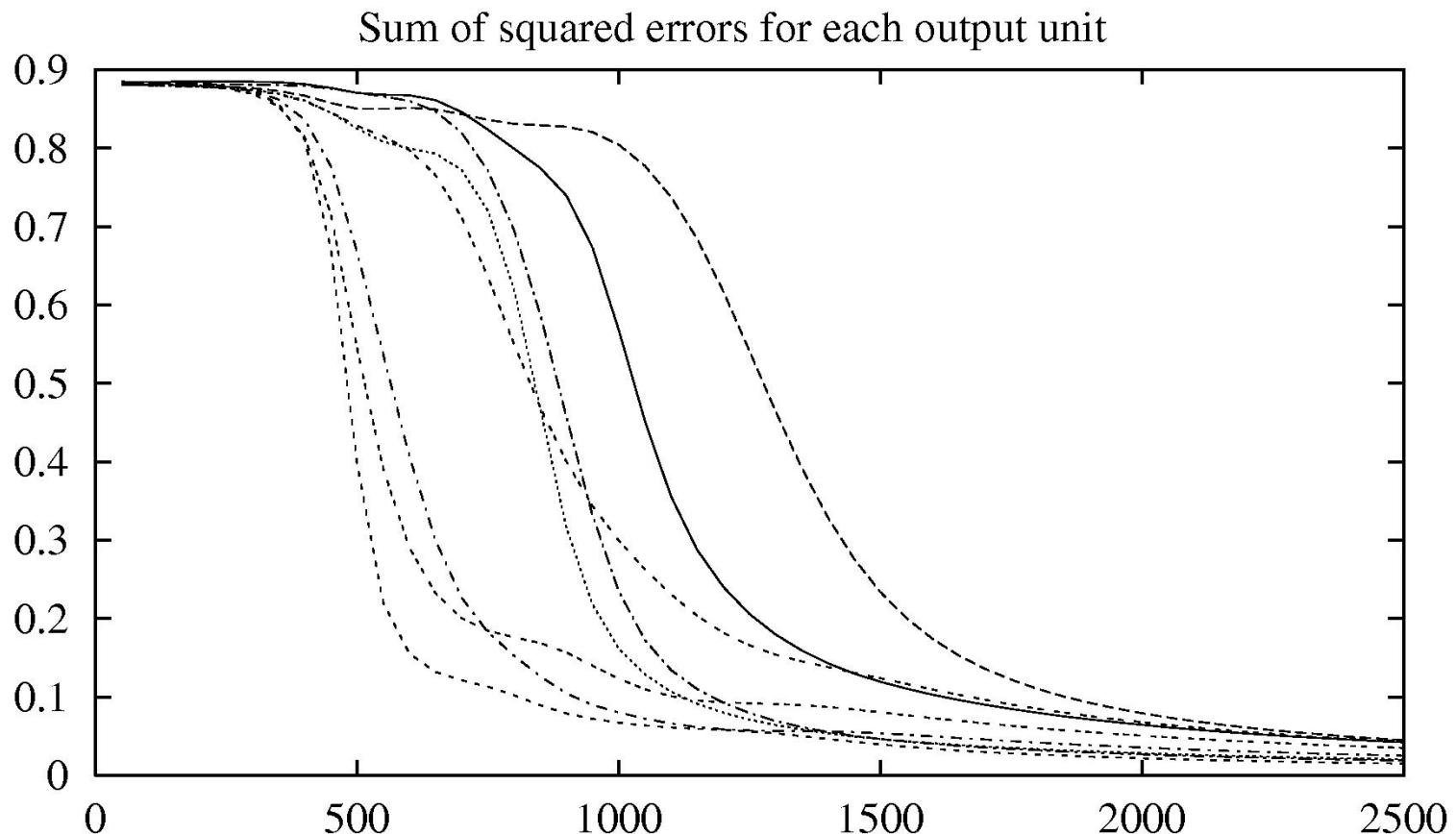
Input	Output
10000000	→ 10000000
01000000	→ 01000000
00100000	→ 00100000
00010000	→ 00010000
00001000	→ 00001000
00000100	→ 00000100
00000010	→ 00000010
00000001	→ 00000001

Can this be learned?

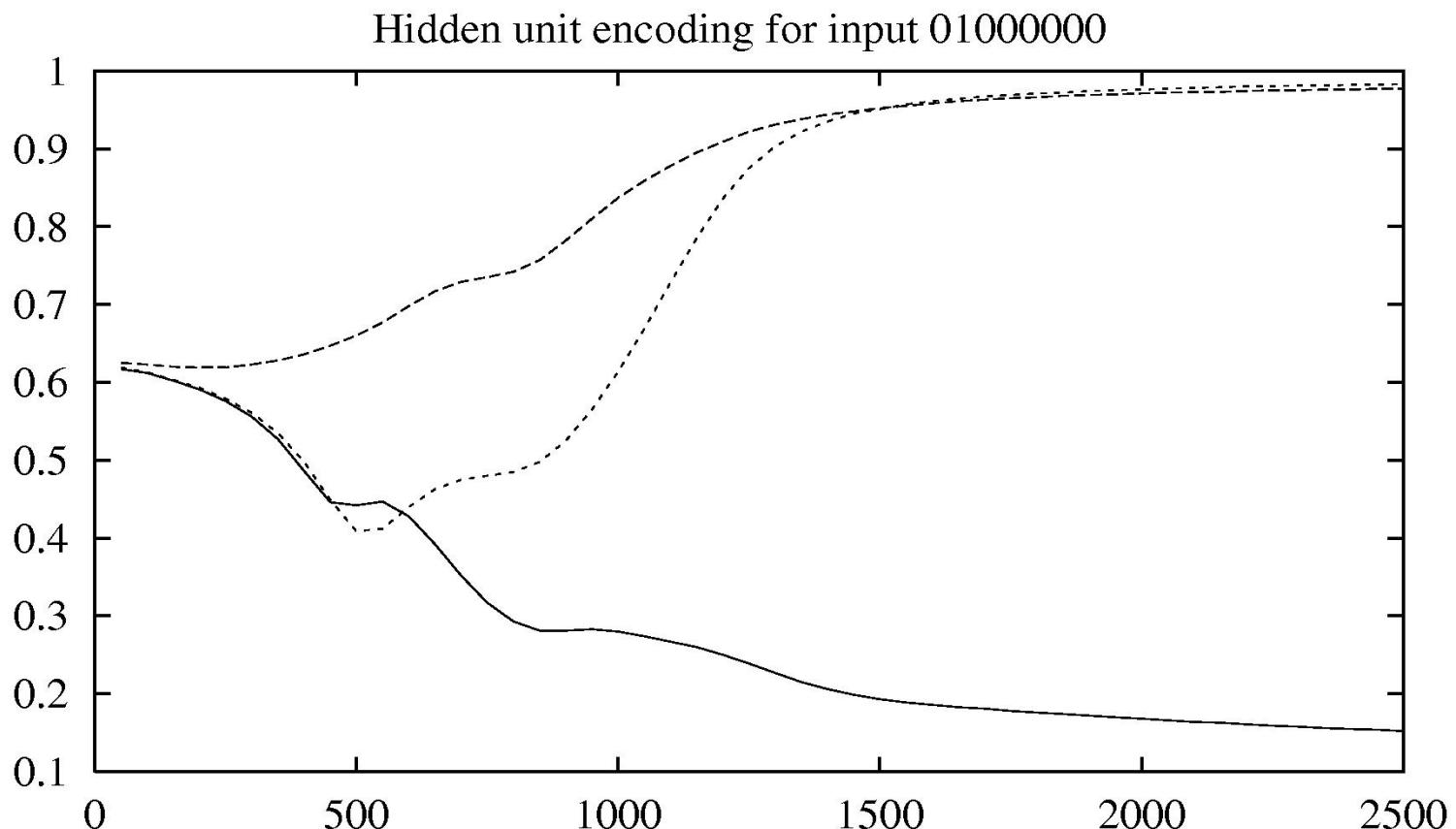
Learned hidden layer representation:

Input	Hidden Values			Output	
	.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

Training

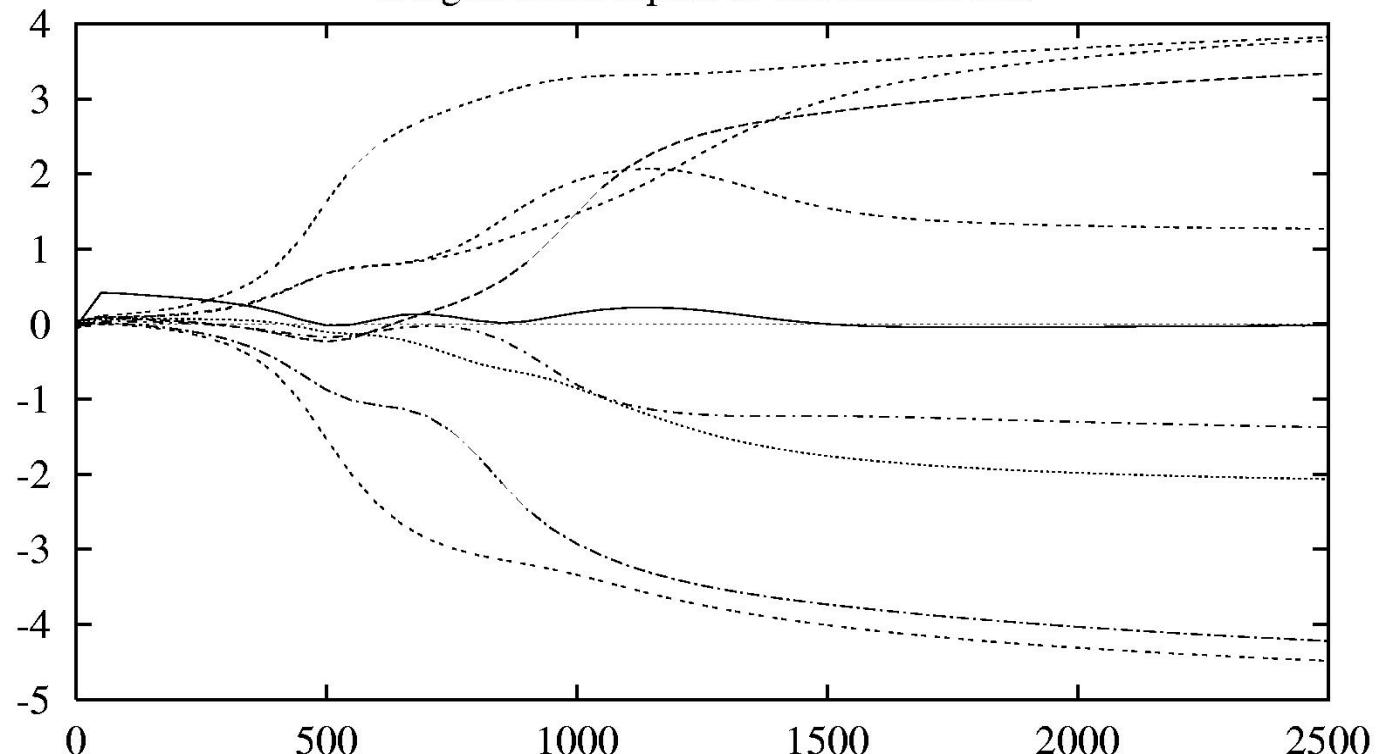


Training



Training

Weights from inputs to one hidden unit



Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- 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

Expressiveness of Neural Nets

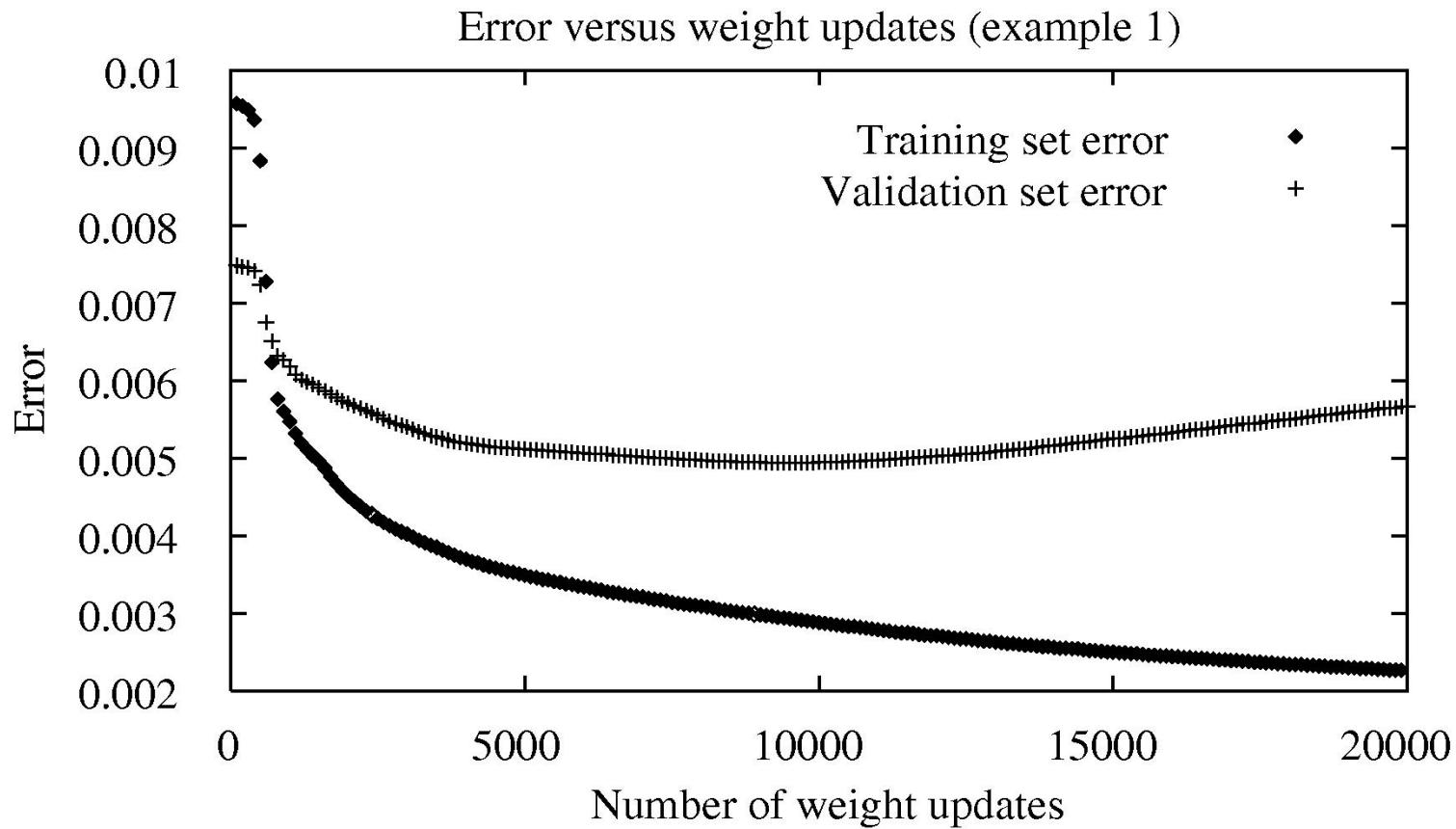
Boolean functions:

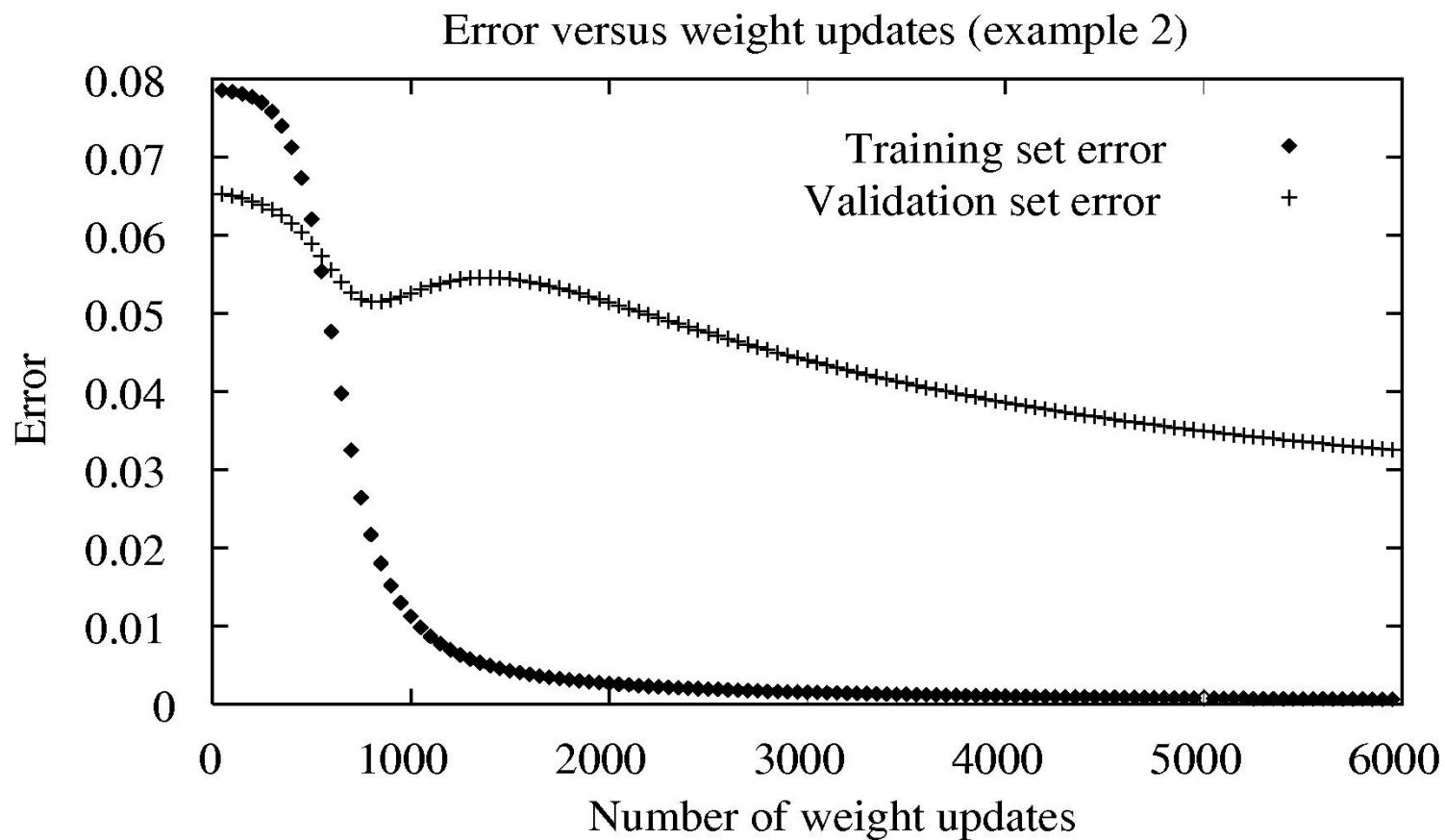
- Every Boolean function can be represented by network with single hidden layer
- But might require exponential (in number of inputs) hidden units

Continuous functions:

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

Overfitting in Neural Nets





Neural Networks: Summary

- Perceptrons
- Gradient descent
- Multilayer networks
- Backpropagation