

# Artificial Neural Networks

Based on “Machine Learning”, T. Mitchell, McGRAW Hill, 1997, ch. 4

Acknowledgement:

The present slides are an adaptation of slides drawn by T. Mitchell

## Connectionist Models

### Consider **humans**:

- Neuron switching time: .001 sec.
  - Number of neurons:  $10^{10}$
  - Connections per neuron:  $10^{4-5}$
  - Scene recognition time: 0.1 sec.
  - 100 inference steps doesn't seem like enough
- much parallel computation

### Properties of **artificial neural nets**

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

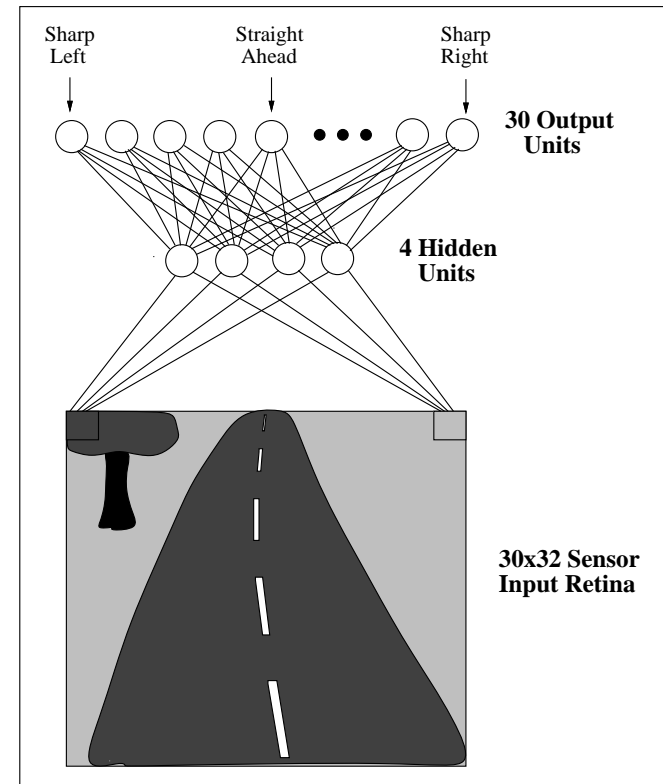
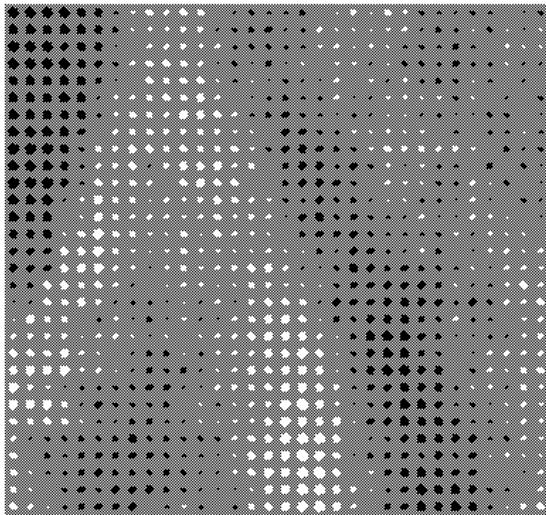
## When to Consider Neural Networks

- 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
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

## PLAN

1. Examples:  
ALVINN driving system; Face recognition
2. The perceptron  
The linear unit  
The gradient descent learning rule
3. Multilayer networks of sigmoid units  
The Backpropagation algorithm  
Hidden layer representations
4. Advanced topics

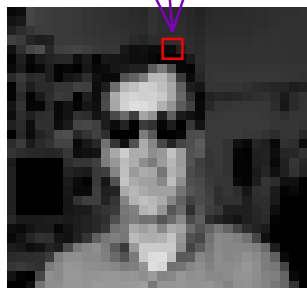
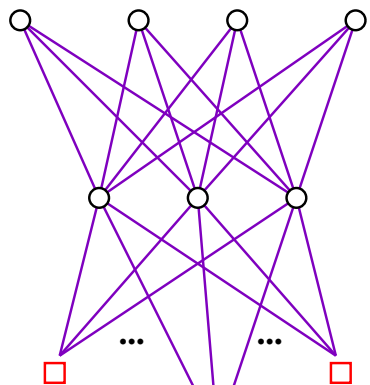
# 1. NN Examples: ALVINN drives at 70 mph on highways



# Neural Nets for Face Recognition

5.

left strt right up



30x32  
inputs

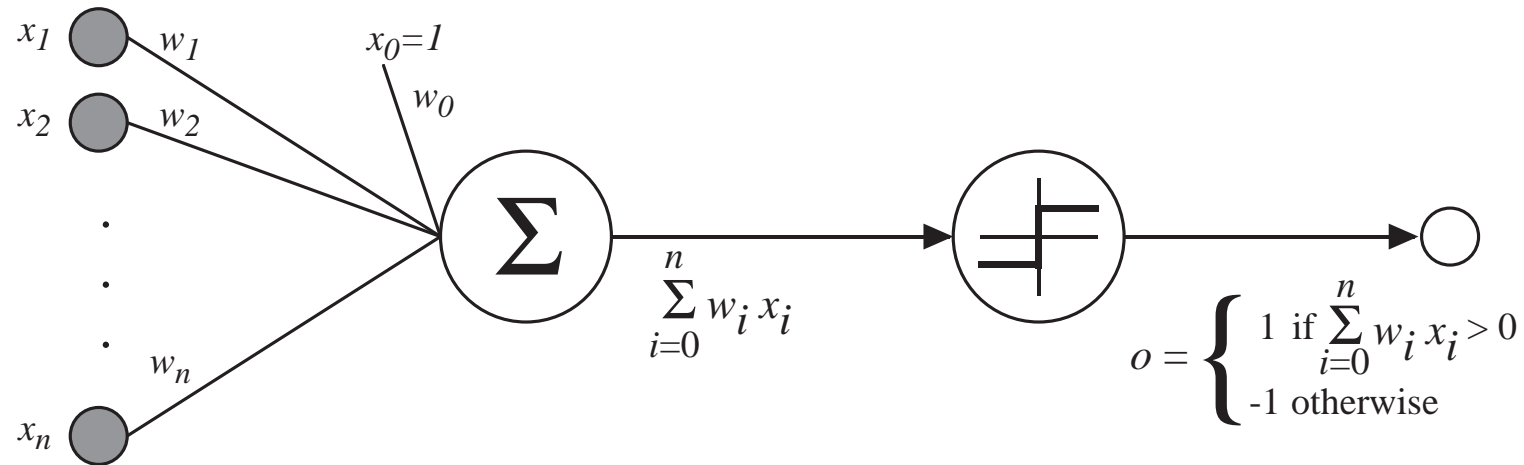


Typical input images

90% accurate learning head pose, and recognizing 1-of-20 faces

## 2. The Perceptron

[Rosenblat, 1962]

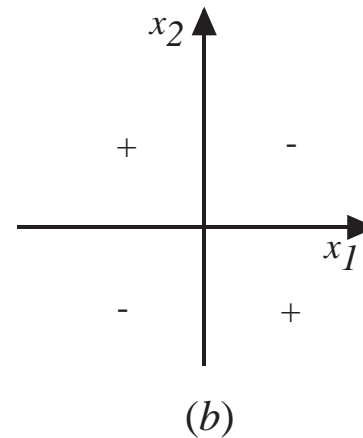
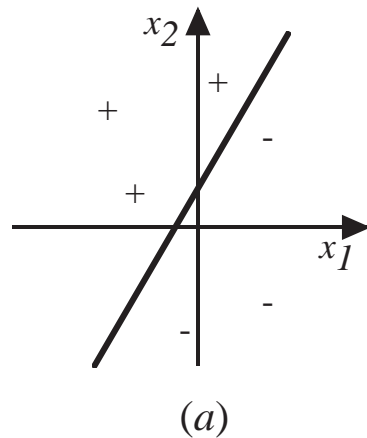


$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

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



Represents some useful functions

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

But certain examples may not be linearly separable

- Therefore, we'll want networks of these...

## The Perceptron Training Rule

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

with

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

Where:

- $t = c(\vec{x})$  is target value
- $o$  is perceptron output
- $\eta$  is small positive constant (e.g., .1) called *learning rate*

It will **converge**

- if the training data is linearly separable
- and  $\eta$  is sufficiently small.

## 2'. The Linear Unit

To understand the perceptron's training rule, consider a simpler *linear unit*, where

$$o = w_0 + w_1x_1 + \cdots + w_nx_n$$

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

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

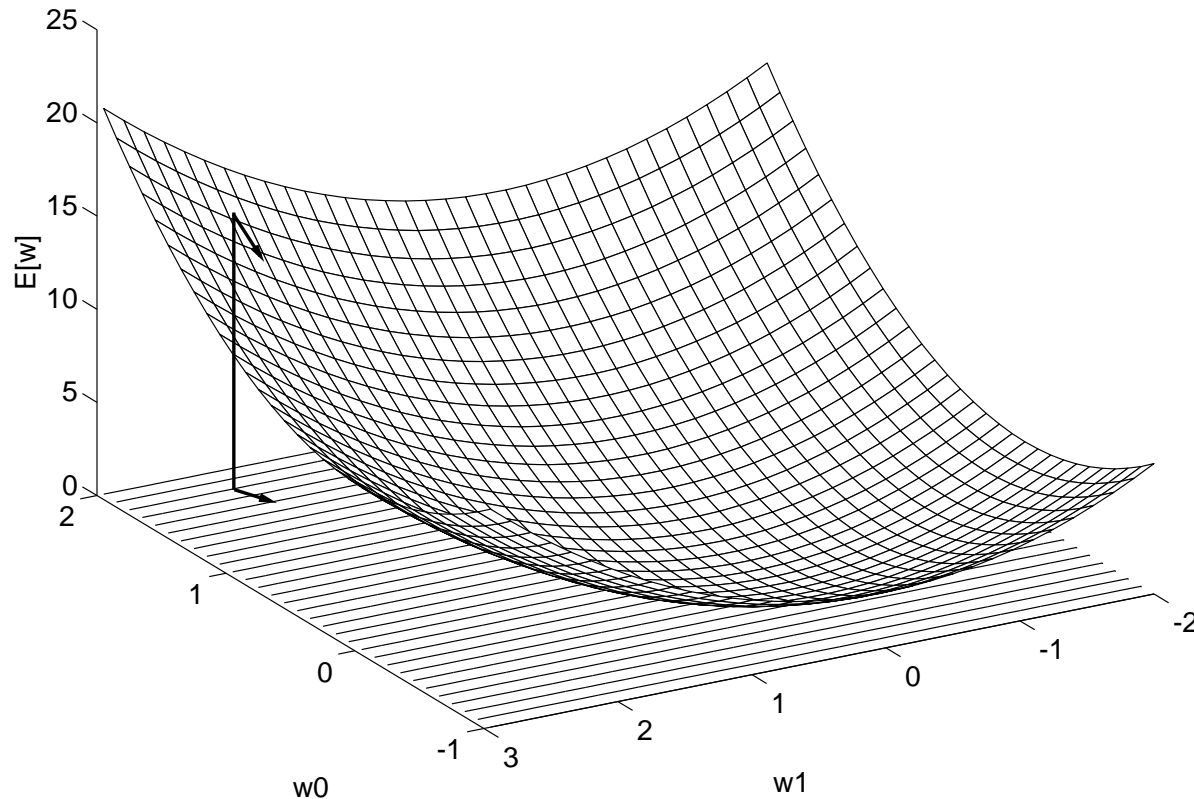
where  $D$  is set of training examples.

The linear unit uses the descent gradient training rule, presented on the next slides.

Remark:

Ch. 6 (Bayesian Learning) shows that the hypothesis  $h = (w_0, w_1, \dots, w_n)$  that minimises  $E$  is the most probable hypothesis given the training data.

# The Gradient Descent Rule



**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: Computation

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

Therefore

$$\Delta w_i = -\eta \sum_d (t_d - o_d) x_{i,d}$$

# The Gradient Descent Algorithm

GRADIENT-DESCENT(*training\_examples*,  $\eta$ )

*Each training example is a pair of the form  $\langle \vec{x}, t \rangle$ , where*

*$\vec{x}$  is the vector of input values*

*$t$  is the target output value.*

*$\eta$  is the learning rate (e.g., .05).*

- Initialize each  $w_i$  to some small random value
- Until the termination condition is met
  - Initialize each  $\Delta w_i$  to zero.
  - For each  $\langle \vec{x}, t \rangle$  in *training\_examples*
    - \* Input the instance  $\vec{x}$  to the unit and compute the output  $o$
    - \* For each linear unit weight  $w_i$

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

- For each linear unit weight  $w_i$

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

The **gradient descent** training rule used by the **linear unit** is guaranteed to **converge** to a hypothesis with minimum squared error

- given a sufficiently small learning rate  $\eta$
- even when the training data contains noise
- even when the training data is not separable by  $H$

**Note:** If  $\eta$  is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification of the algorithm is to gradually reduce the value of  $\eta$  as the number of gradient descent steps grows.

## Remark

**Gradient descent** is an **important general paradigm for learning**. It is a strategy for searching through a large or infinite hypothesis space that **can be applied whenever**

- the hypothesis space contains continuously parametrized hypotheses
- the error can be differentiated w.r.t. these hypothesis parameters.

**Practical difficulties** in applying the gradient descent method:

- if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.
- converging to a local minimum can sometimes be quite slow.

To alleviate these difficulties, a variantion called **incremental (or: stochastic) gradient descent** was proposed (see next slide).

## Incremental (Stochastic) Gradient Descent

**Batch mode** Gradient Descent:

Do until satisfied

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 satisfied

- For each training example  $d$  in  $D$ 
  1. Compute the gradient  $\nabla E_d[\vec{w}]$
  2.  $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

---

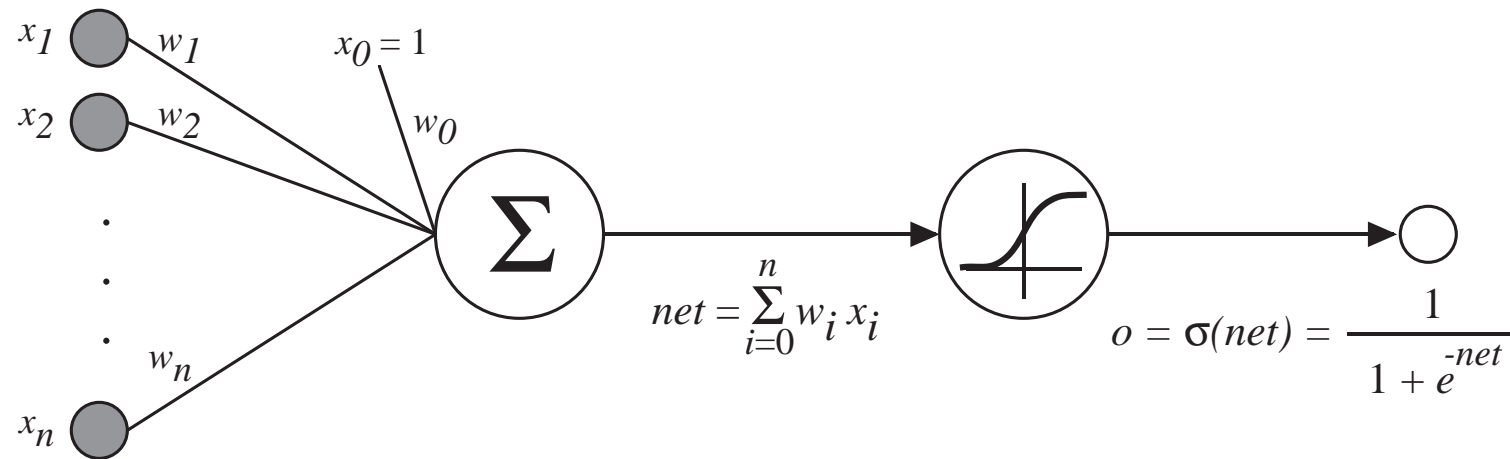

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

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


---

The *Incremental Gradient Descent* can approximate the *Batch Gradient Descent* arbitrarily closely if  $\eta$  is made small enough.

## 2''. The 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  $\rightarrow$  **Backpropagation**

## Error Gradient for a Sigmoid Unit

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

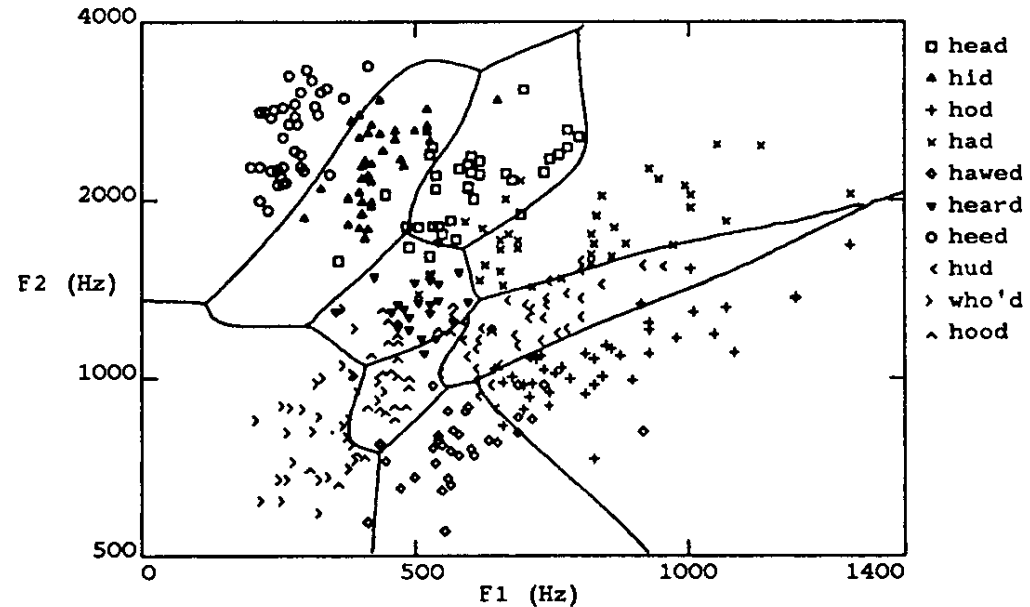
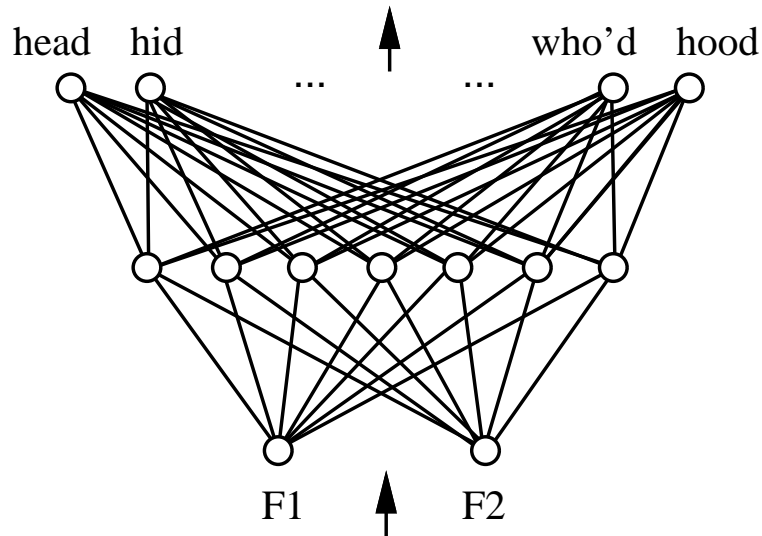
But

$$\begin{aligned}
 \frac{\partial o_d}{\partial net_d} &= \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d) \\
 \frac{\partial net_d}{\partial w_i} &= \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}
 \end{aligned}$$

So:

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

### 3. Multilayer Networks of Sigmoid Units



This network was trained to recognize 1 of 10 vowel sounds occurring in the context “h\_d” (e.g. “head”, “hid”). The inputs have been obtained from a spectral analysis of sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is the highest.

The plot on the right illustrates the highly non-linear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network.

## 3.1 The Backpropagation Algorithm

Initialize all weights to small random numbers.

Until satisfied,

for each training example,

1. input the training example to the network and compute the network outputs

2. for each output unit  $k$ :  $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$

3. for each hidden unit  $h$ :  $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{hk} \delta_k$

4. update each network weight  $w_{ij}$ :  $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$

where  $\Delta w_{ij} = \eta \delta_j x_{ij}$ .

**Note:** To see how these weight-updating rules were derived, please refer to [Tom Mitchell, 1997] pag. 101–103.

## More on Backpropagation

- Gradient descent over the 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 run multiple times)
- Often include weight **momentum**  $\alpha$

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

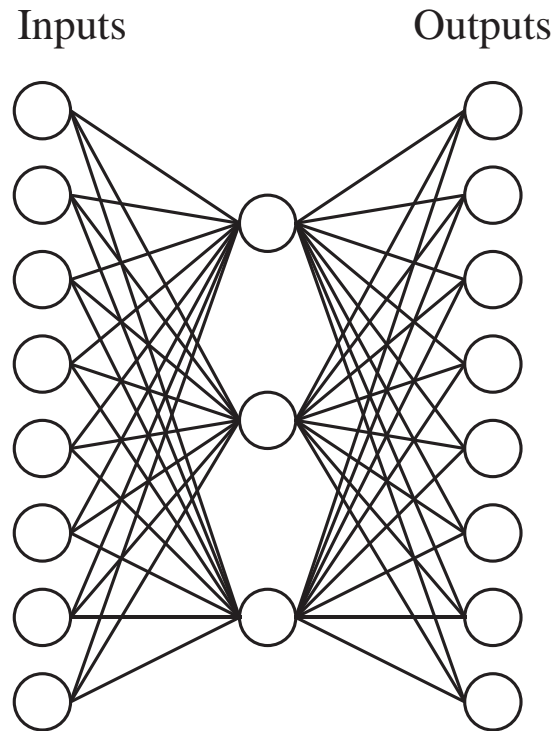
- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations → slow!
- Using network after training is very fast

## Convergence of Backpropagation

### Nature of convergence

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

## 3.2 Learning Hidden Layer Representations



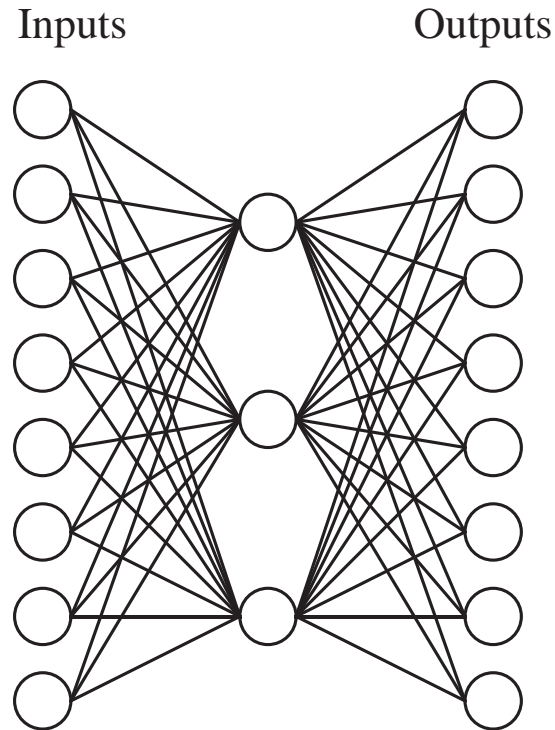
A target function:

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

This network was trained to learn the identity function, using the 8 training examples shown.

# Learning Hidden Layer Representations

A network:

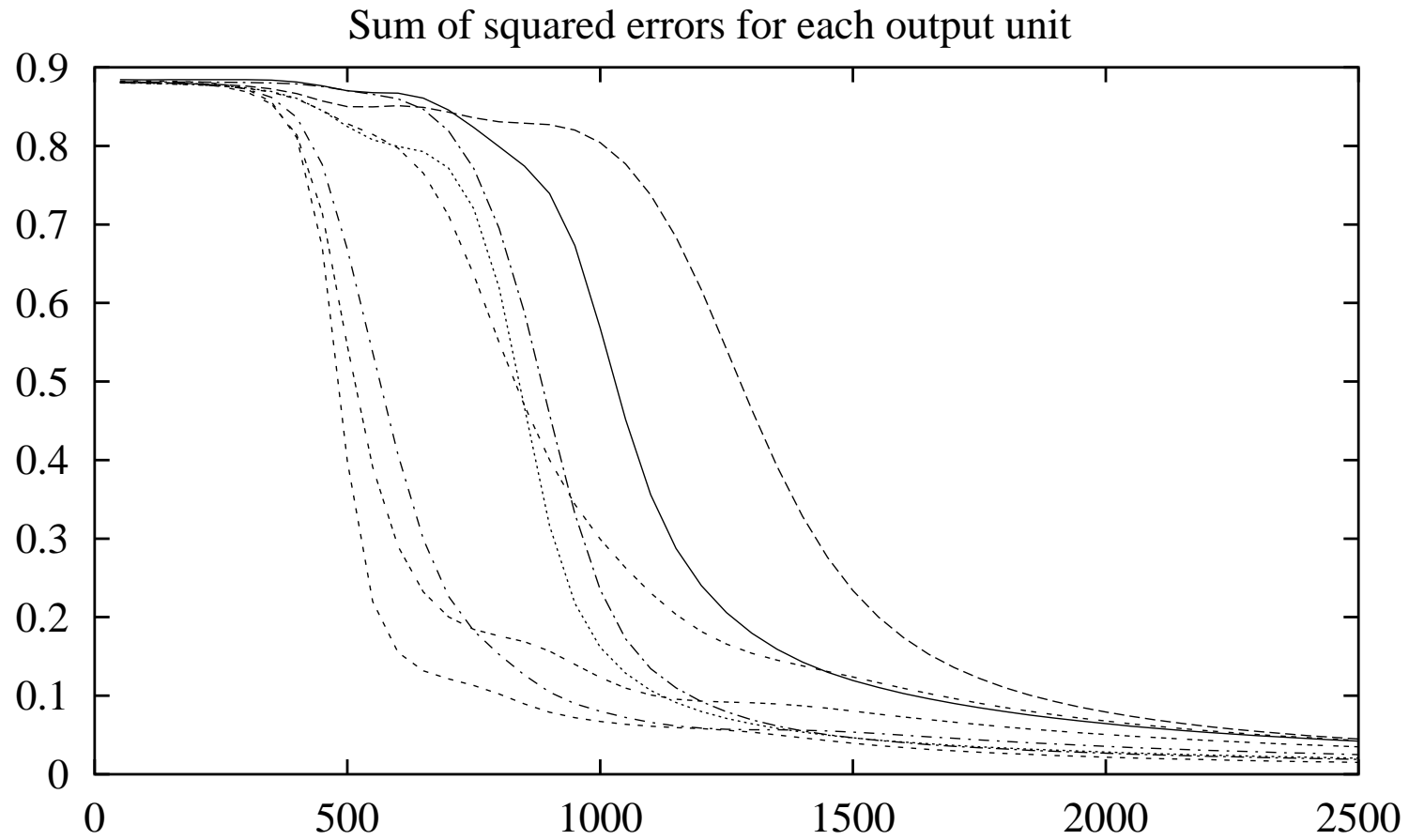


Learned hidden layer representation:

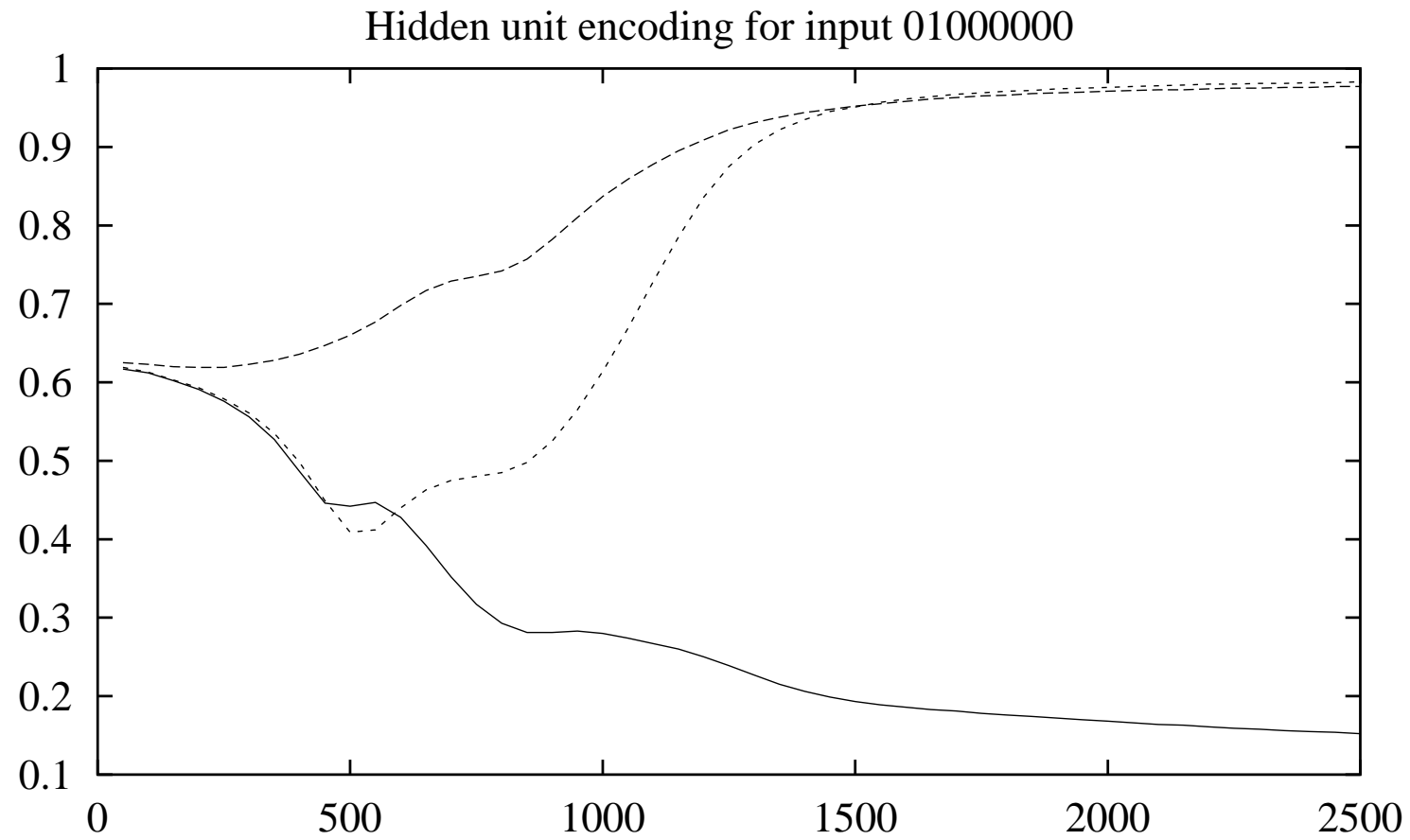
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

After 8000 training epochs, the 3 hidden unit values encode the 8 distinct inputs. Note that if the encoded values are rounded to 0 or 1, the result is the standard binary encoding for 8 distinct values (however not the usual one, i.e.  $1 \rightarrow 001$ ,  $2 \rightarrow 010$ , etc).

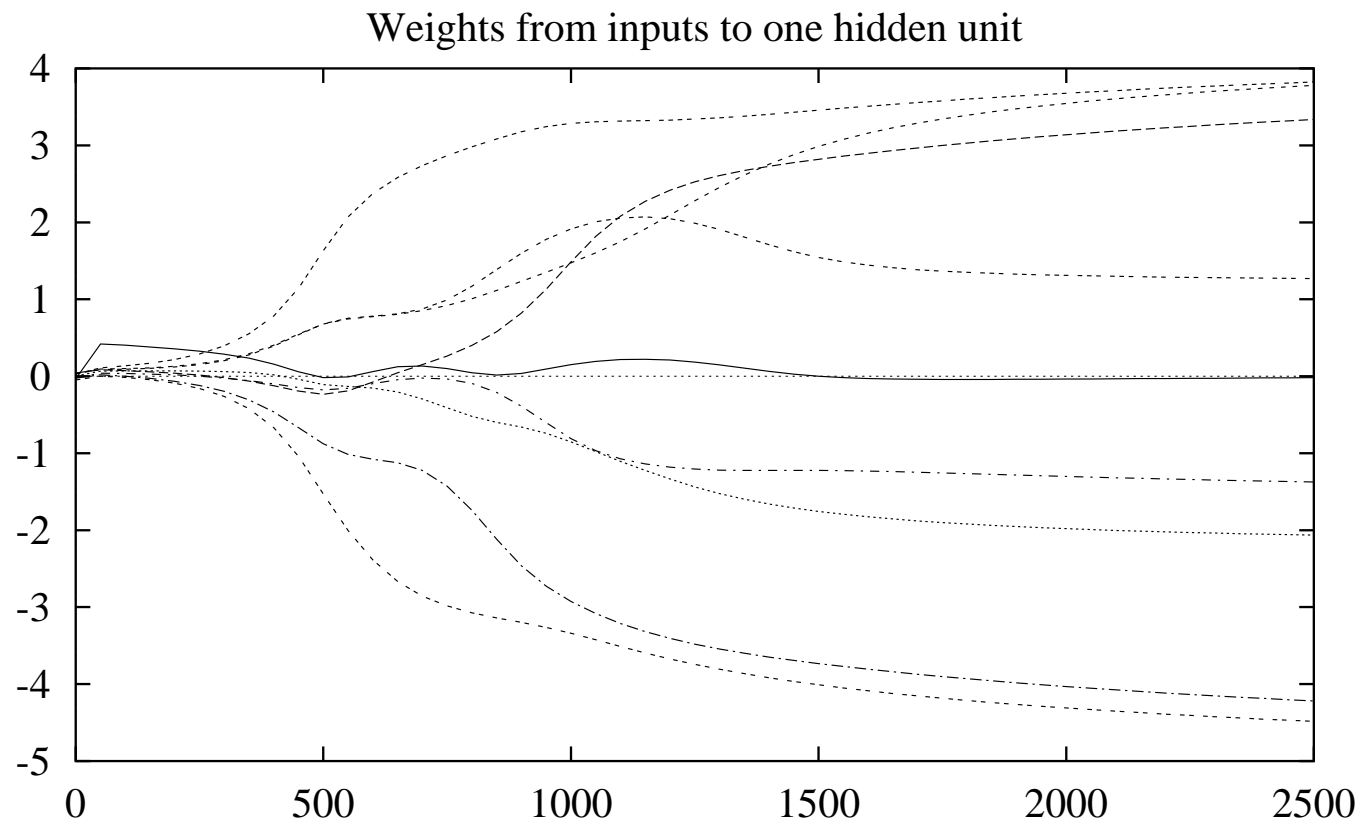
# Training (I)



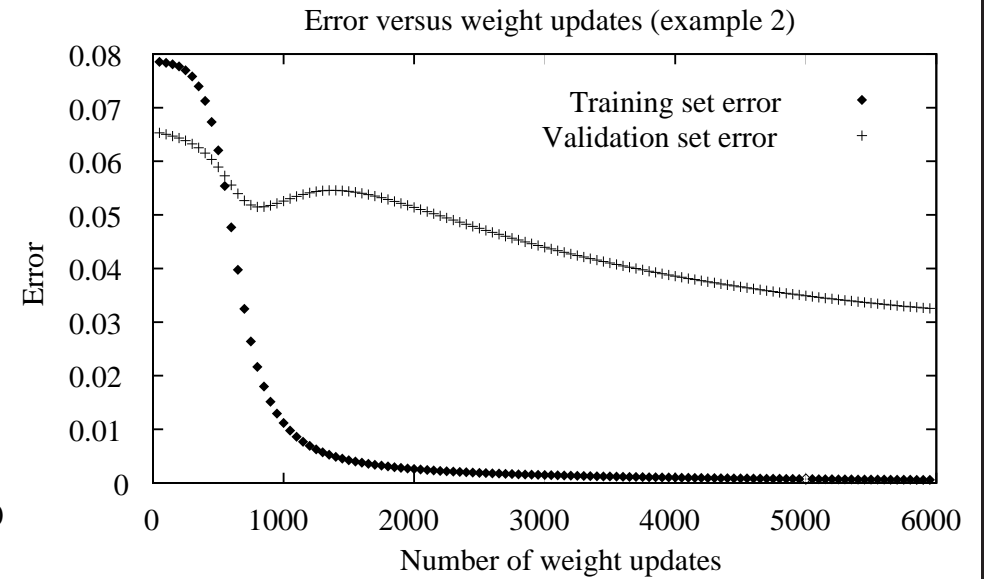
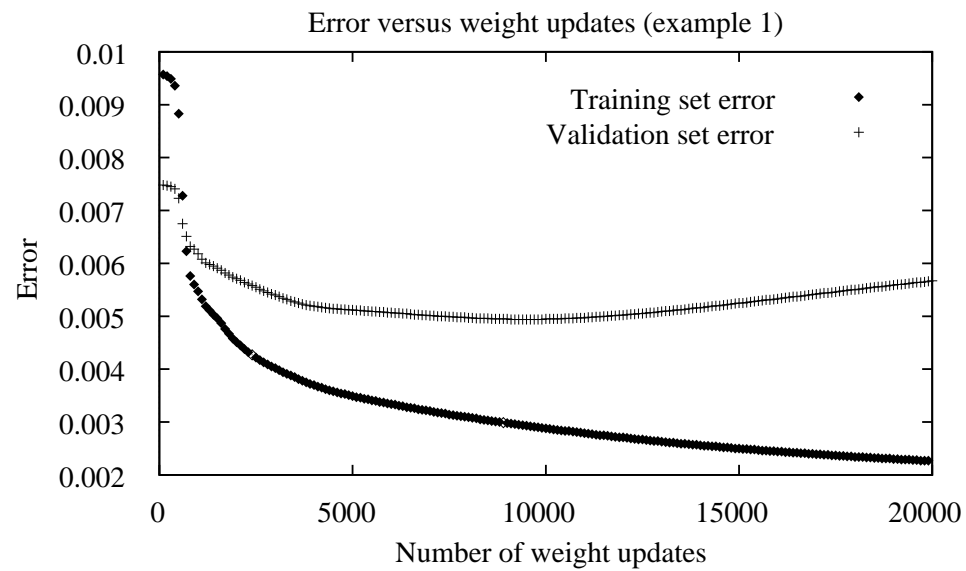
# Training (II)



## Training (III)



## 3.2 Overfitting in ANNs



## 4. Advanced Topics

### 4.1 Alternative Error Functions

Penalize large weights:

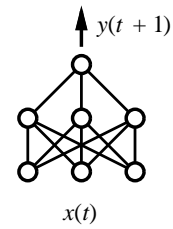
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

Train on target slopes as well as values:

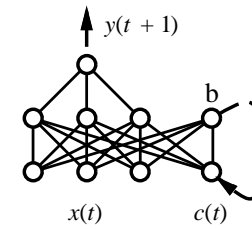
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} \left[ (t_{kd} - o_{kd})^2 + \mu \sum_{j \in \text{inputs}} \left( \frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

Tie together weights:

- e.g., in phoneme recognition network

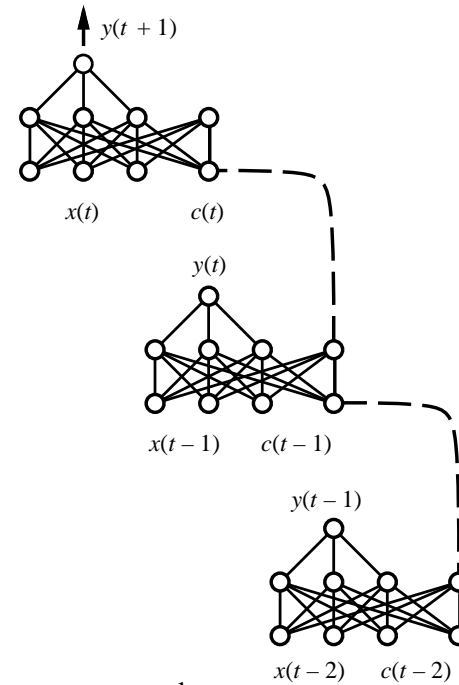


(a) Feedforward network



(b) Recurrent network

## 4.2 Recurrent Networks

(c) Recurrent network  
unfolded in time

## 4.3 Expressive Capabilities of ANNs

### Boolean functions:

- Every boolean function can be represented by a network with a single hidden layer,
- but it might require exponential (in the number of inputs) hidden units.

### Continuous functions:

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