

ARTIFICIAL NEURAL NETWORKS

[Read Ch. 4]

[Read Ch. 1 and Ch. 2 and Ch. 3 of 03-60-561]

[Recommended exercises 4.1, 4.2, 4.5, 4.9, 4.11]

- Threshold units
- Gradient descent
- Multilayer networks
- Backpropagation
- Hidden layer representations
- Example: Face Recognition
- Advanced topics

Connectionist Models

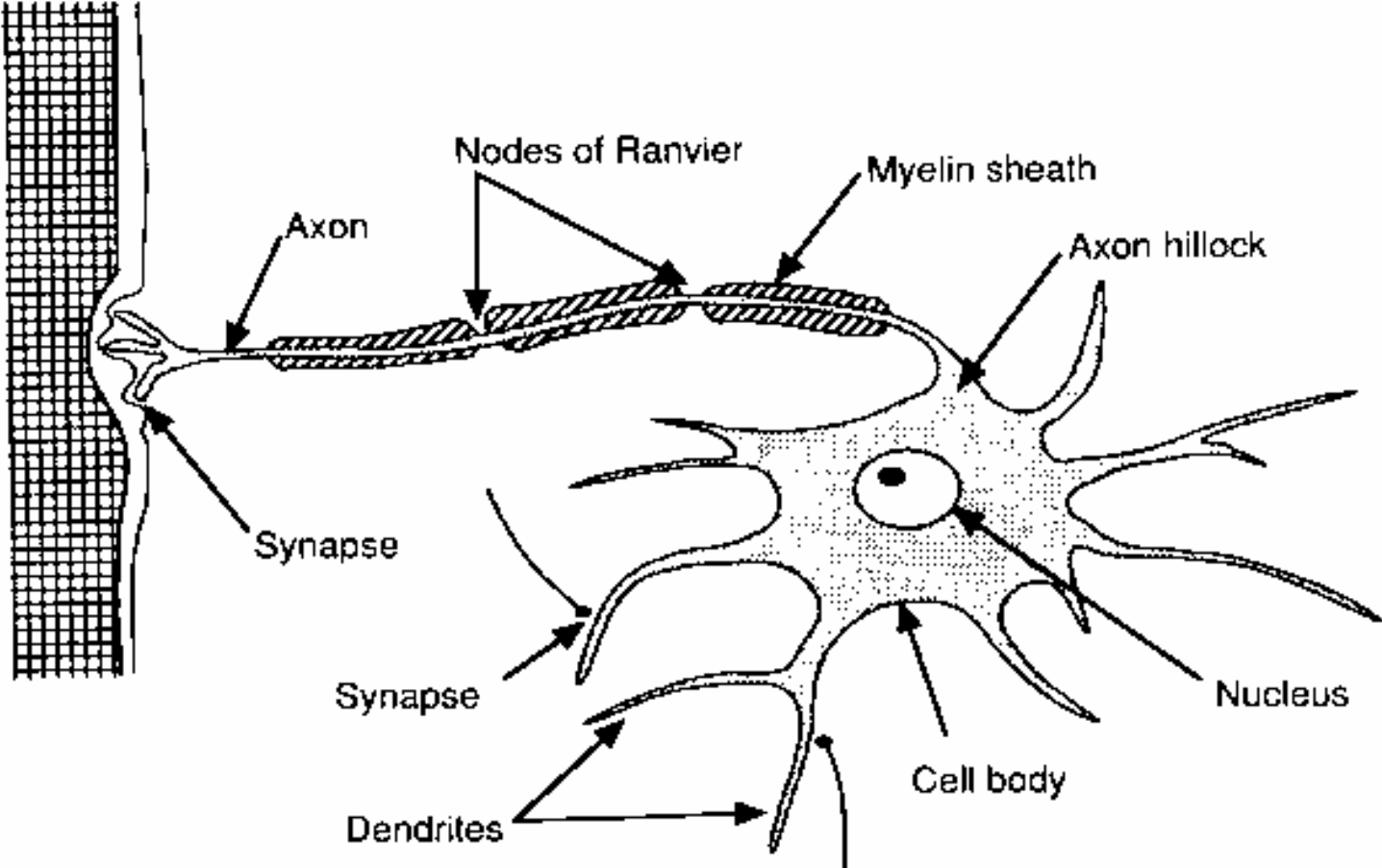
Consider humans:

- 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 artificial neural nets (ANN's):

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

Biological Neuron



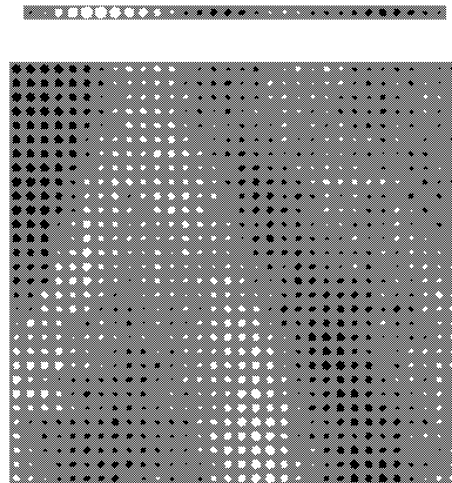
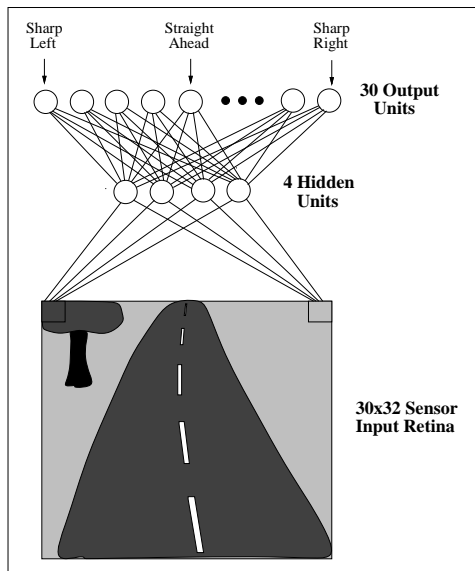
When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued
- 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

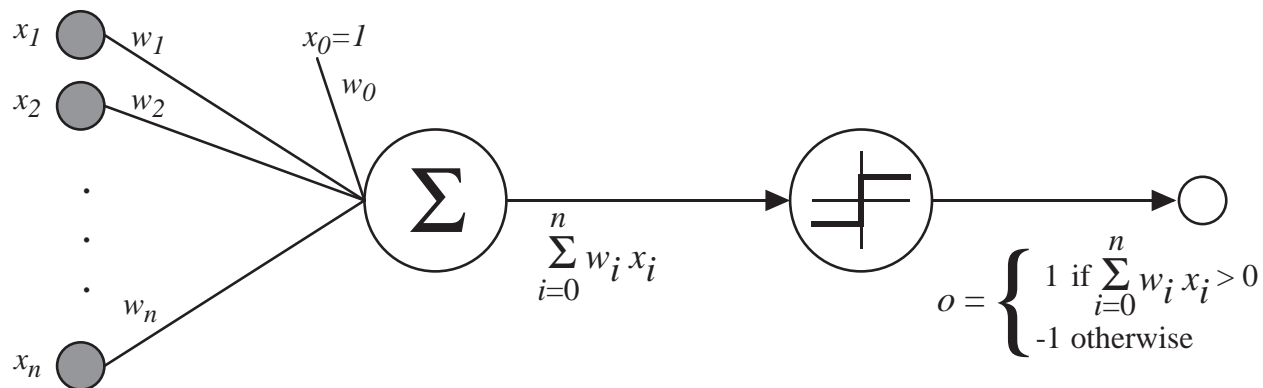
Examples:

- Speech phoneme recognition [Waibel]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction

ALVINN Drives 70 mph on Highways



The Perceptron



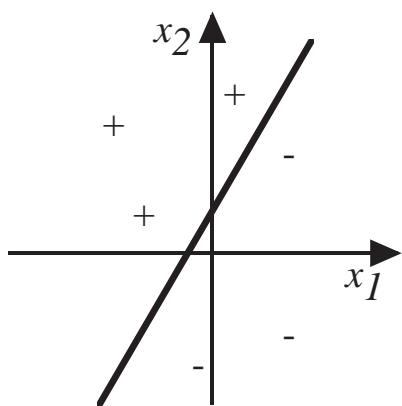
$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

- The perceptron represents some useful functions

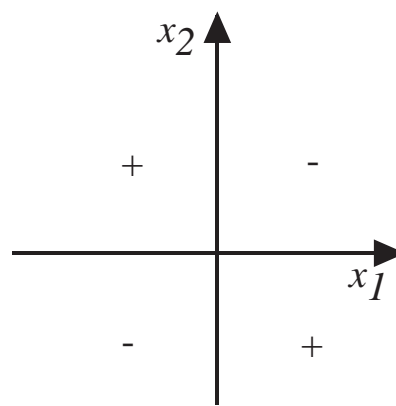
What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?

- But some functions are not representable

e.g., not linearly separable



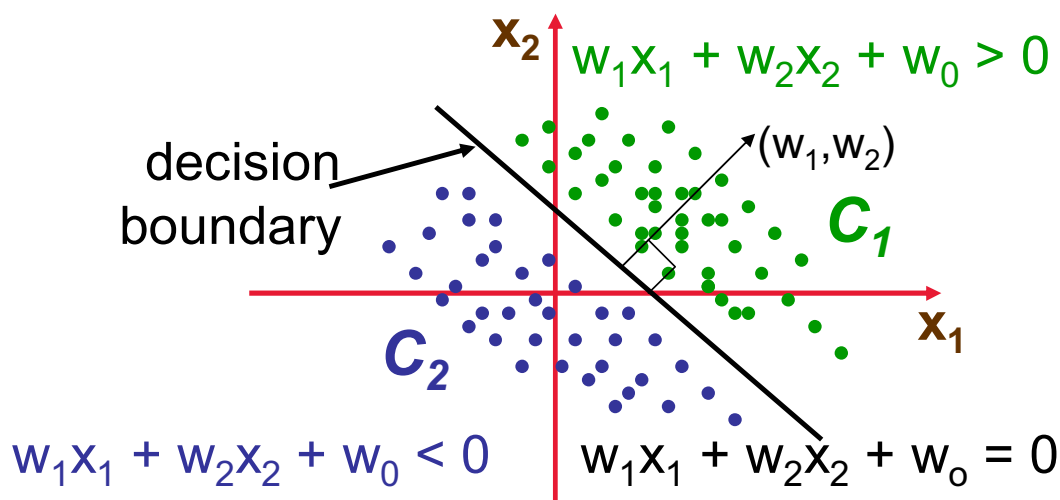
(a)



(b)

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

Connection with Geometry



$\sum_{i=1}^n w_i x_i + w_0 = 0$ describes a hyperplane which divides the instance space \mathcal{R}^n into two half-spaces

$$\chi_+ = \{X_p \in \mathcal{R}^n \mid W \bullet X_p + w_0 > 0\} \quad \text{and} \quad \chi_- = \{X_p \in \mathcal{R}^n \mid W \bullet X_p + w_0 < 0\}$$

Instance space \mathcal{R}^n

Hypothesis space is the set of $(n-1)$ -dimensional hyperplanes defined in the n -dimensional instance space

A hypothesis is defined by $\sum_{i=0}^n w_i x_i = 0$

Orientation of the hyperplane is governed by $(w_1 \dots w_n)^T$ and the perpendicular distance of the hyperplane from the origin is given by

$$\left(\frac{|w_0|}{\sqrt{(w_1^2 + w_2^2 + \dots + w_n^2)}} \right)$$

Perceptron Training Rule

- Weights are updated in opposite direction of the error

$$w_i \leftarrow w_i + \Delta w_i \text{ where } \Delta w_i = \eta(t - o)x_i$$

1. $t = c(\vec{x})$: target value
 2. o : perceptron's actual output
 3. η : small constant ($0 < \eta \leq 1$) called *learning rate*
- Perceptron learning algorithm converges when
 1. Training data is linearly separable
 2. η is sufficiently small

- **Learning by Gradient Descent**

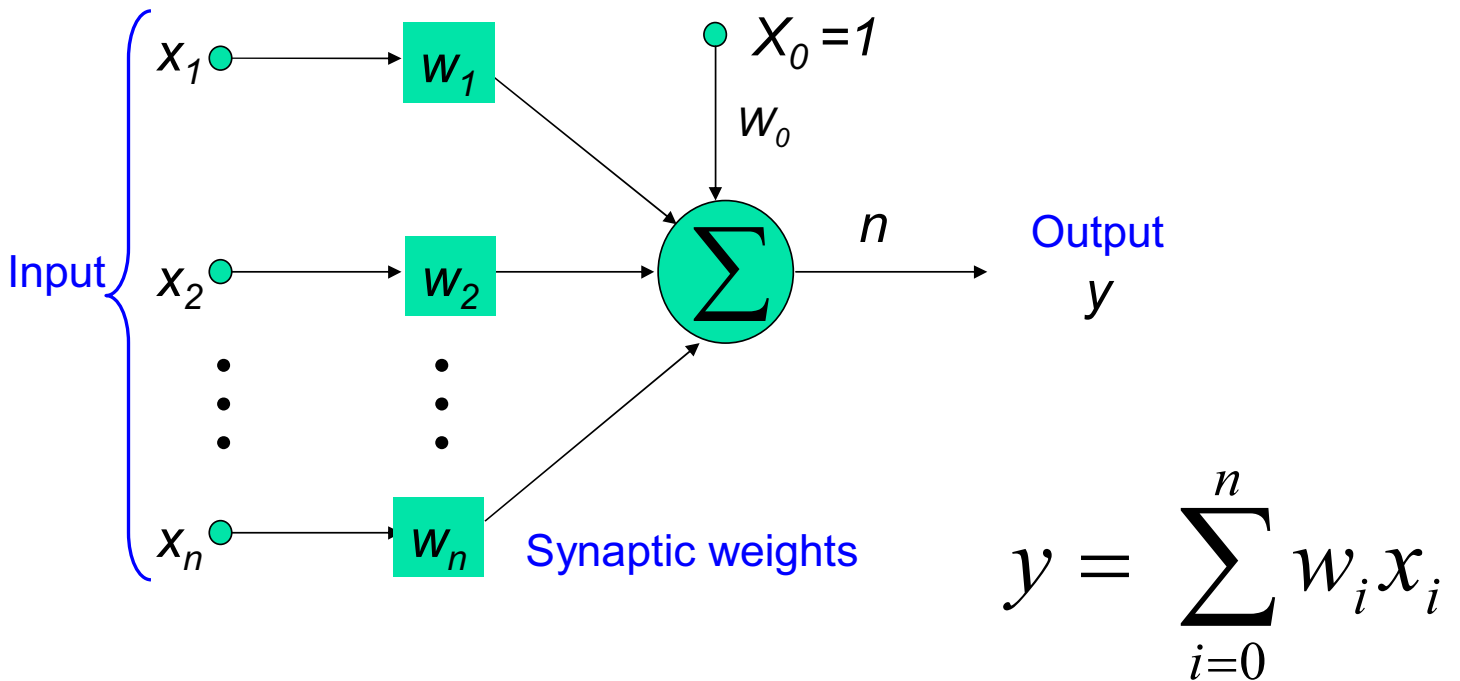
Given a *linear unit* with $o = w_0 + w_1x_1 + \dots + w_nx_n$

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

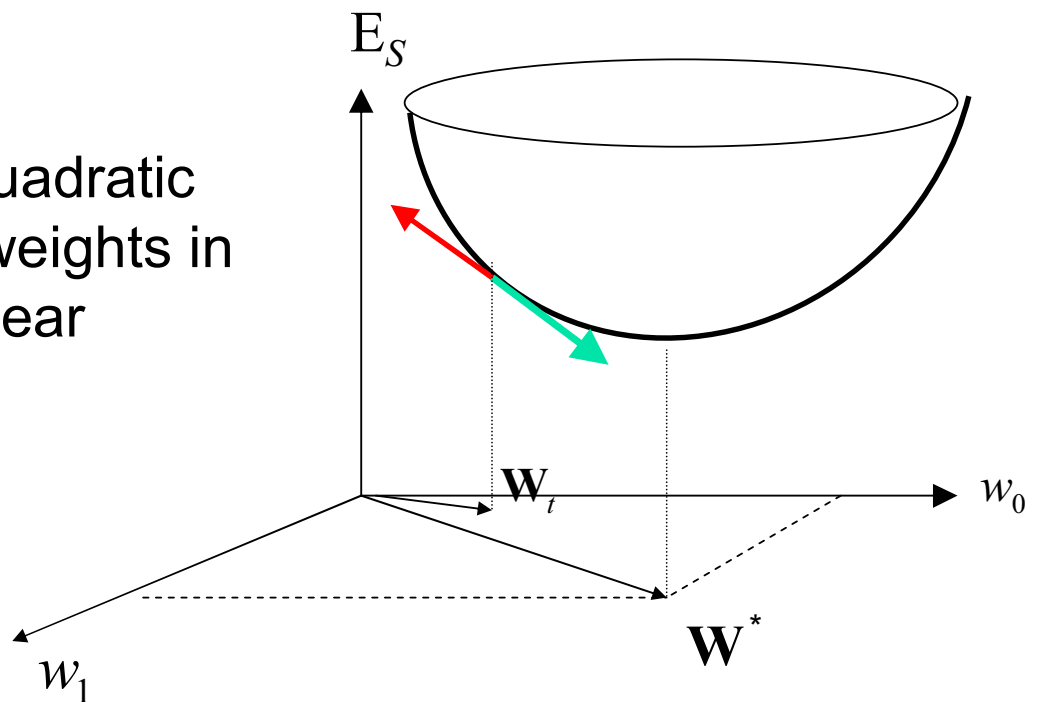
$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

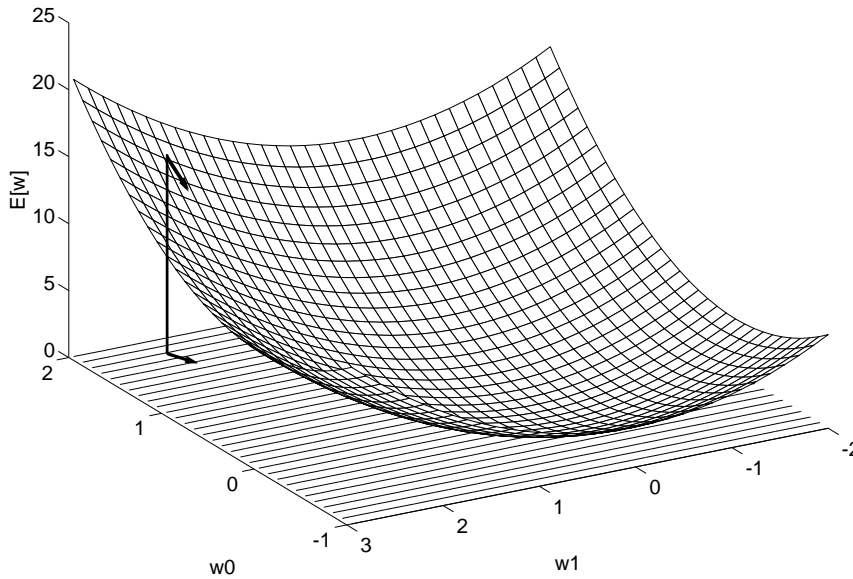
Adaline, Error Function and Gradient Descent



The error is a quadratic function of the weights in the case of a linear neuron



Gradient Descent



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

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

Adaline Algorithm

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where \vec{x} is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

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
 1. Input the instance \vec{x} to the unit and compute the output o
 2. 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
 1. Training examples are linearly separable
 2. Sufficiently small learning rate η
- Adaline training rule uses gradient descent
 1. Guaranteed to converge to hypothesis with minimum squared error
 2. Given sufficiently small learning rate η
 3. Even when training data contains noise
 4. Even when training data not separable by H
 5. Problems
 - Slow convergence to local or global minimum
 - What if there are many local minima?
 - May not find a global minimum

Incremental (Stochastic) Gradient Descent

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

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

Do until satisfied

– For each training example d in D

1. Compute the gradient $\nabla E_d[\vec{w}]$

2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

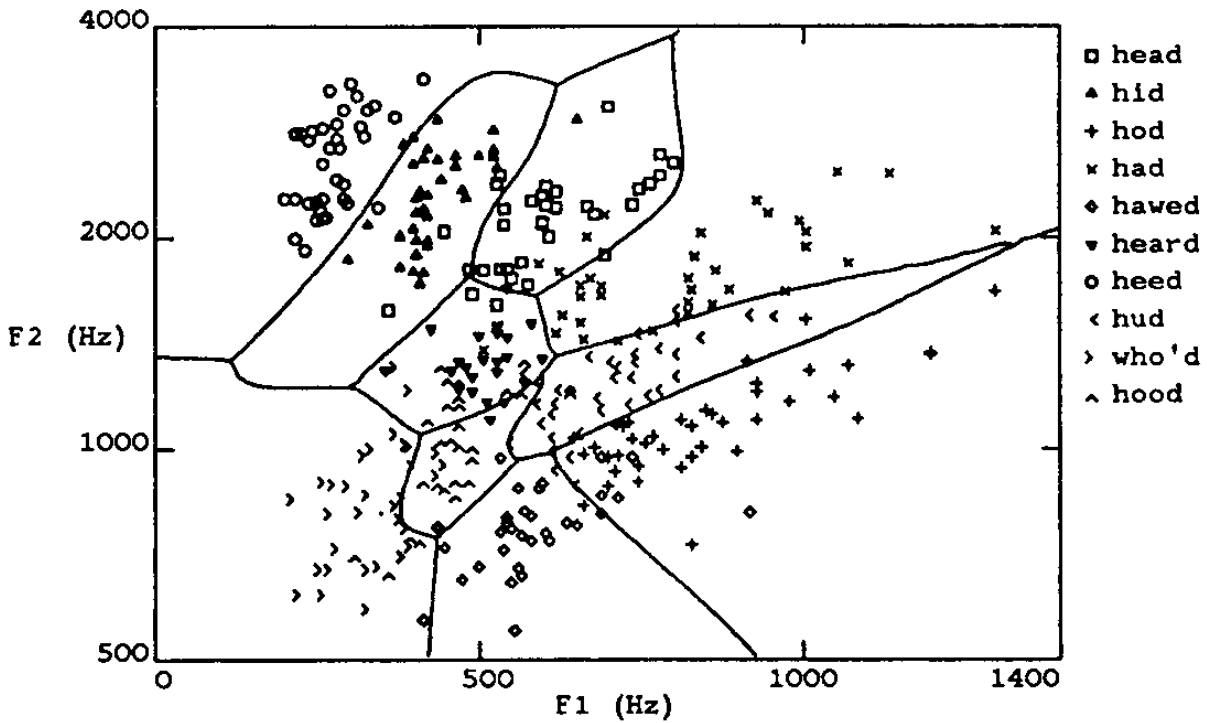
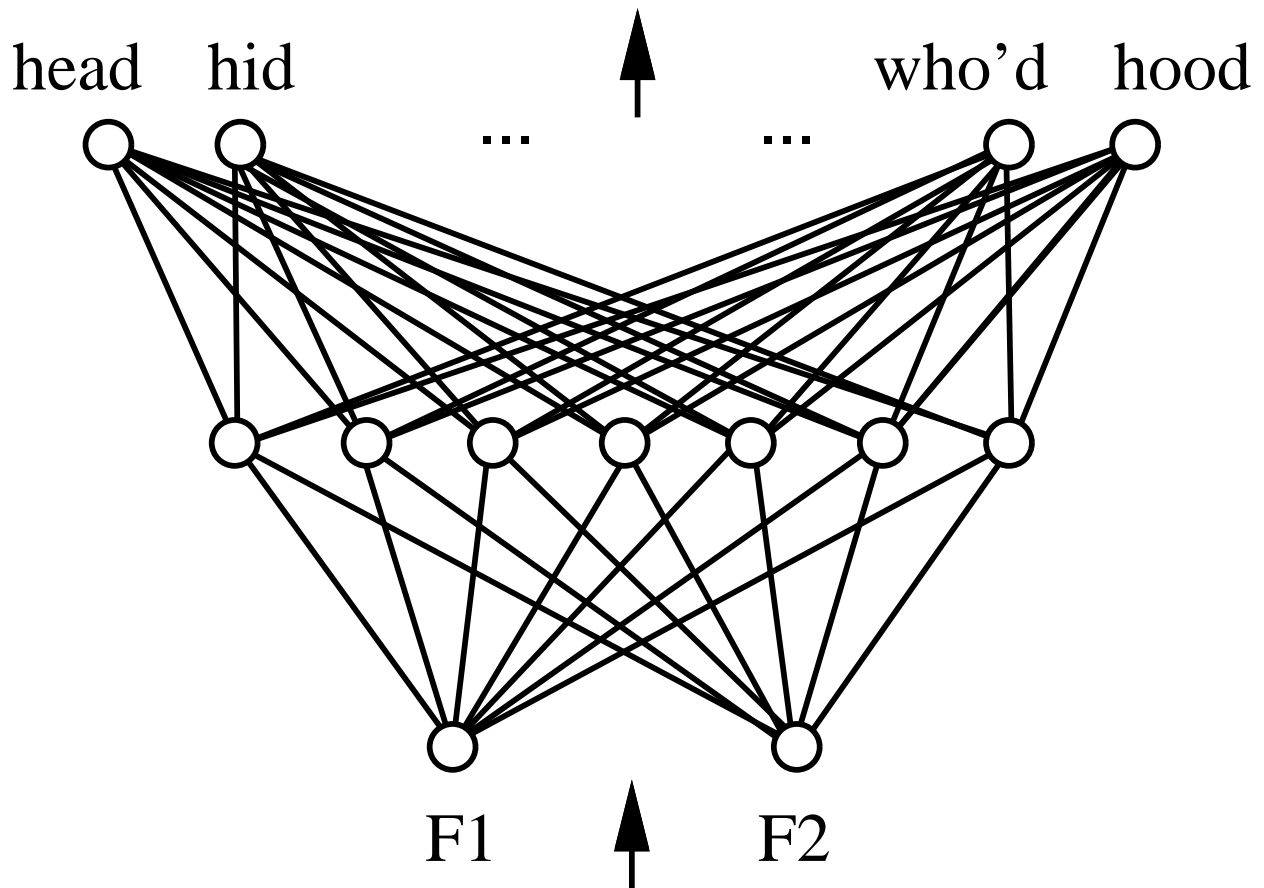
- Incremental Gradient Descent can approximate Batch Gradient Descent arbitrarily closely if η made small enough

1. Only update weights for current example

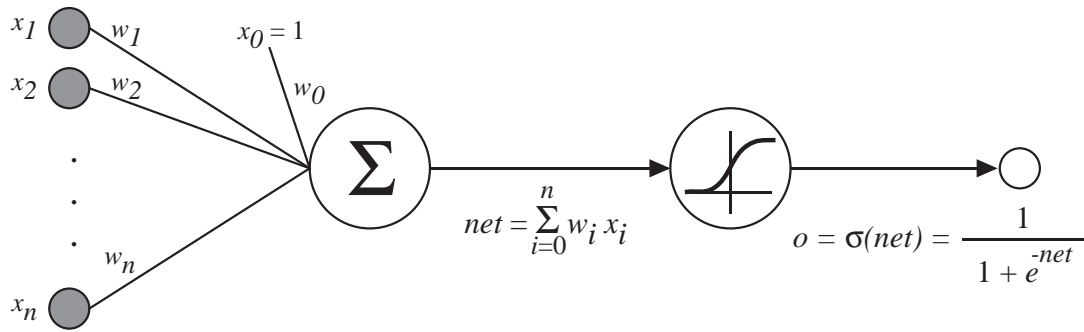
2. Fast updates

3. Performs well when multiple local minima exist

Multilayer Networks of Sigmoid Units



Sigmoid Unit



- Sigmoid function: $\sigma(x) = \frac{1}{1 + e^{-x}}$

Properties:

1. Continuous and non-linear

2. Differentiable: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

Hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Derivative = $[1 + \tanh(x)][1 - \tanh(x)]$

Better alternative to the sigmoid

- We can derive gradient decent rules to train

1. One sigmoid unit

2. *Multilayer networks* of sigmoid units

→ 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 we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

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

So:

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

Backpropagation Algorithm

Until termination criterion is met, Do

- For each training example $\langle \vec{x}, t \rangle$, Do

{Propagate the input forward through the network}

1. Input \vec{x} to network and get all network outputs

{Propagate the errors backward through the network}

2. For each output unit k , calculate its error term

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

3. For each hidden unit h , calculate its error term

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$$

4. Update each network weight $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where

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

More on 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 run multiple times)

- Often include weight *momentum* α

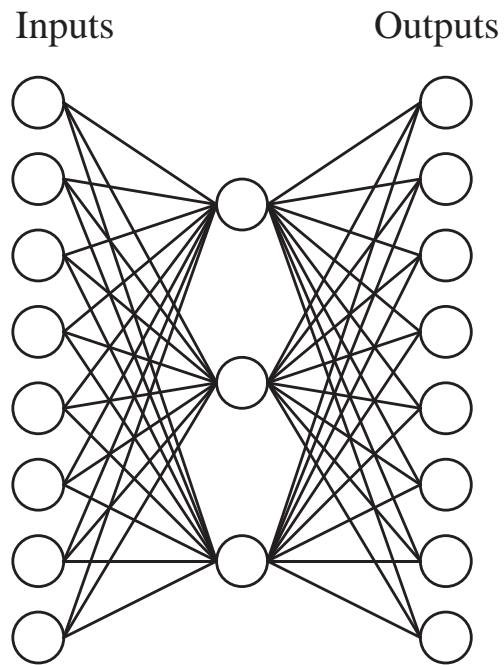
$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- 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

Learning Hidden Layer Representations



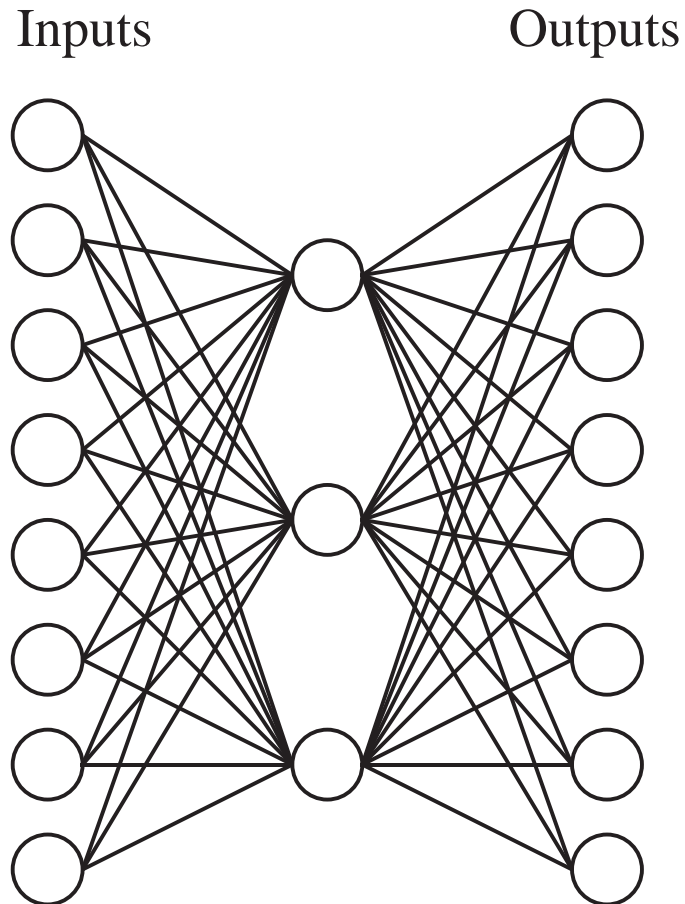
A target function:

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

Can this be learned??

Learning Hidden Layer Representations

A network:

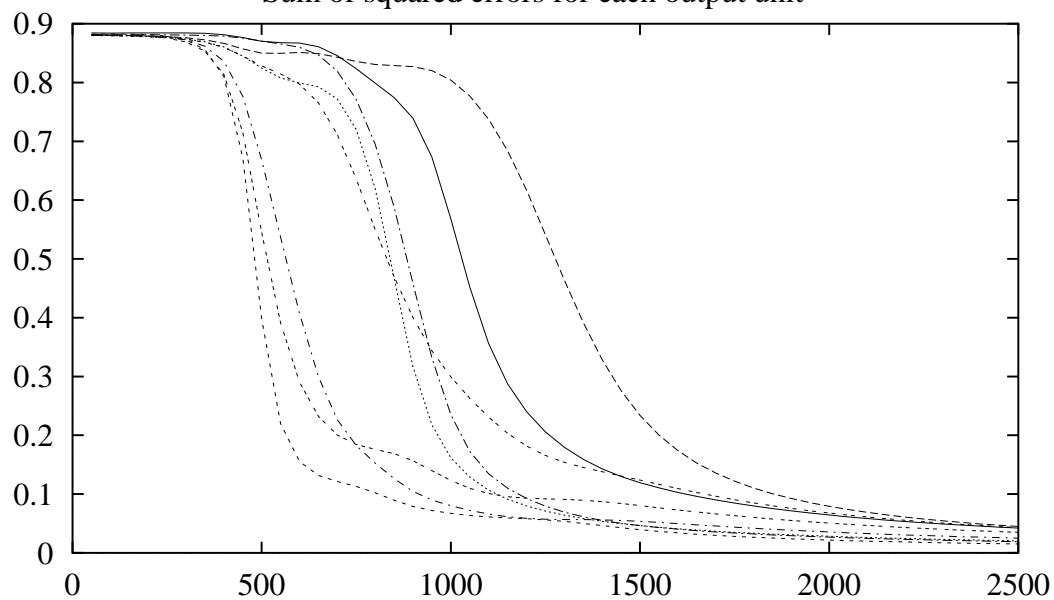


Learned hidden layer representation:

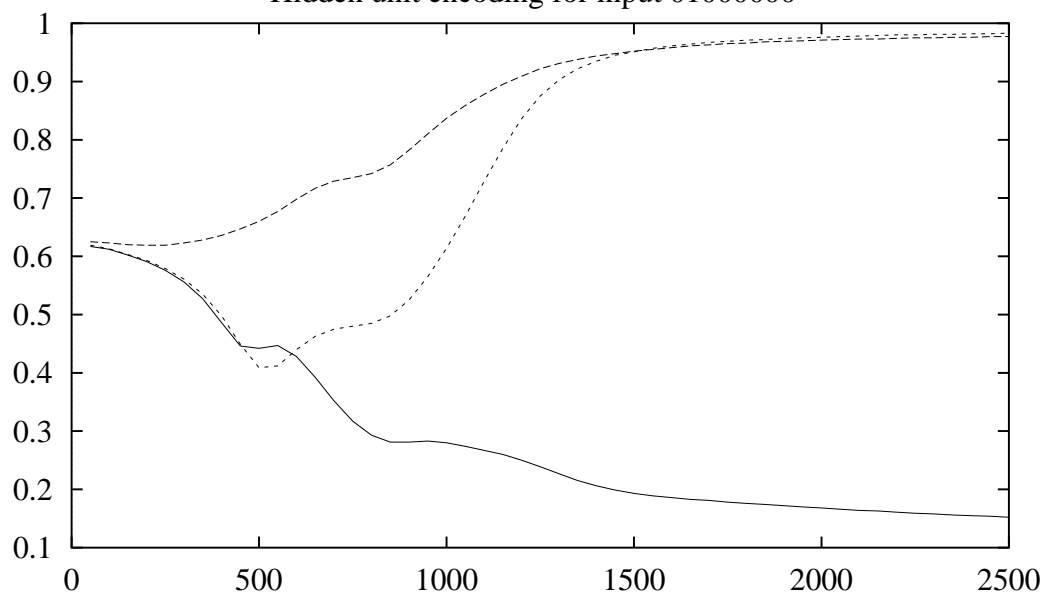
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Training

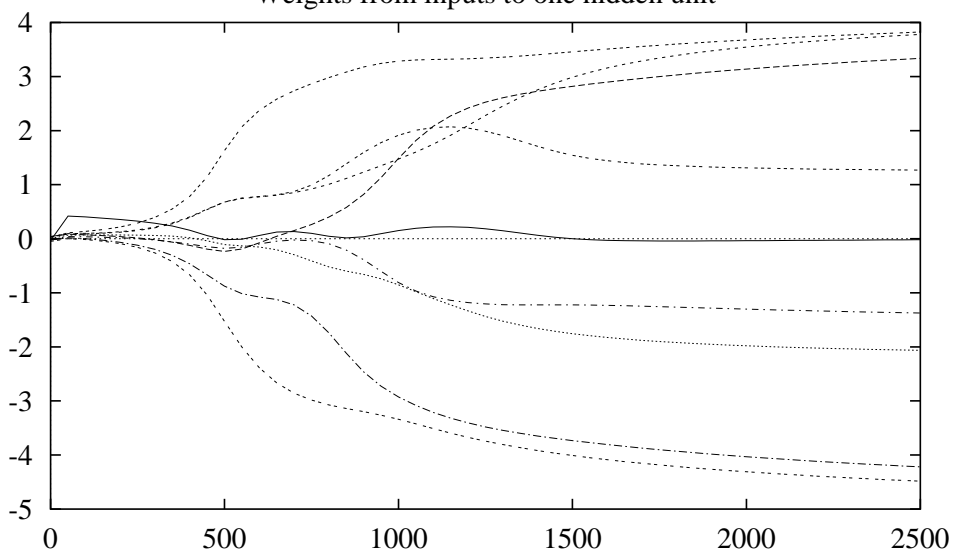
Sum of squared errors for each output unit



Hidden unit encoding for input 01000000



Weights from inputs to one hidden unit



Convergence of Backpropagation

- Gradient descent to some local minimum
 1. Perhaps not global minimum . . .
 2. Add momentum
 3. Stochastic gradient descent
 4. Train multiple nets with different initial weights
 5. Which network topology (architecture)?
 6. Other methods to escape local minima
- Nature of convergence
 1. Initialize weights near zero
 2. Therefore, initial networks near-linear
 3. Increasingly non-linear functions possible as training progresses

Expressive Capabilities of ANNs

- Boolean functions:

Every boolean function can be represented by network with single hidden layer

But might require exponential 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]

Applies only to networks with sigmoid as hidden units and adaline as output units

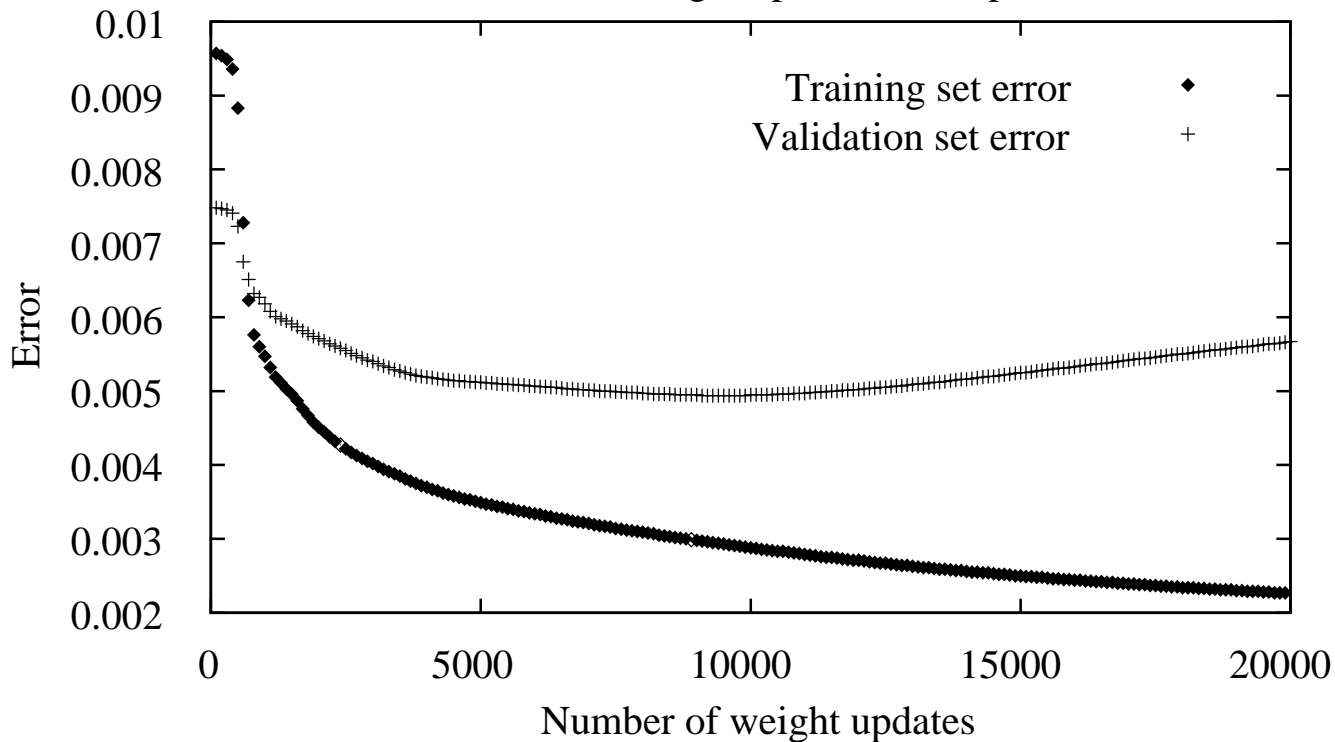
- Arbitrary functions:

Any function can be approximated to arbitrary accuracy by a network with two hidden layers

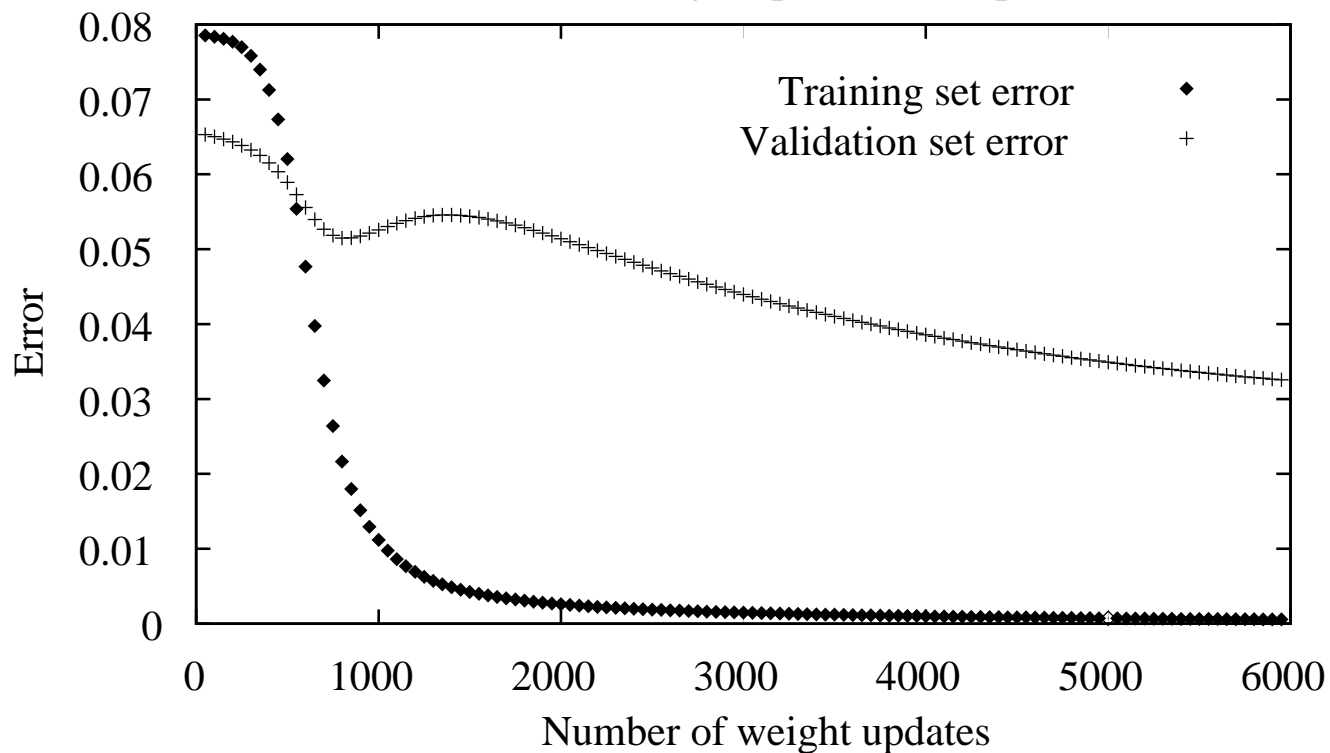
Applies only to networks with sigmoid as hidden units and adaline as output units [Cybenko 1988].

Overfitting in ANNs

Error versus weight updates (example 1)

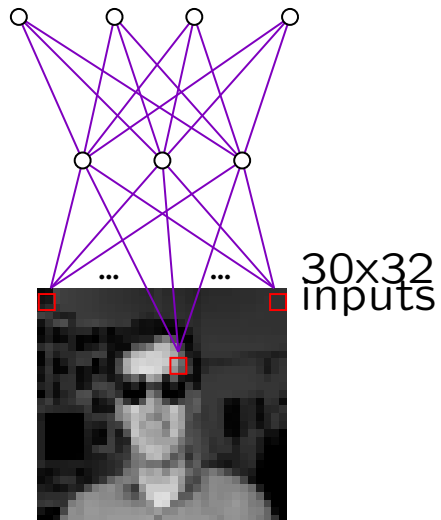


Error versus weight updates (example 2)



Neural Nets for Face Recognition

left strt right up



Typical input images

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

<http://www.cs.cmu.edu/~tom/faces.html>

Alternative Error Functions and Minimization

- 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:

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

- Minimize the *cross entropy* of network w.r.t. target

$$- \sum_{d \in D} t_d \log o_d + (1 - t_d) \log(1 - o_d)$$

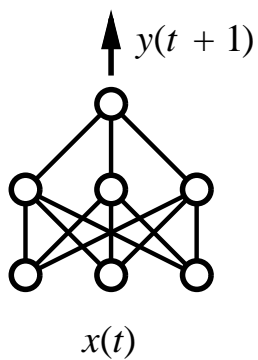
- Minimize *root (mean) squared error*

- Weight sharing

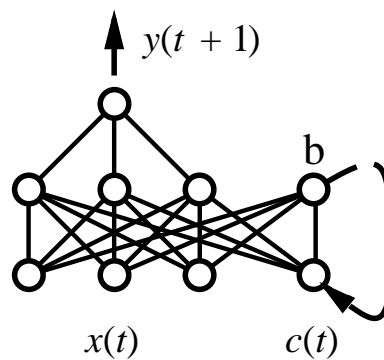
- *Line search*

- *Conjugate gradient*

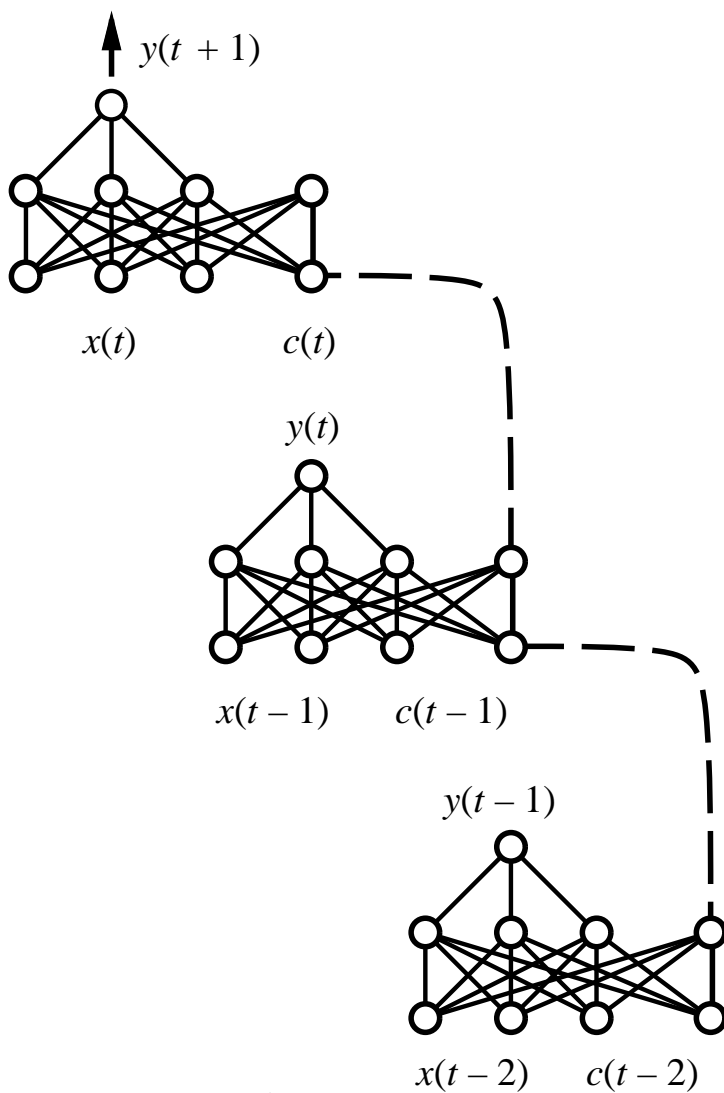
Recurrent Networks



(a) Feedforward network



(b) Recurrent network



(c) Recurrent network unfolded in time