



CMSC726 Spring 2006: Neural Networks

readings: Mitchell ch. 4

sources: course slides are based on material from a variety of sources, including Tom Dietterich, Carlos Guestrin, Terran Lane, **Rich Maclin**, Ray Mooney, Andrew Moore, Andrew Ng, Jude Shavlik, and others.

● ● ● | Artificial Neural Networks

- 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 step does not seem like enough

must use lots of parallel computation!

Properties of artificial neural nets (ANNs):

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

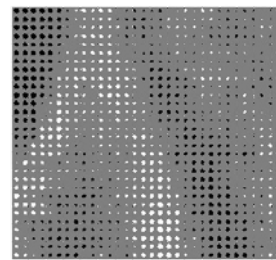
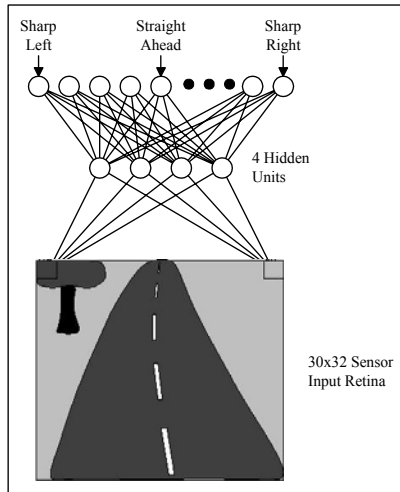
● ● ● | When to Consider Neural Networks

- 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*

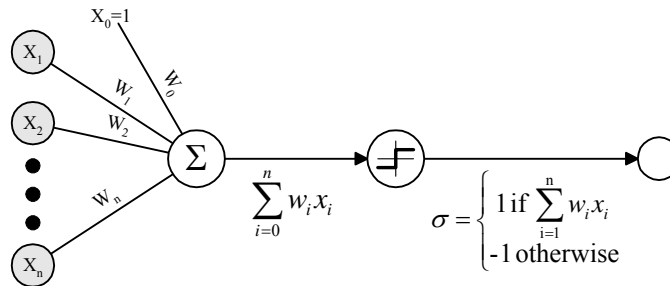
Examples:

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

● ● ● | ALVINN drives 70 mph on highways



● ● ● | Perceptron

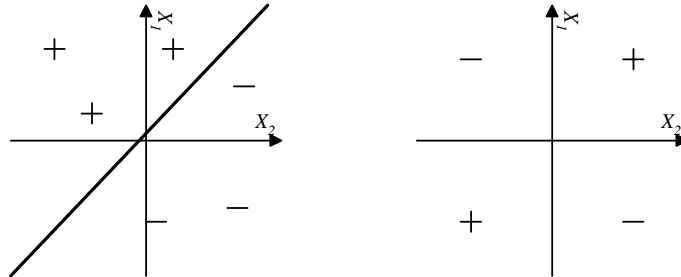


$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

Sometimes we will 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 Perceptron



Represents some useful functions

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

But some functions not representable

- e.g., not linearly separable
- therefore, we will want networks of these ...

● ● ● | Perceptron Training Rule

○

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

where

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

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

Can prove it will converge

- If training data is linearly separable
- and η is sufficiently small

● ● ● | Linear Threshold Unit

To understand, consider simple linear unit, where

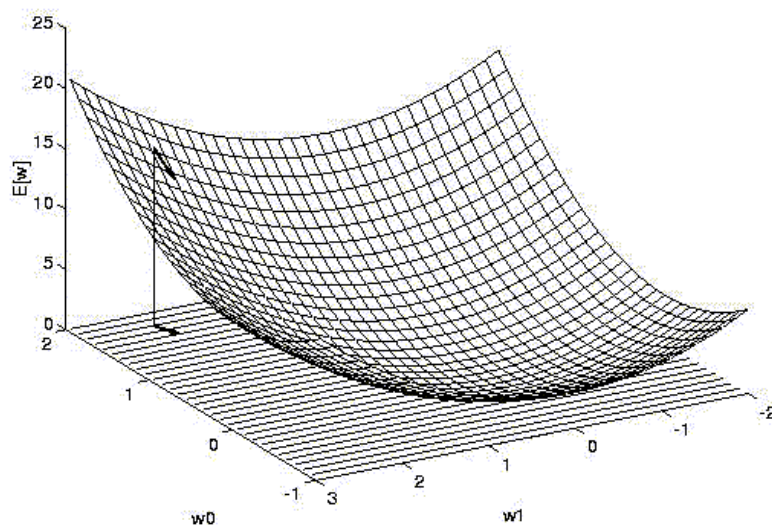
$$o = w_0 + w_1x_1 + \dots + w_nx_n$$

Idea : learn w_i 's that minimize the squared error

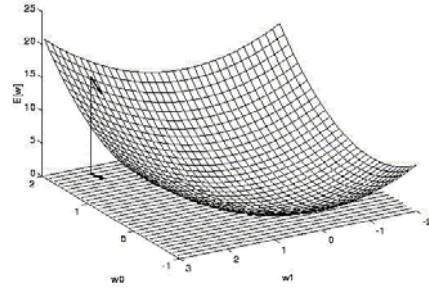
$$E[\tilde{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is the set of training examples

● ● ● | Gradient Descent



● ● ● | Gradient Descent



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

$$\text{Training rule: } \Delta w_i = -\eta \nabla E[\vec{w}]$$

$$\text{i.e., } \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

● ● ● | 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($\text{training_examples}, \eta$)

Each training examples is a pair of the form $\langle \bar{x}, t \rangle$, where \bar{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 $\langle \bar{x}, t \rangle$ in training_examples , do
 - * Input the instance \bar{x} 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

● ● ● | Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:

Do until satisfied:

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

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

Incremental mode Gradient Descent:

Do until satisfied:

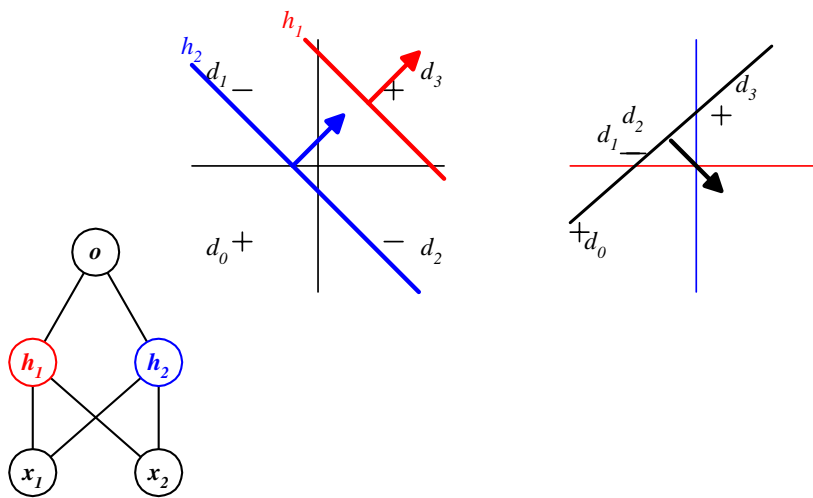
- For each training example d in D

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

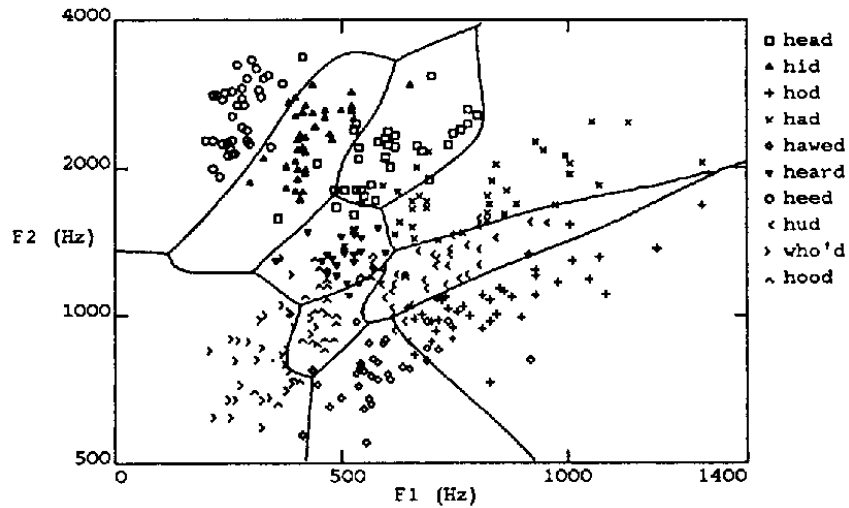
$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

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

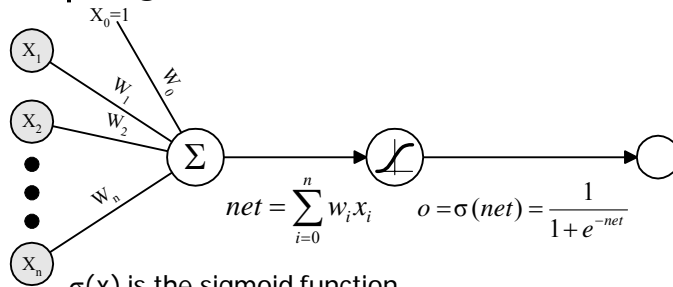
● ● ● | Multilayer Networks of Sigmoid Units



● ● ● | Multilayer Decision Space



● ● ● | Sigmoid Unit



$\sigma(x)$ is the sigmoid function

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

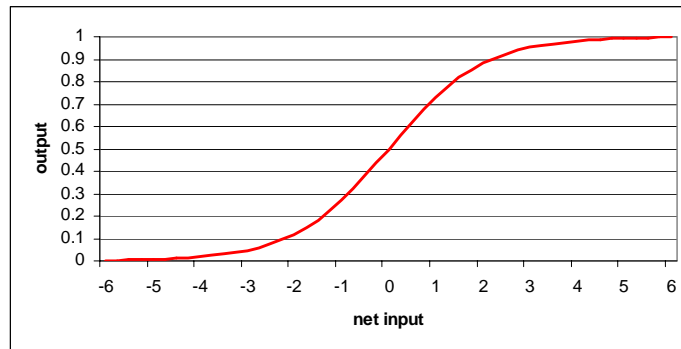
Recall: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

We can derive gradient descent rules to train

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

● ● ● | The Sigmoid Function

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



Sort of a rounded step function

Unlike step function, can take derivative (makes learning possible)

● ● ● | 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 :

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

So :

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

● ● ● | Backpropagation Algorithm

Initialize all weights to small random numbers. Until satisfied, do

- For each training example, do
 1. Input the training example and compute the 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
$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{k,h} \delta_k$$
 4. Update each network weight $w_{i,j}$

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

where

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

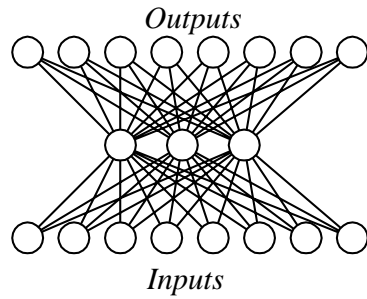
● ● ● | More on Backpropagation

- 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

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

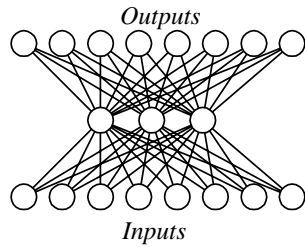
- Minimizes error over training examples
- Will it generalize well to subsequent examples?
- Training can take thousands of iterations -- **slow!**
 - Using network after training is fast

● ● ● | Learning Hidden Layer Representations



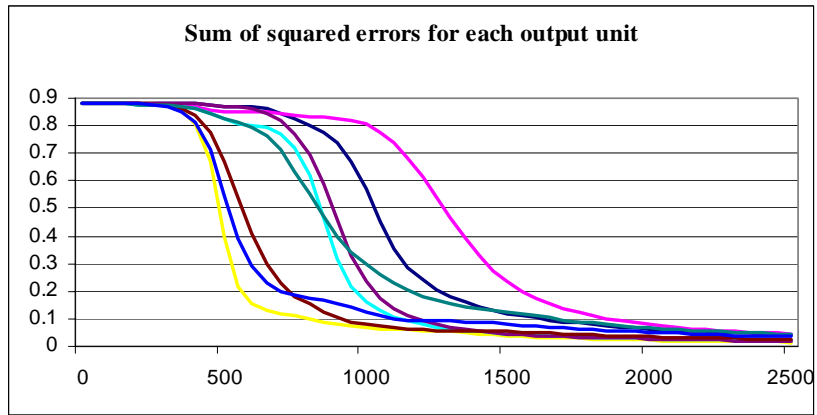
Input	Output
10000000	→ 10000000
01000000	→ 01000000
00100000	→ 00100000
00010000	→ 00010000
00001000	→ 00001000
00000100	→ 00000100
00000010	→ 00000010
00000001	→ 00000001

● ● ● | Learning Hidden Layer Representations

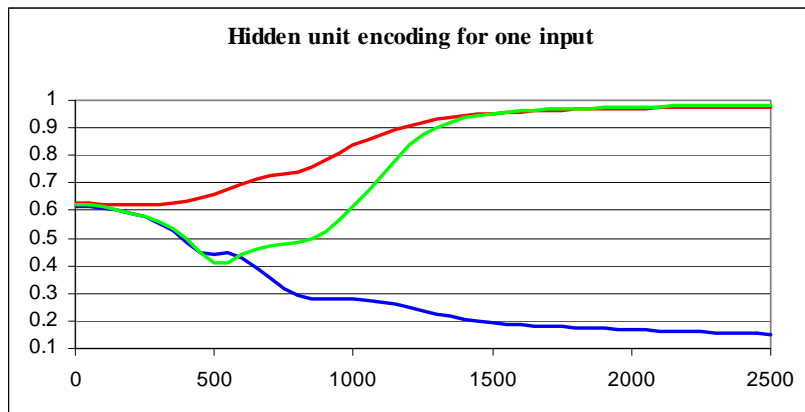


Input	Output
10000000	→ .89.04.08 → 10000000
01000000	→ .01.11.88 → 01000000
00100000	→ .01.97.27 → 00100000
00010000	→ .99.97.71 → 00010000
00001000	→ .03.05.02 → 00001000
00000100	→ .22.99.99 → 00000100
00000010	→ .80.01.98 → 00000010
00000001	→ .60.94.01 → 00000001

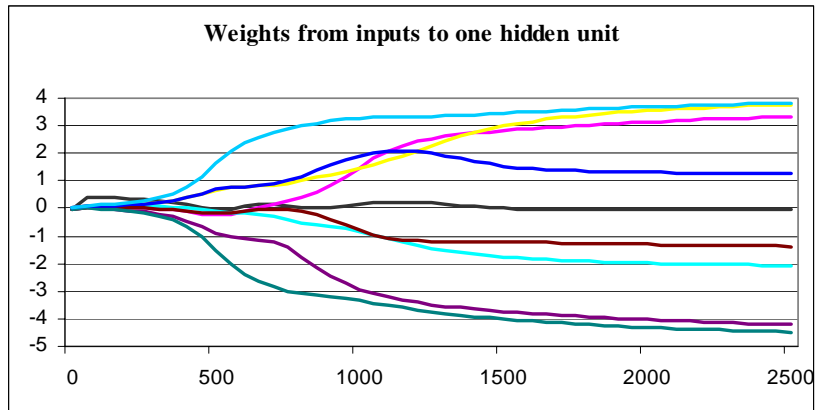
● ● ● | Output Unit Error during Training



● ● ● | Hidden Unit Encoding



● ● ● | Input to Hidden Weights



● ● ● | Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum
- Momentum can cause quicker convergence
- Stochastic gradient descent also results in faster convergence
- Can train multiple networks and get different results (using different initial weights)

Nature of convergence

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

● ● ● | Expressive Capabilities of ANNs

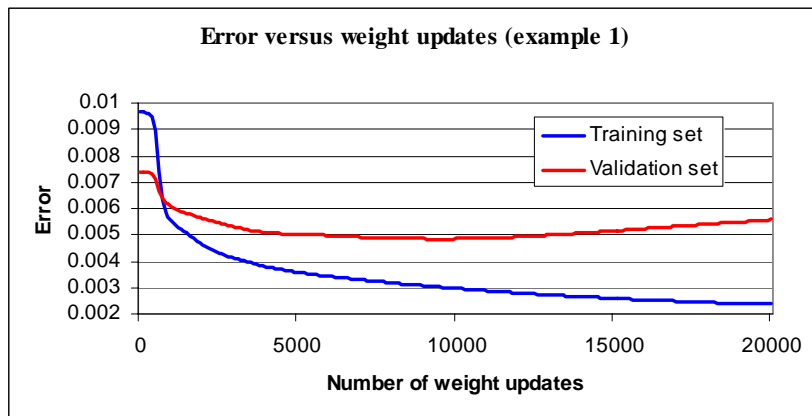
Boolean functions:

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

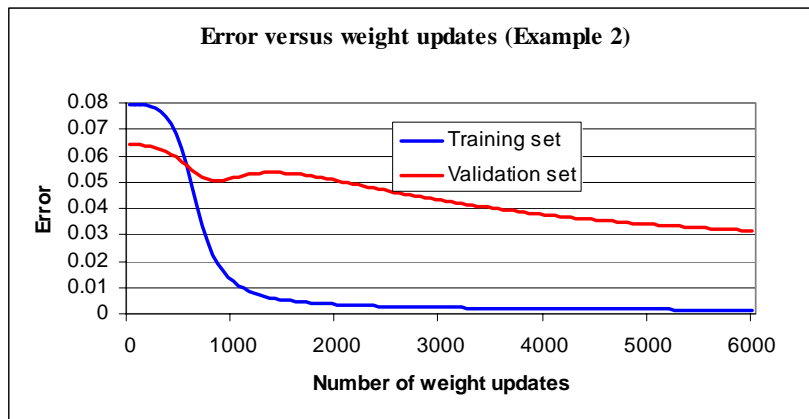
Continuous functions:

- 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]

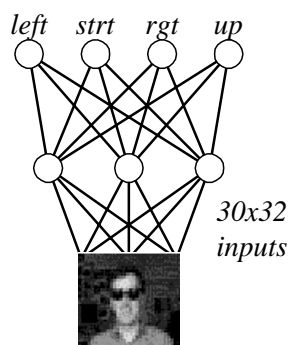
● ● ● | Overfitting in ANNs



● ● ● | Overfitting in ANNs



● ● ● | Neural Nets for Face Recognition

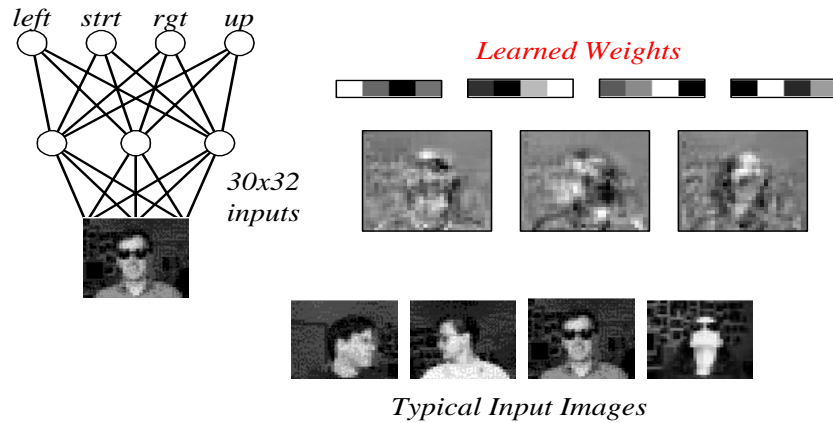


90% accurate learning
head pose, and recognizing
1-of-20 faces



Typical Input Images

● ● ● | Learned Network Weights



● ● ● | Alternative Error Functions

Penalize large weights :

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

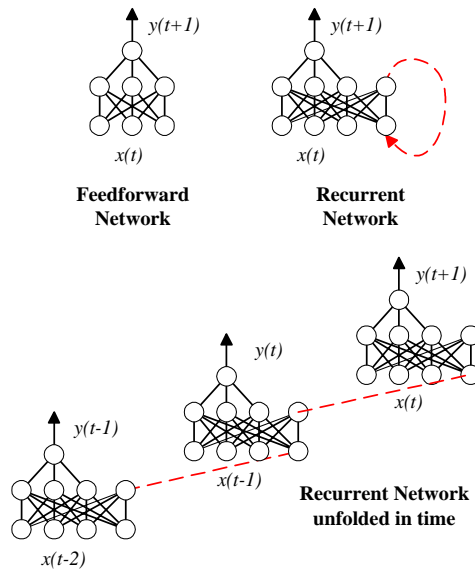
Train on target slopes as well as values :

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

Tie together weights :

- e.g., in phoneme recognition

● ● ● | Recurrent Networks



● ● ● | What you should know

- ANNs are practical method for learning real-valued and vector-valued functions over continuous and discrete inputs
- Backpropagation can be used to find weights for multi-layer ANNs
- Overfitting is an issue for ANNs
- Many, many variants that we have not covered