# Neural Networks

Slides taken from Vibhav Gogate, Pedro Domingos, Carlos Guestrin, Luke Zettlemoyer and Dan Weld

# Who needs probabilities?

- Previously: model data with distributions
- Joint: P(X,Y)
  - e.g. Naïve Bayes
- Conditional: P(Y|X)
  - e.g. Logistic Regression
- But wait, why probabilities?
- Lets try to be error-driven!

| mpg | cylinders | displacemen | horsepower | weight | acceleration | modelyear | make |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| good | 4 | 97 | 75 | 2265 | 18.2 | 77 | asia |
| bad | 6 | 199 | 90 | 2648 | 15 | 70 | ameri |
| bad | 4 | 121 | 110 | 2600 | 12.8 | 77 | europ |
| bad | 8 | 350 | 175 | 4100 | 13 | 73 | ameri |
| bad | 6 | 198 | 95 | 3102 | 16.5 | 74 | ameri |
| bad | 4 | 108 | 94 | 2379 | 16.5 | 73 | asia |
| bad | 4 | 113 | 95 | 2228 | 14 | 71 | asia |
| bad | 8 | 302 | 139 | 3570 | 12.8 | 78 | ameri |
| : | : | : | : | : | : | : | : |
| : | : | : | : | : | : | : | : |
| : | : | : | : | : | : | : | : |
| good | 4 | 120 | 79 | 2625 | 18.6 | 82 | ameri |
| bad | 8 | 455 | 225 | 4425 | 10 | 70 | ameri |
| good | 4 | 107 | 86 | 2464 | 15.5 | 76 | europ |
| bad | 5 | 131 | 103 | 2830 | 15.9 | 78 | europ |
| | | | | | | | |

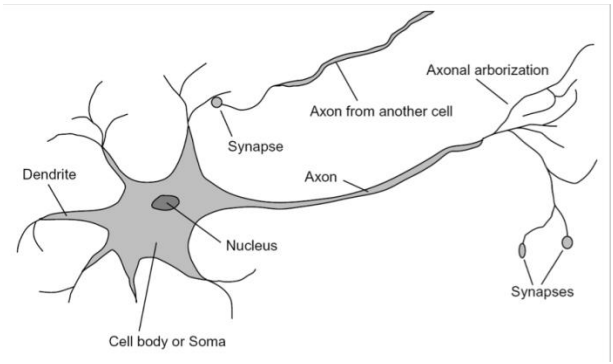
# Generative vs. Discriminative

- Generative classifiers:
  - E.g. naïve Bayes
  - A joint probability model with evidence variables
  - Query model for causes given evidence

- Discriminative classifiers:
  - No generative model, no Bayes rule, often no probabilities at all!
  - Try to predict the label Y directly from X
  - Robust, accurate with varied features
  - Loosely: mistake driven rather than model driven

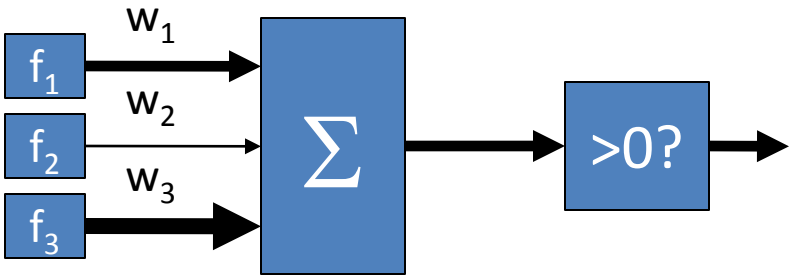
# Linear Classifiers

- Inputs are feature values
- Each feature has a weight
- Sum is the activation

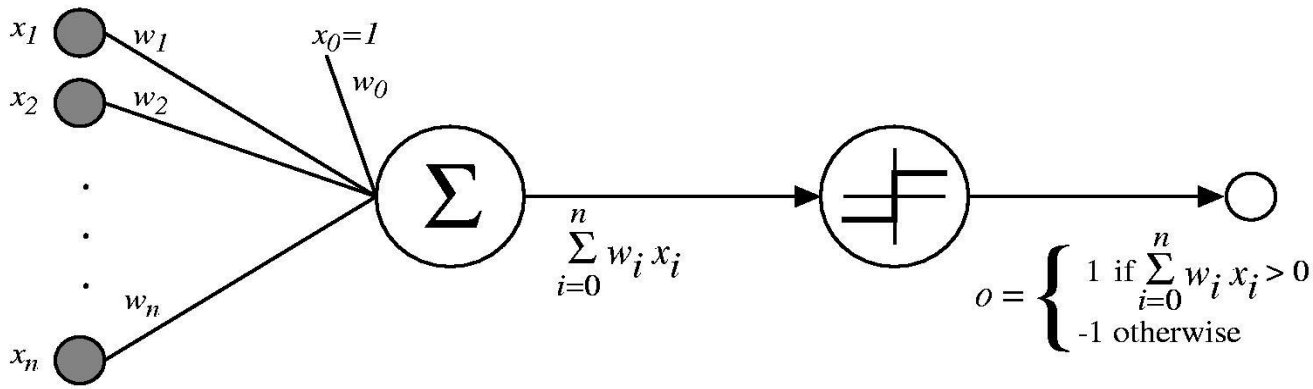



$$\text{activation}_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)$$

- If the activation is:
  - Positive, output *class 1*
  - Negative, output *class 2*



# Perceptron




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

Sometimes we'll use simpler vector notation:

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

# Gradient Descent

To understand, consider simpler *linear unit*, where

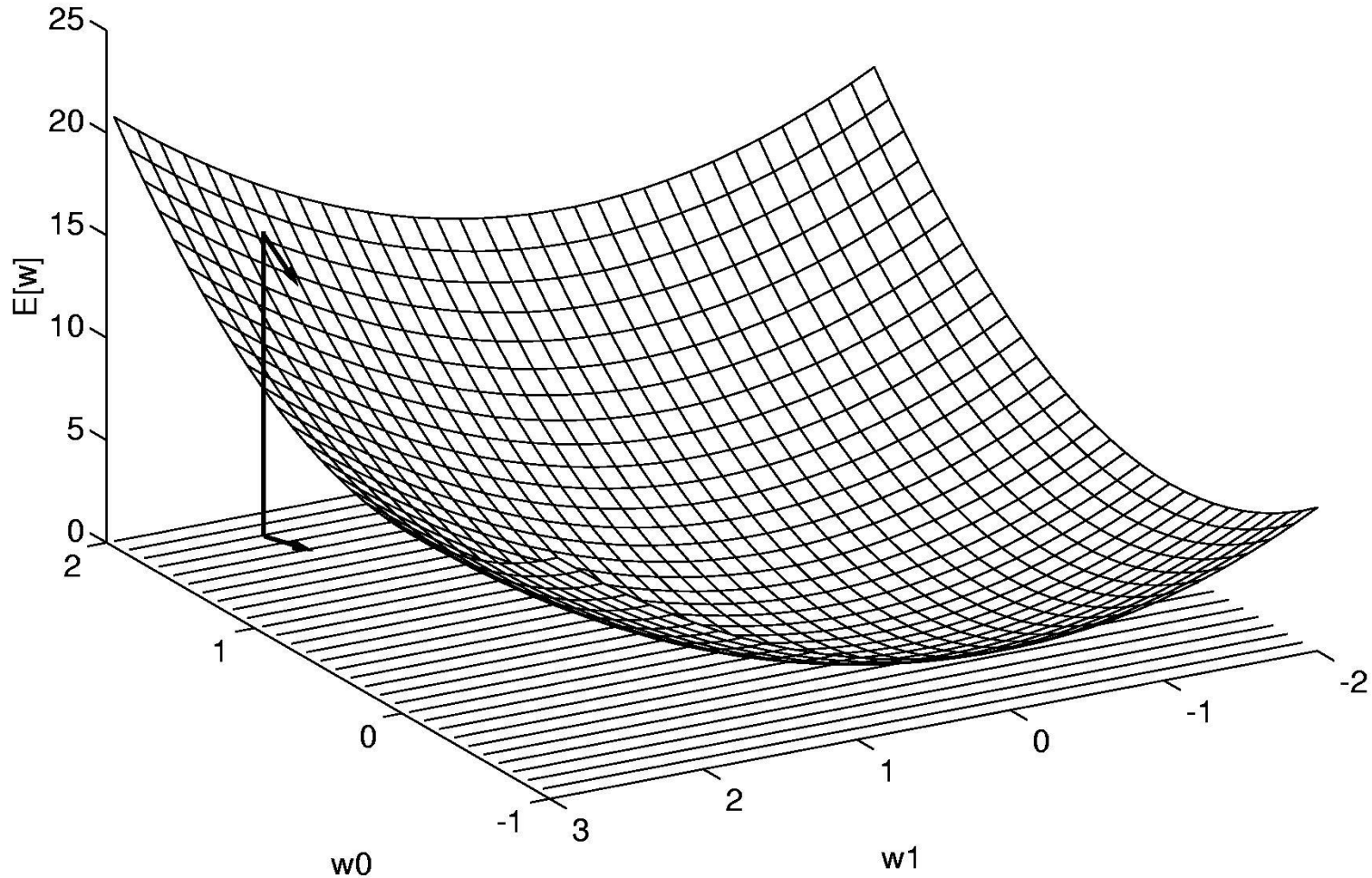
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

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:

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

Training rule:

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

I.e.:

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

# Gradient Descent

GRADIENT-DESCENT($training_examples, \eta$)

Initialize each $w_i$ to some small random value

Until the termination condition is met, Do

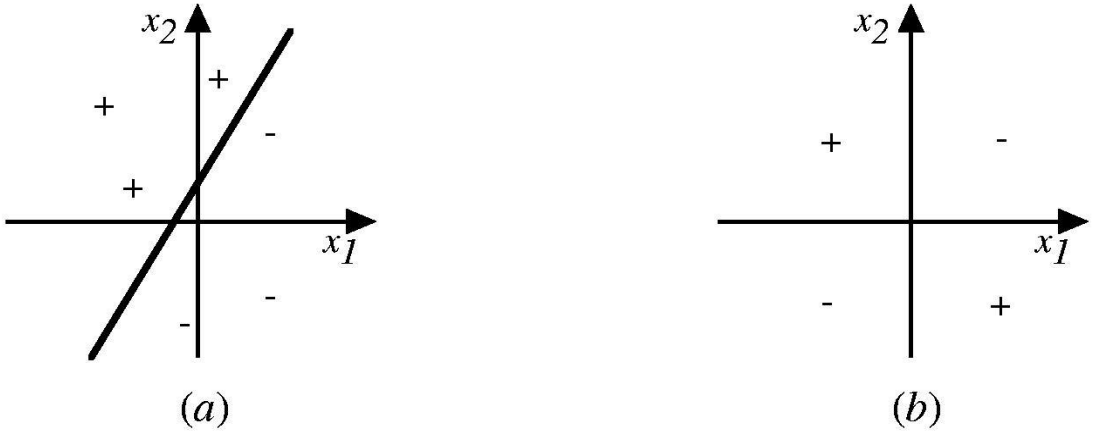
- Initialize each $\Delta w_i$ to zero.
- For each $\langle \vec{x}, t \rangle$ in $training_examples$, Do
  - Input instance $\vec{x}$ to unit and compute output $o$
  - For each linear unit weight $w_i$, Do

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

- For each linear unit weight $w_i$, Do

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

# Decision Surface of a Perceptron




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
- $\eta$ is small constant (e.g., 0.1) called *learning rate*

# From Logistic Regression to the Perceptron: 2 easy steps!

- Logistic Regression: (in vector notation): y is {0,1}

$$w = w + \eta \sum_j [y_j^* - p(y_j^* | x_j, w)] f(x_j)$$

- Perceptron: y is {0,1}, y(x;w) is prediction given w

$$w = w + [y^* - y(x;w)] f(x)$$

Differences?

•Drop the $\Sigma_j$ over training examples: online vs. batch learning

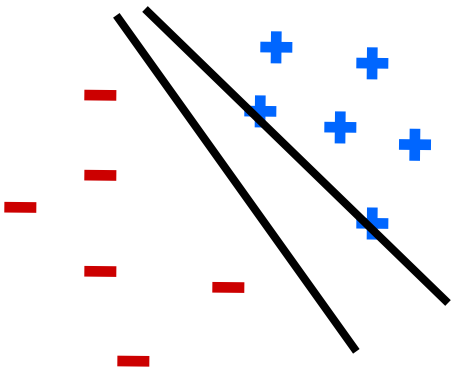
•Drop the dist'n: probabilistic vs. error driven learning

# Properties of Perceptrons

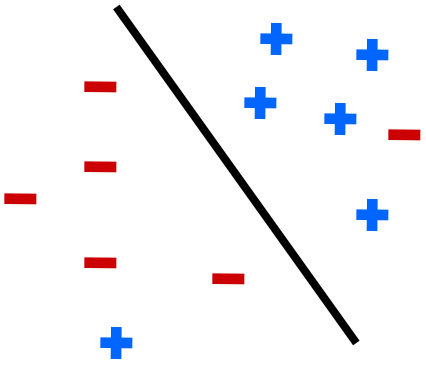
- Separability: some parameters get the training set perfectly correct

- Convergence: if the training is separable, perceptron will eventually converge (binary case)

Separable

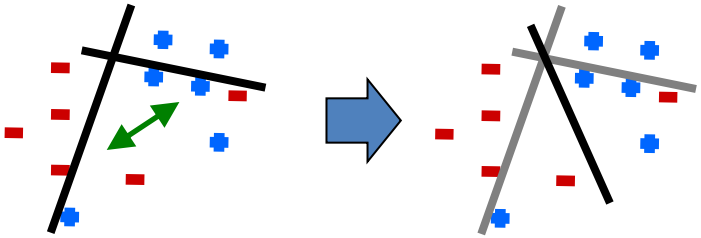


Non-Separable

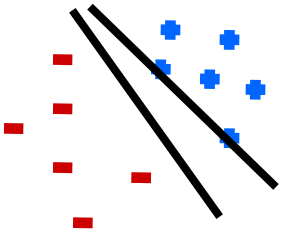


# Problems with the Perceptron

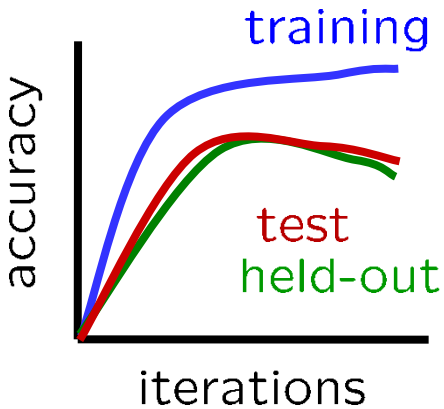
- Noise: if the data isn't separable, weights might thrash
  - Averaging weight vectors over time can help (averaged perceptron)



- Mediocre generalization: finds a "barely" separating solution



- Overtraining: test / validation accuracy usually rises, then falls
  - Overtraining is a kind of overfitting



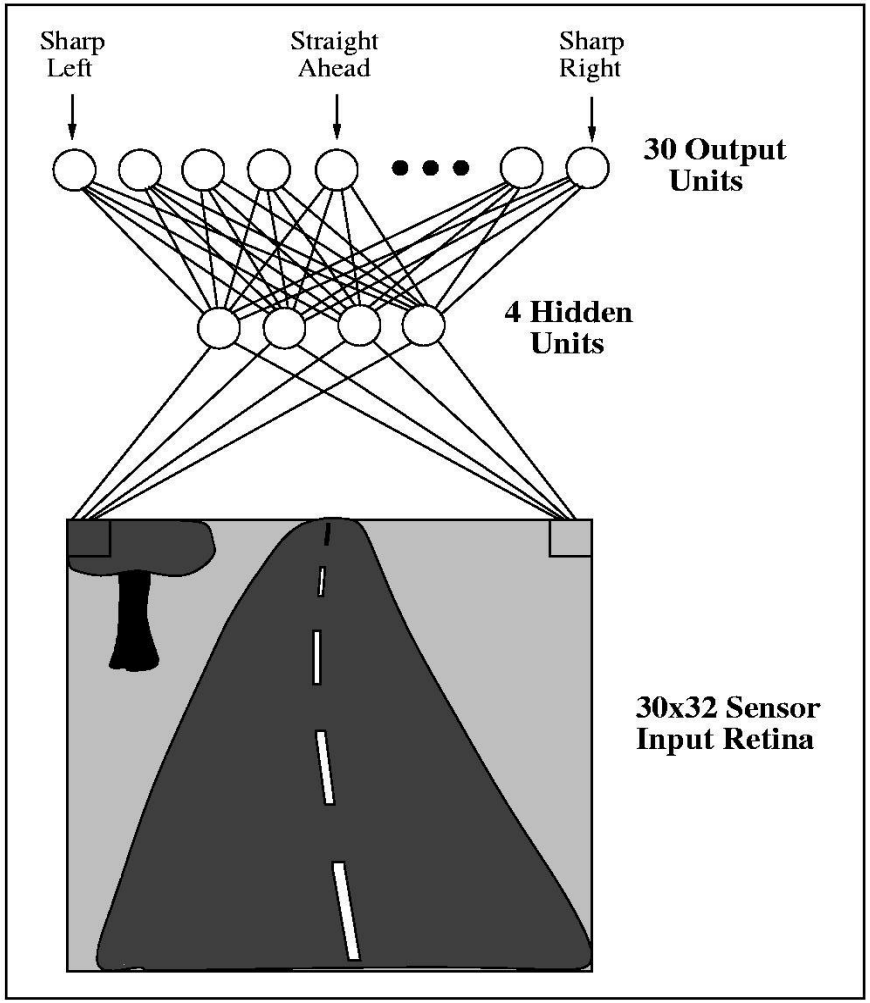
# Summary

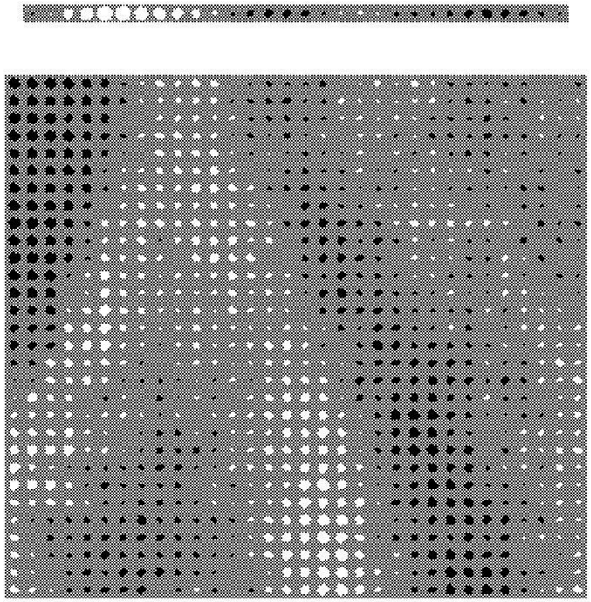
Perceptron training rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate $\eta$

Linear unit training rule uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate $\eta$
- Even when training data contains noise
- Even when training data not separable by $H$


Sharp
Left
Straight
Ahead
Sharp
Right
30 Output
Units
4 Hidden
Units
30x32 Sensor
Input Retina



# Connectionist Models

**Consider humans:**

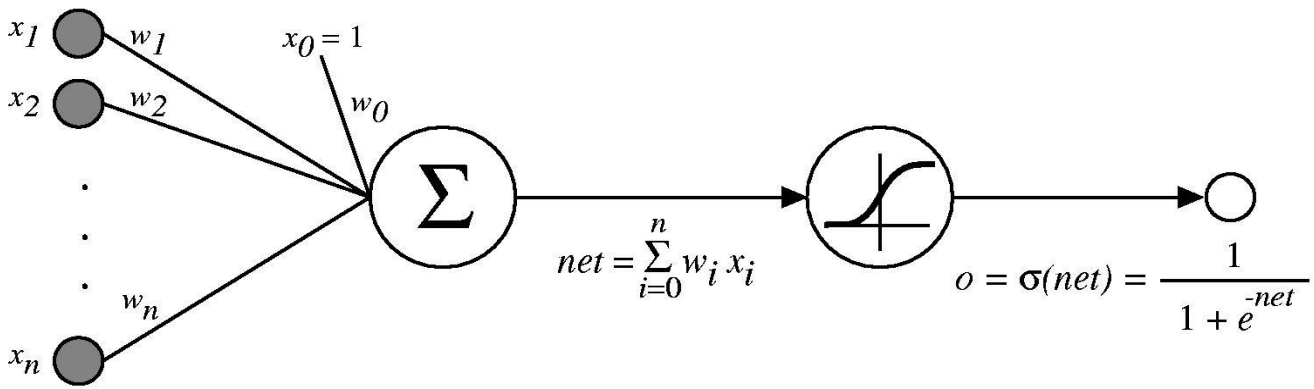
- Neuron switching time $\sim$ .001 second
- Number of neurons $\sim 10^{10}$
- Connections per neuron $\sim 10^{4-5}$
- Scene recognition time $\sim$ .1 second
- 100 inference steps doesn't seem like enough

$\Rightarrow$ 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

# Sigmoid Unit

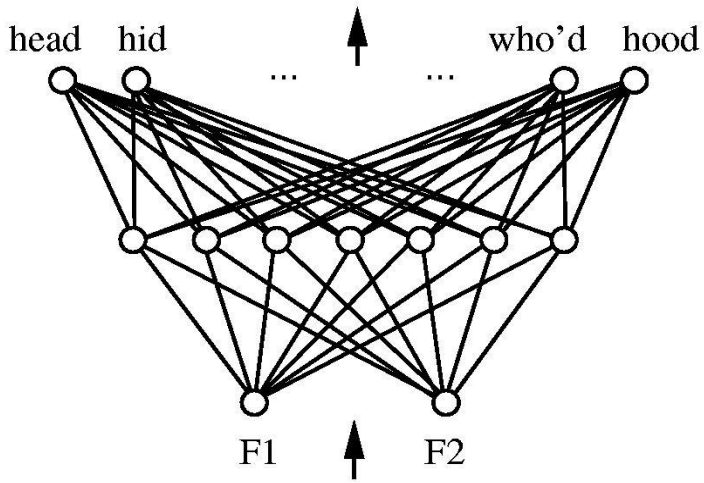



$\sigma(x)$ is the sigmoid function

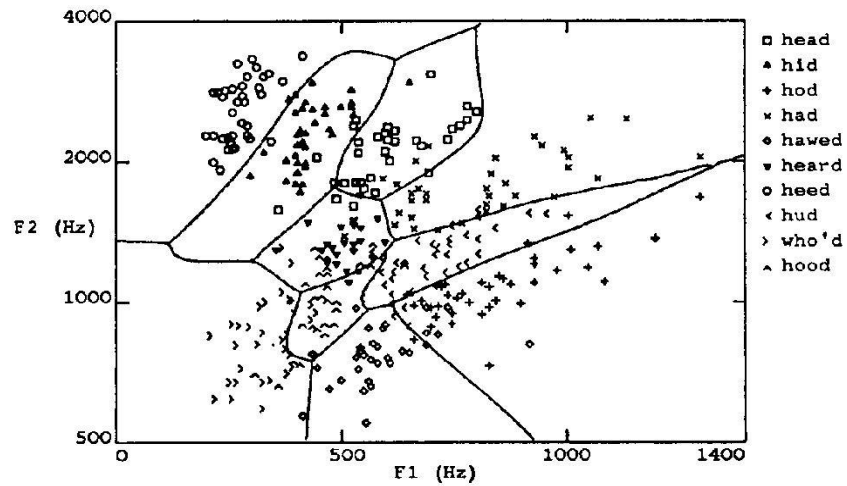
$$\frac{1}{1 + e^{-x}}$$

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

# Multilayer Networks of Sigmoid Units




$$out(\mathbf{x}) \;=\; g\left(w_0 + \sum_k w_k g(w_0^k + \sum_i w_i^k x_i)\right)$$



We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units $\rightarrow$ Backpropagation

# Backpropagation Algorithm

Initialize all weights to small random numbers
Until convergence, Do
  For each training example, Do

1. Input it to network and compute 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 $h$

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

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

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

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

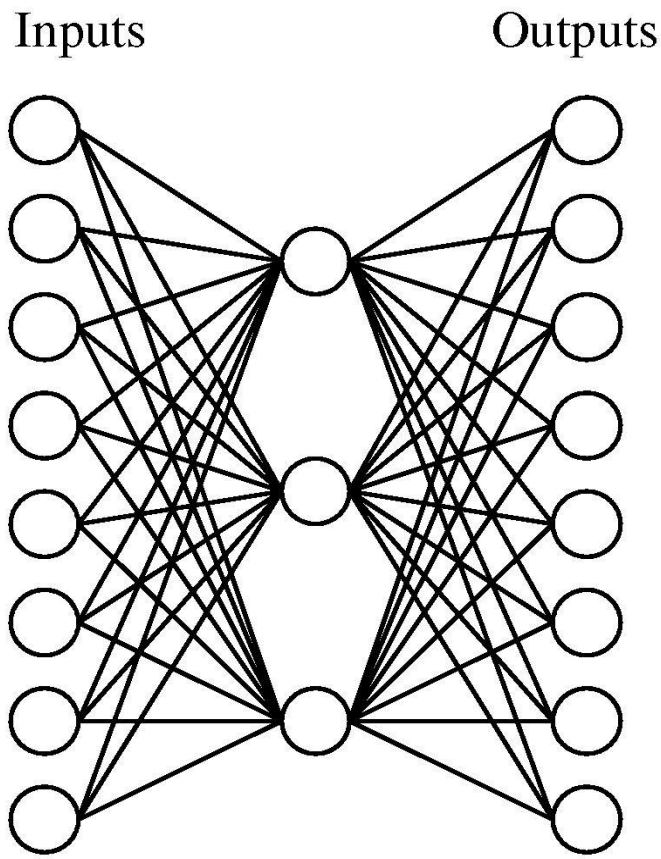
# More on Backpropagation

- 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* $\alpha$

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

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

# Learning Hidden Layer Representations



A target function:

| Input | | Output |
|---|---|---|
| 10000000 | $\longrightarrow$ | 10000000 |
| 01000000 | $\longrightarrow$ | 01000000 |
| 00100000 | $\longrightarrow$ | 00100000 |
| 00010000 | $\longrightarrow$ | 00010000 |
| 00001000 | $\longrightarrow$ | 00001000 |
| 00000100 | $\longrightarrow$ | 00000100 |
| 00000010 | $\longrightarrow$ | 00000010 |
| 00000001 | $\longrightarrow$ | 00000001 |

Can this be learned?

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 |

# Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different inital 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

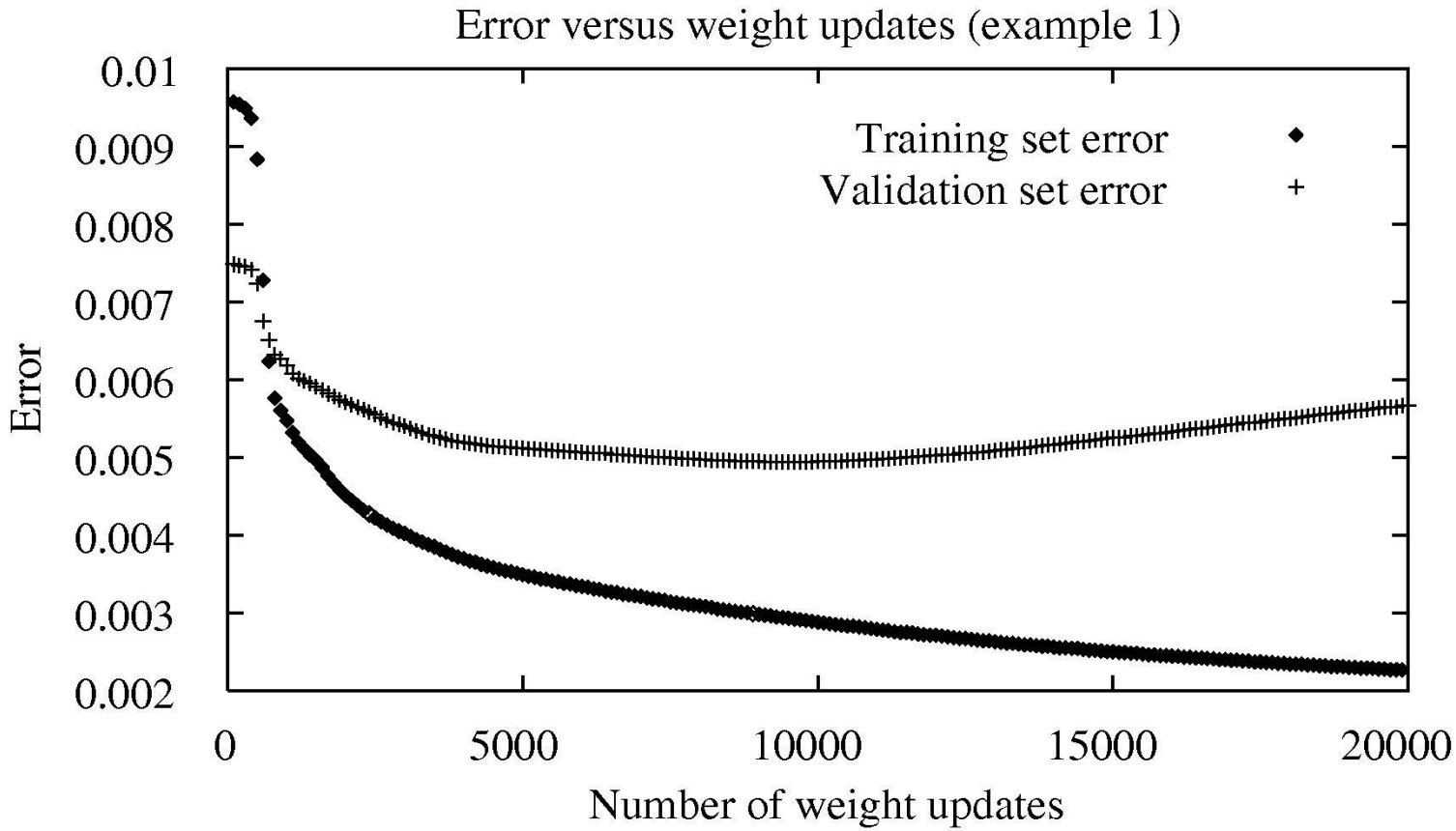
Boolean functions:

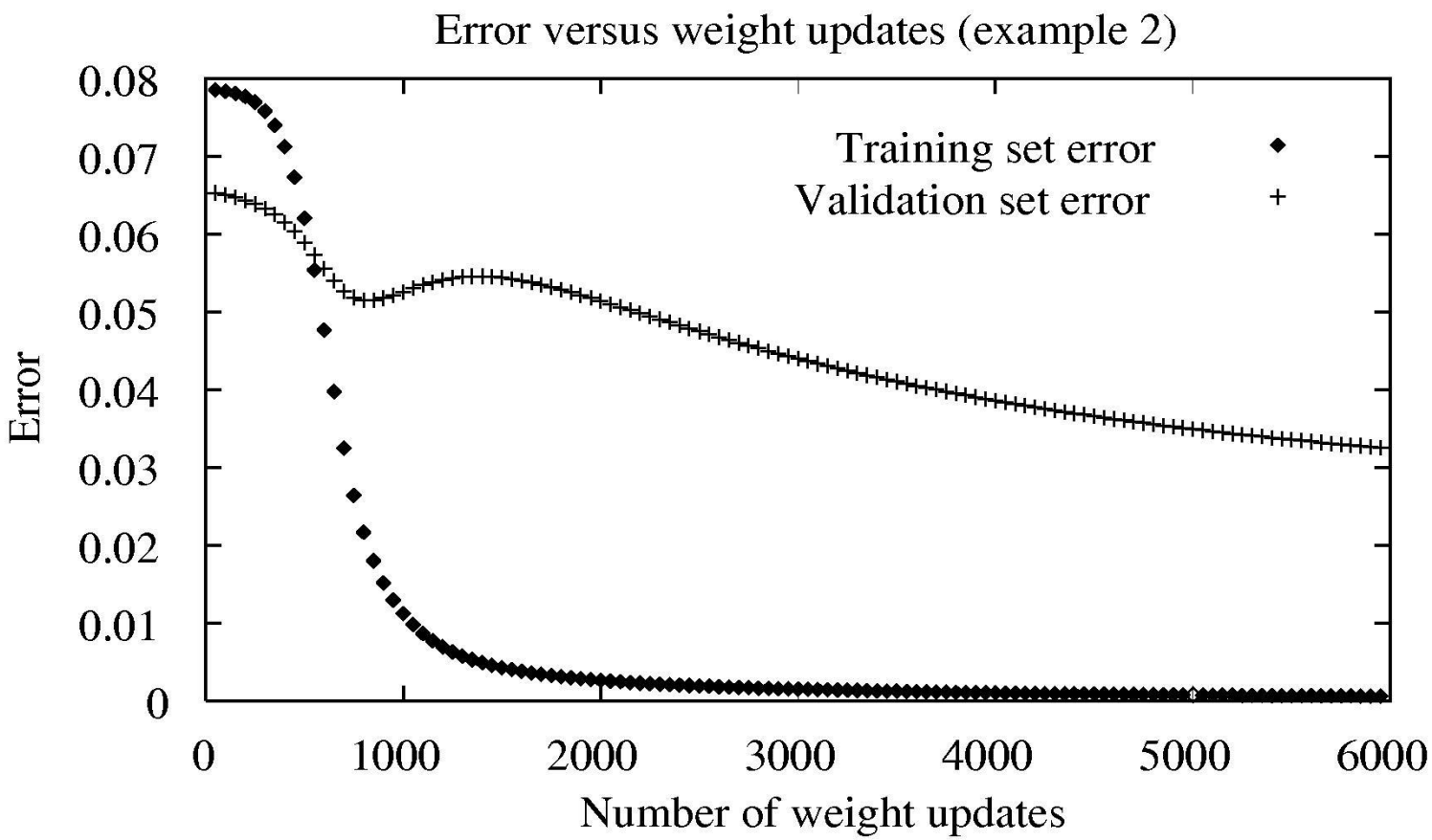
- Every Boolean function can be represented by network with single hidden layer
- But might require exponential (in number of inputs) hidden units

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




Error versus weight updates (example 2)
Training set error
Validation set error
Error
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
1000
2000
3000
4000
5000
6000
Number of weight updates

# Overfitting Avoidance

Penalize large weights:

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

Weight sharing

Early stopping

# Neural Networks: Summary

- Perceptrons
- Gradient descent
- Multilayer networks
- Backpropagation