

# CSCI 5090/7090- Machine Learning

Spring 2018

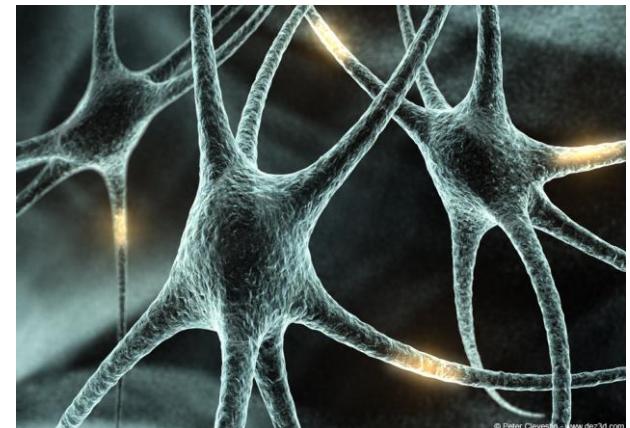
Mehdi Allahyari  
Georgia Southern University

## Neural Networks

(slides borrowed from Tom Mitchell, Maria Florina Balcan, Ali Farhadi)

# Artificial Neural Network (ANN)

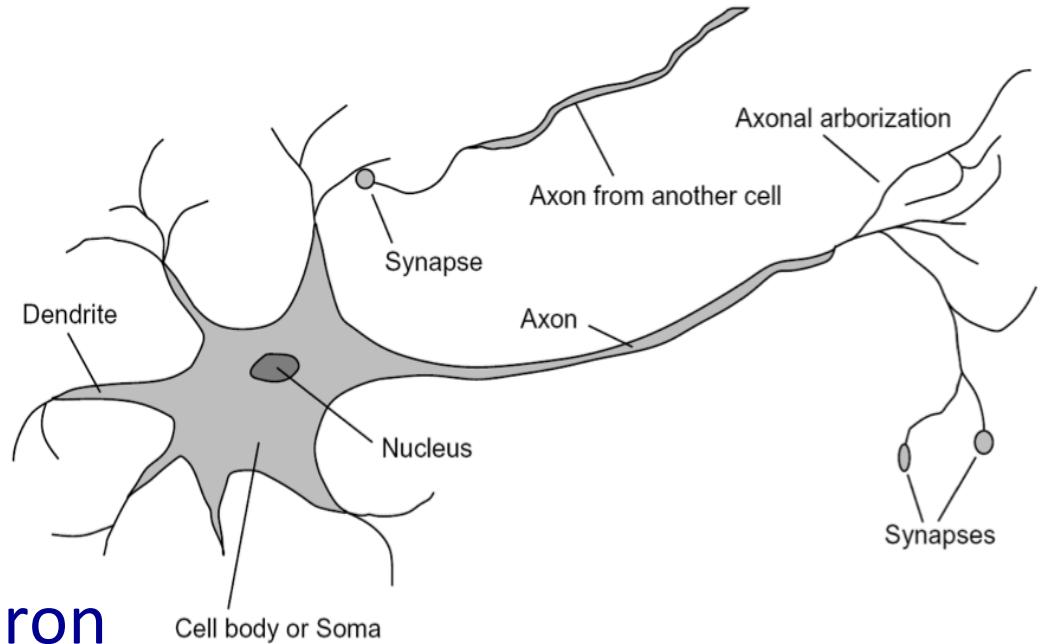
- **Biological systems** built of very complex webs of interconnected neurons.
- Highly connected to other neurons, and performs computations by combining signals from other neurons.
- Outputs of these computations may be transmitted to one or more other neurons.

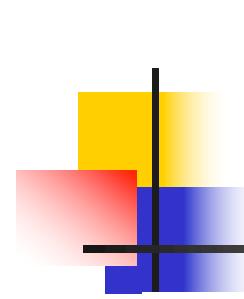


- **Artificial Neural Networks** built out of a densely interconnected set of simple units (e.g., sigmoid units).
- Each unit takes real-valued inputs (possibly the outputs of other units) and produces a real-valued output (which may become input to many other units).

# Human Neuron

- **Switching time**
  - $\sim 0.001$  second
- **Number of neurons**
  - $10^{10}$
- **Connections per neuron**
  - $10^{4-5}$
- **Scene recognition time**
  - 0.1 seconds
- **Number of cycles per scene recognition?**
  - 100 → much parallel computation!





# 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:

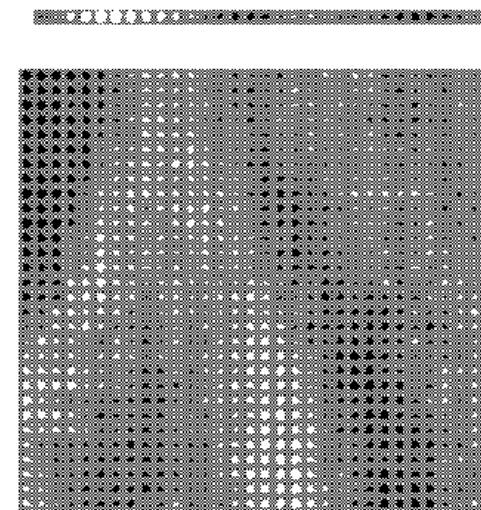
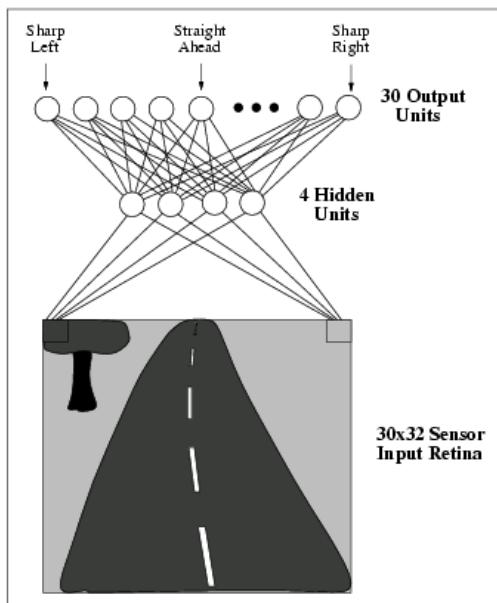
- Speech phoneme recognition [Waibel]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction

# Example



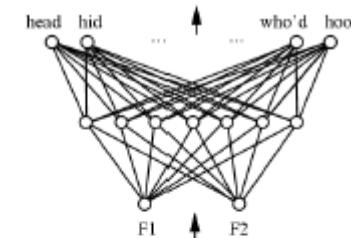
ALVINN

[Pomerleau 1993]



# Artificial Neural Networks to learn $f: X \rightarrow Y$

- $f_w$  typically a non-linear function,  $f_w: X \rightarrow Y$
- $X$  feature space: (vector of) continuous and/or discrete vars
- $Y$  output space: (vector of) continuous and/or discrete vars
- $f_w$  network of basic units

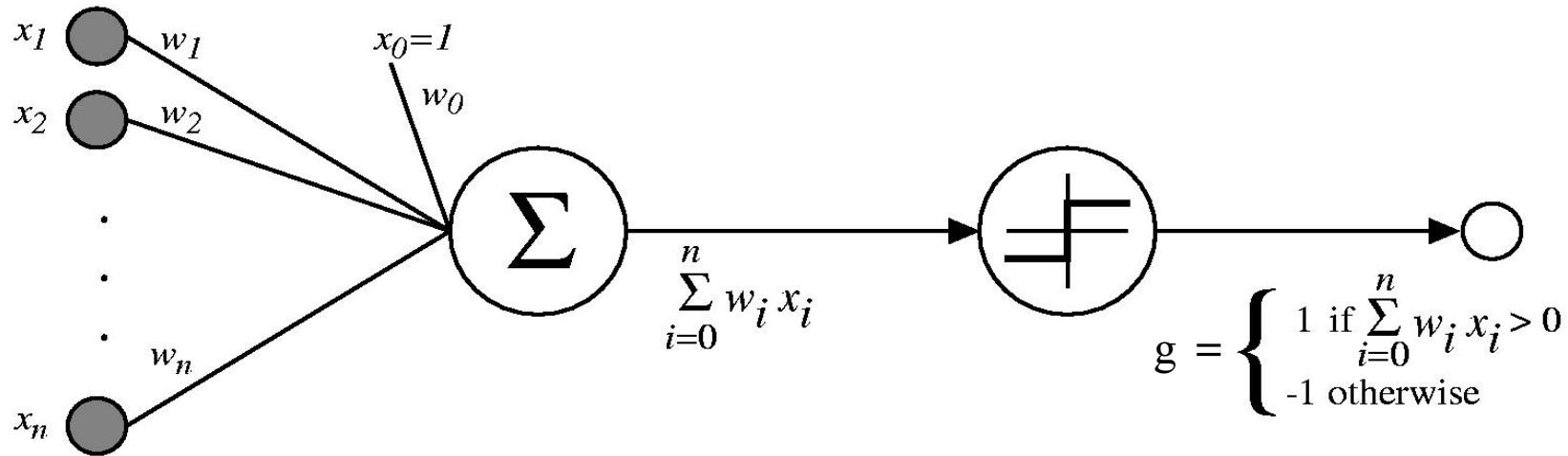


**Learning algorithm:** given  $(x_d, t_d)_{d \in D}$ , train weights  $w$  of all units to minimize sum of squared errors of predicted network outputs.

Find parameters  $w$  to minimize  $\sum_{d \in D} (f_w(x_d) - t_d)^2$

Use gradient descent!

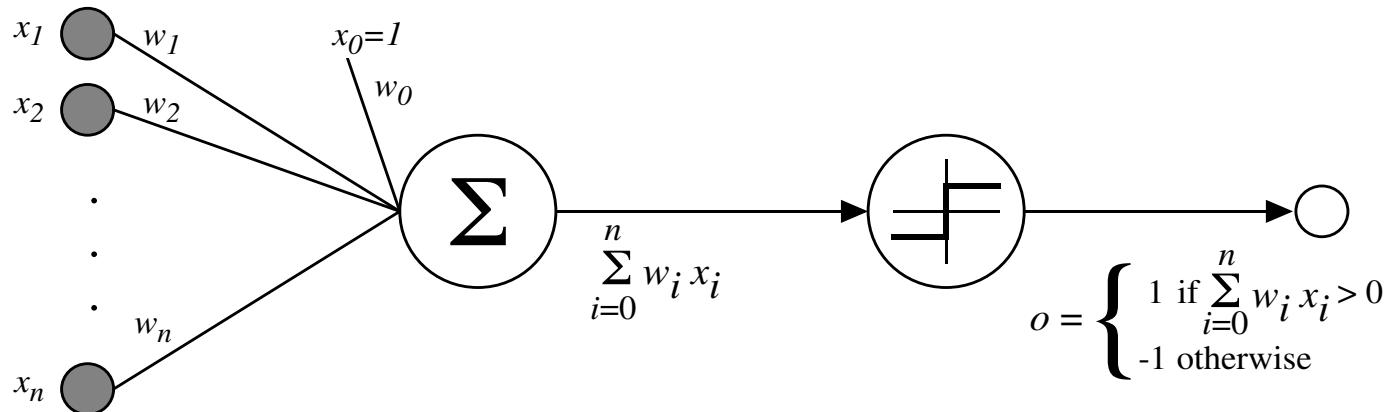
# Perceptron as a Neural Network



This is one neuron:

- Input edges  $x_1 \dots x_n$ , along with basis
- The sum is represented graphically
- Sum passed through an activation function  $g$

# Perceptron



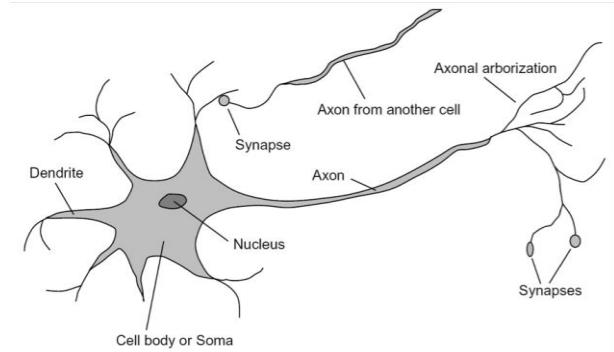
$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

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

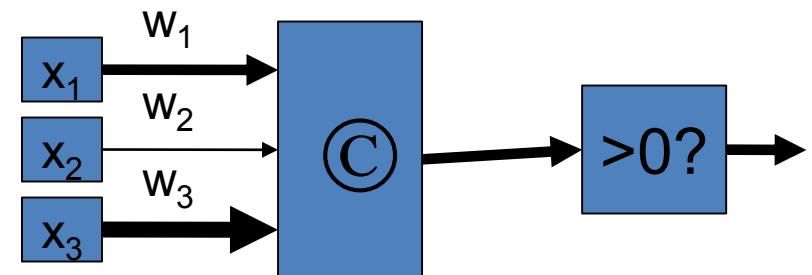
# Linear Classifiers

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



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

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



# Example: Spam

- Imagine 3 features (spam is “positive” class):
  - free (number of occurrences of “free”)
  - money (occurrences of “money”)
  - BIAS (intercept, always has value 1)

“free money”

	<i>x</i>	<i>w</i>
BIAS	: 1	: -3
free	: 1	: 4
money	: 1	: 2
...		...

$$(1)(-3) + (1)(4) + (1)(2) \dots = 3$$

$w \bullet x > 0 \rightarrow \text{SPAM!!!}$

# Binary Decision Rule

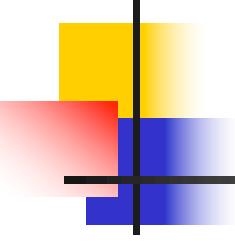
- Imagine 3 features (spam is “positive” class):
  - free (number of occurrences of “free”)
  - money (occurrences of “money”)
  - BIAS (intercept, always has value 1)

“free money”

	$x$	$w$
BIAS	: 1	: -3
free	: 1	: 4
money	: 1	: 2
...		...

$$(1)(-3) + (1)(4) + (1)(2) \dots = 3$$

$w \cdot x > 0 \rightarrow \text{SPAM!!!}$



# Perceptron Training Rule

---

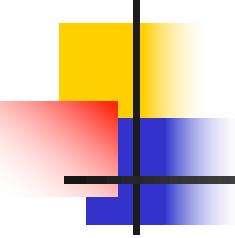
$$w_i \leftarrow w_i + \Delta w_i$$

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., .1) called *learning rate*



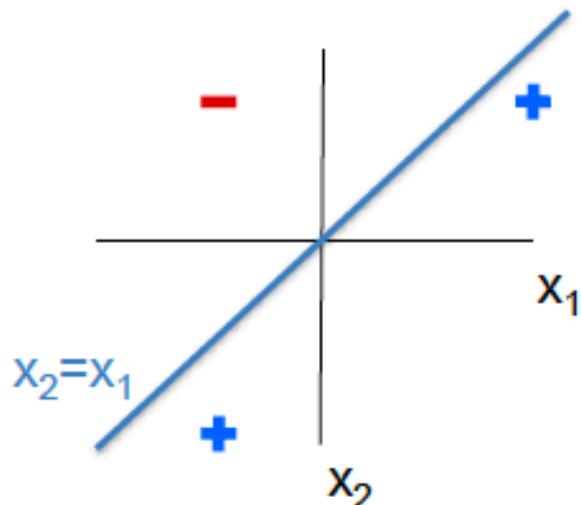
# Binary Perceptron Algorithm

- Start with zero weights:  $w=0$
- For  $t=1..T$  ( $T$  passes over data)
  - For  $i=1..n$ : (each training example)
    - Classify with current weights
$$y = \text{sign}(w \cdot x^i)$$
      - $\text{sign}(x)$  is  $+1$  if  $x>0$ , else  $-1$
    - If correct (i.e.,  $y=y^i$ ), no change!
    - If wrong: update

$$w = w + y^i x^i$$

- For  $t=1..T$ ,  $i=1..n$ :
  - $y = \text{sign}(w \cdot x^i)$
  - if  $y \neq y^i$   
 $w = w + y^i x^i$

$x_1$	$x_2$	$y$
3	2	1
-2	2	-1
-2	-3	1



Initial:

- $w = [0,0]$
- $t=1, i=1$
- $[0,0] \bullet [3,2] = 0$ ,  $\text{sign}(0)=-1$
- $w = [0,0] + [3,2] = [3,2]$
- $t=1, i=2$
- $[3,2] \bullet [-2,2] = -2$ ,  $\text{sign}(-2)=-1$
- $t=1, i=3$
- $[3,2] \bullet [-2,-3] = -12$ ,  $\text{sign}(-12)=-1$
- $w = [3,2] + [-2,-3] = [1,-1]$
- $t=2, i=1$
- $[1,-1] \bullet [3,2] = 1$ ,  $\text{sign}(1)=1$
- $t=2, i=2$
- $[1,-1] \bullet [-2,2] = -4$ ,  $\text{sign}(-4)=-1$
- $t=2, i=3$
- $[1,-1] \bullet [-2,-3] = 1$ ,  $\text{sign}(1)=1$

Converged!!!

- $y = w_1 x_1 + w_2 x_2 \rightarrow y = x_1 - x_2$
- So, at  $y=0 \rightarrow x_2 = x_1$

# Multiclass Decision Rule

- If we have more than two classes:

- Have a weight vector for each class:  $w_y$
  - Calculate an activation for each class

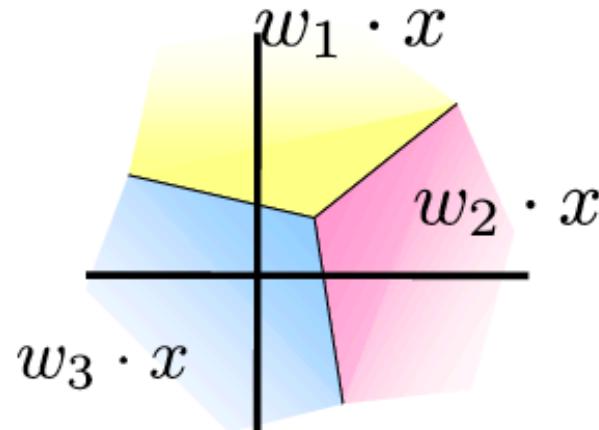
$$\text{activation}_w(x, y) = w_y \cdot x$$

- Highest activation wins

$$y^* = \arg \max_y (\text{activation}_w(x, y))$$

Example:  $y$  is  $\{1, 2, 3\}$

- We are fitting three planes:  $w_1$ ,  $w_2$ ,  $w_3$
- Predict  $i$  when  $w_i \cdot x$  is highest



# Example

“win the vote”



$x$

BIAS	:	1
win	:	1
game	:	0
vote	:	1
the	:	1
...		

$w_{SPORTS}$

BIAS	:	-2
win	:	4
game	:	4
vote	:	0
the	:	0
...		

$$x \cdot w_{SPORTS} = 2$$

$w_{POLITICS}$

BIAS	:	1
win	:	2
game	:	0
vote	:	4
the	:	0
...		

$$x \cdot w_{POLITICS} = 7$$

$w_{TECH}$

BIAS	:	2
win	:	0
game	:	2
vote	:	0
the	:	0
...		

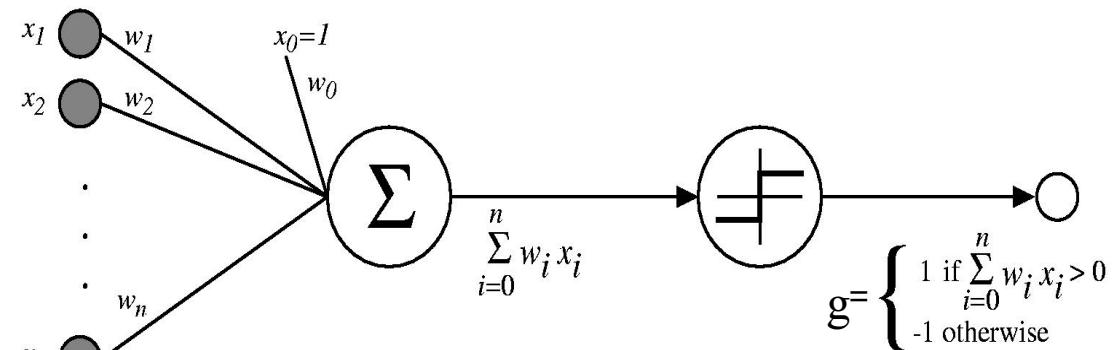
$$x \cdot w_{TECH} = 2$$

POLITICS wins!!!

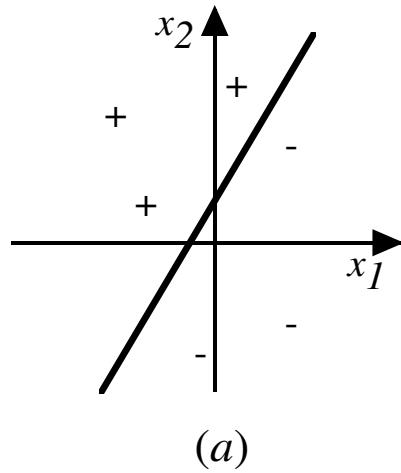
# Perceptron, linear classification

Boolean functions:  $x_i \in \{0,1\}$

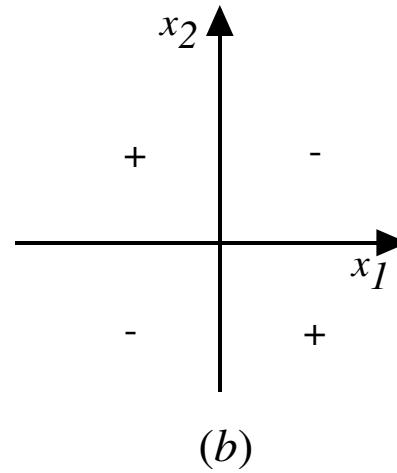
- Can learn  $x_1 \vee x_2$ ?
  - $-0.5 + x_1 + x_2$
- Can learn  $x_1 \wedge x_2$ ?
  - $-1.5 + x_1 + x_2$
- Can learn any conjunction or disjunction?
  - $0.5 + x_1 + \dots + x_n$
  - $(-n+0.5) + x_1 + \dots + x_n$
- Can learn majority?
  - $(-0.5*n) + x_1 + \dots + x_n$
- What are we missing? The dreaded XOR!, etc.



# Decision Surface of a Perceptron



(a)



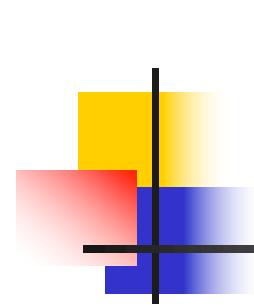
(b)

Represents some useful functions

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

But some functions not representable

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



# Perceptron Training Rule

---

Can prove it will converge

- If training data is linearly separable
- and  $\eta$  sufficiently small

# Going beyond linear classification

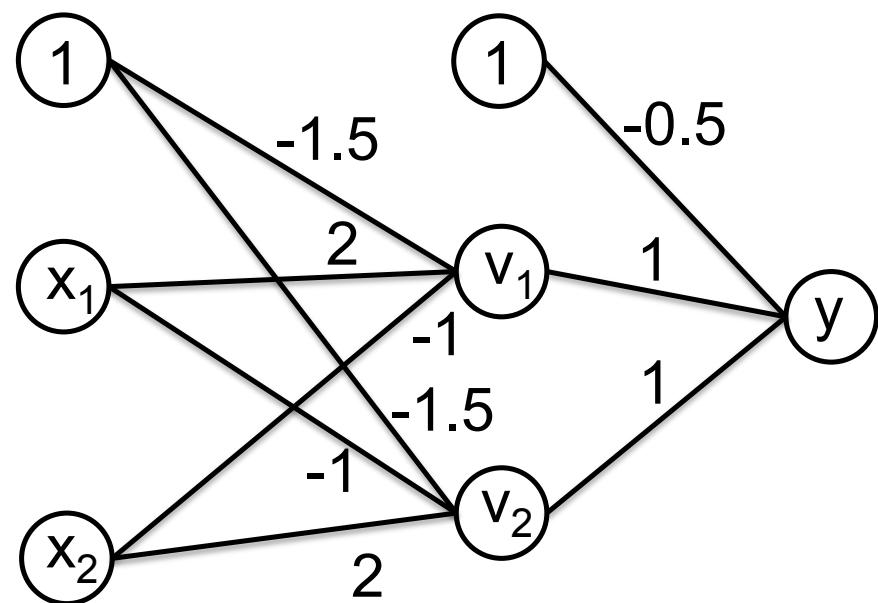
Solving the XOR problem

$$y = x_1 \text{ XOR } x_2 = (x_1 \wedge \neg x_2) \vee (x_2 \wedge \neg x_1)$$

$$\begin{aligned}v_1 &= (x_1 \wedge \neg x_2) \\&= -1.5 + 2x_1 - x_2\end{aligned}$$

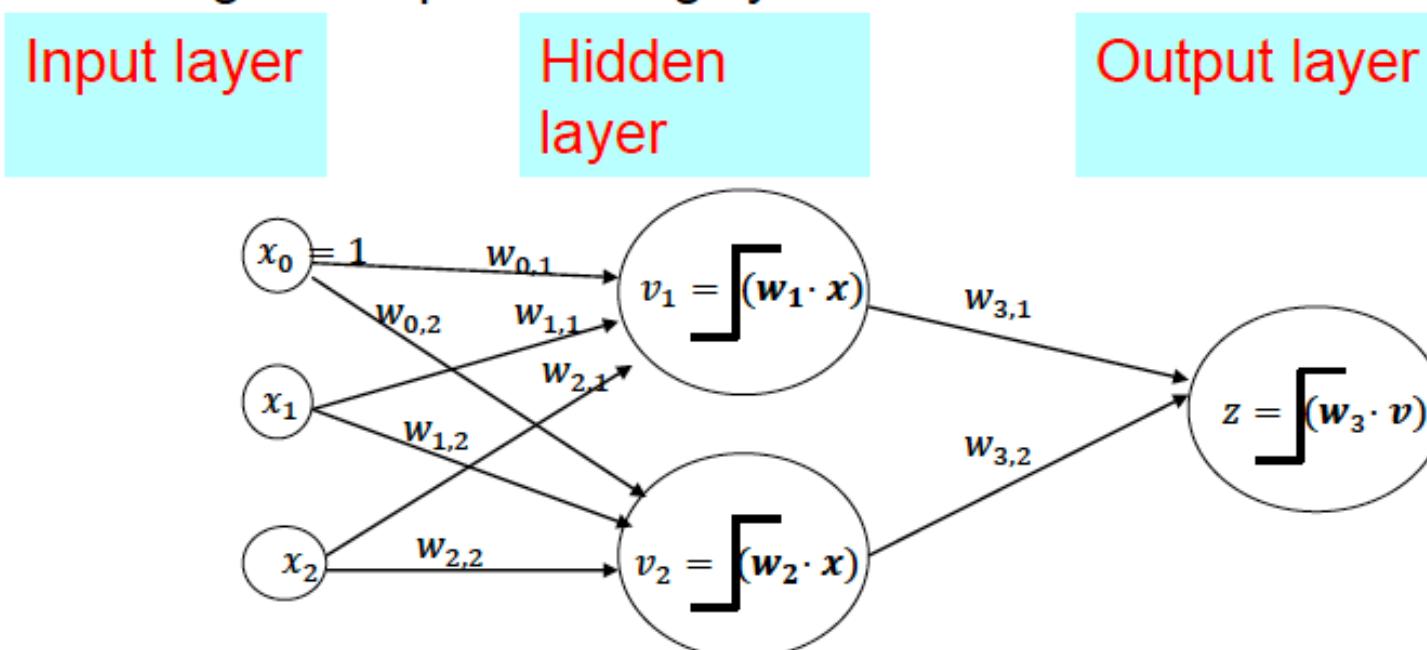
$$\begin{aligned}v_2 &= (x_2 \wedge \neg x_1) \\&= -1.5 + 2x_2 - x_1\end{aligned}$$

$$\begin{aligned}y &= v_1 \vee v_2 \\&= -0.5 + v_1 + v_2\end{aligned}$$



# Multilayer network of Perceptron units

- Advantage: Can produce highly non-linear decision boundaries!

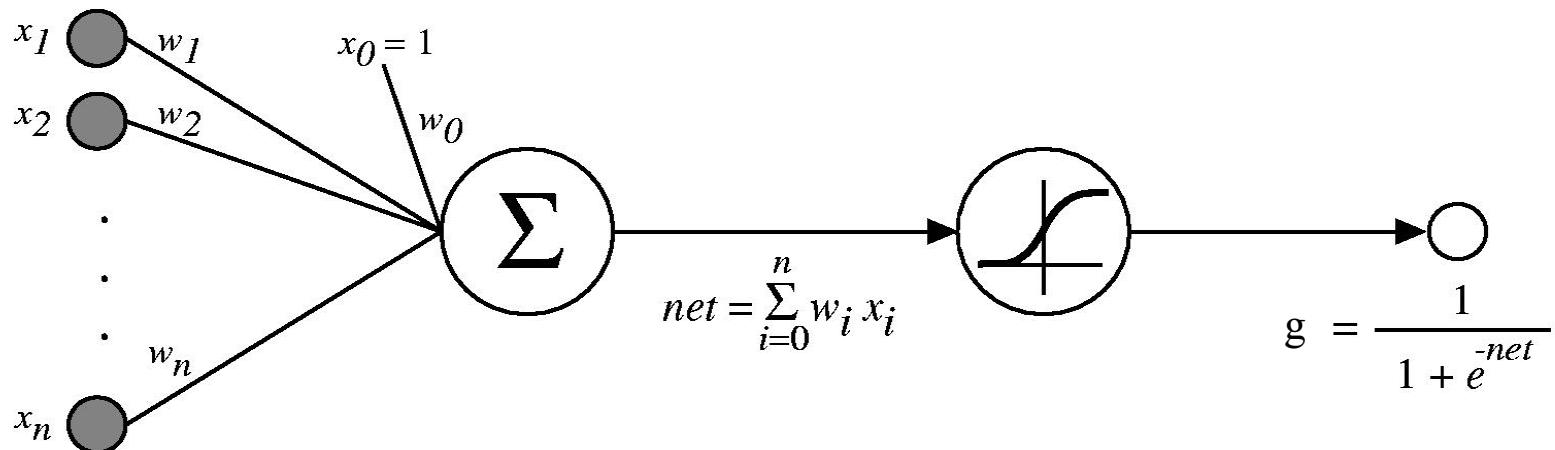
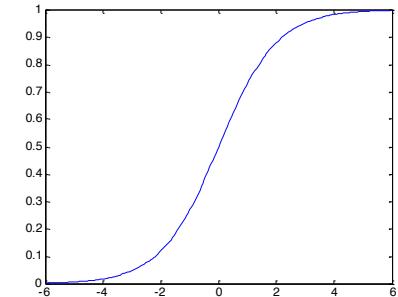


Threshold function:  $\lceil x = 1 \text{ if } x \text{ is positive, } 0 \text{ if } x \text{ is negative.}$

Problem: discontinuous threshold is not differentiable. Can't do gradient descent.

# Sigmoid Neuron

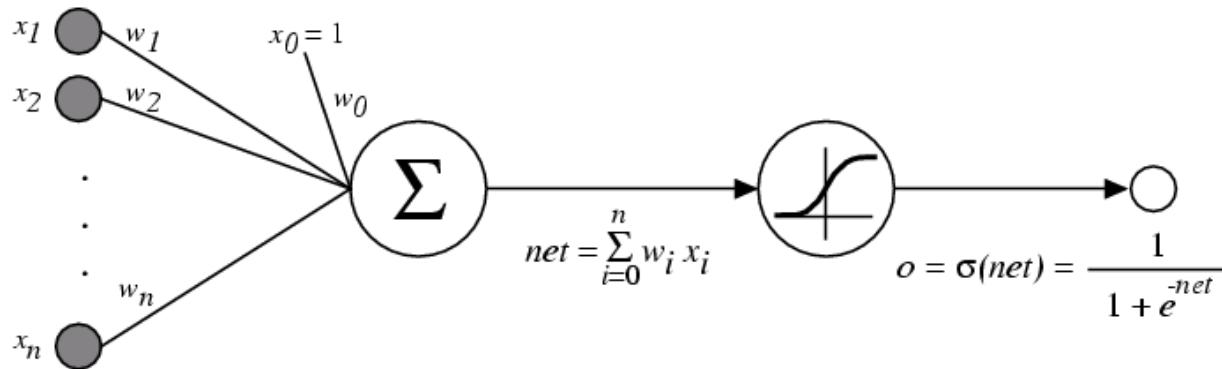
$$g(w_0 + \sum_i w_i x_i) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}}$$



Just change g!

- Why would we want to do this?
- Notice new output range [0,1]. What was it before?
- Look familiar?

# The Sigmoid Unit



$\sigma$  is the sigmoid function;  $\sigma(x) = \frac{1}{1+e^{-x}}$

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

# Optimizing a neuron

$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x))g'(x)$$

We train to minimize sum-squared error

$$\ell(W) = \frac{1}{2} \sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]^2$$

$$\frac{\partial l}{\partial w_i} = - \sum_j [y_j - g(w_0 + \sum_i w_i x_i^j)] \frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x_i^j)$$

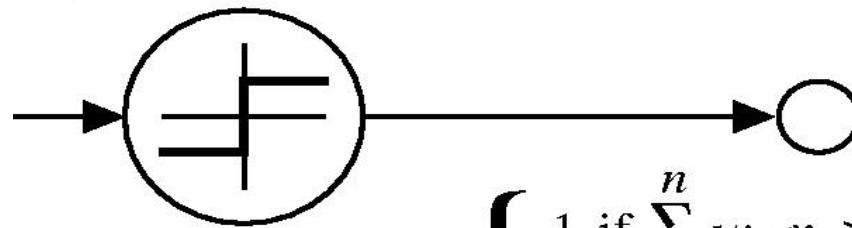
$$\frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x_i^j) = x_i^j \frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x_i^j) = x_i^j g'(w_0 + \sum_i w_i x_i^j)$$

$$\frac{\partial \ell(W)}{\partial w_i} = - \sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)] x_i^j g'(w_0 + \sum_i w_i x_i^j)$$

Solution just depends on  $g'$ : derivative of activation function!

# Re-deriving the perceptron update

$$\frac{\partial \ell(W)}{\partial w_i} = - \sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)] x_i^j g'(w_0 + \sum_i w_i x_i^j)$$



$$g = \begin{cases} 1 & \text{if } \sum_{i=0}^n w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$$

$$\frac{\partial \ell(W)}{\partial w_i} = - \sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)] x_i^j$$

For a specific, incorrect example:

- $w = w + y^*x$  (our familiar update!)

# Sigmoid units: have to differentiate $g$

$$\frac{\partial \ell(W)}{\partial w_i} = - \sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)] x_i^j g'(w_0 + \sum_i w_i x_i^j)$$

$$g(x) = \frac{1}{1 + e^{-x}} \quad g'(x) = g(x)(1 - g(x))$$

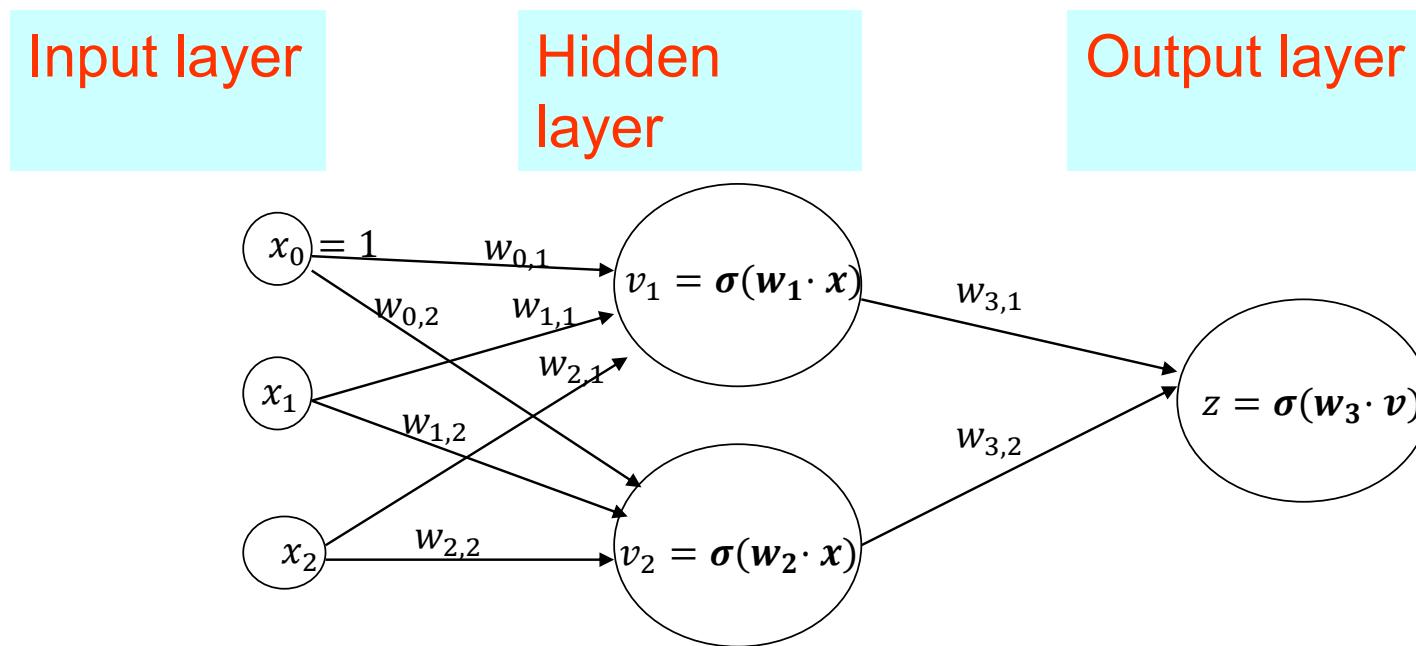
$$w_i \leftarrow w_i + \eta \sum_j x_i^j \delta^j$$

$$\delta^j = [y^j - g(w_0 + \sum_i w_i x_i^j)] g^j (1 - g^j)$$

$$g^j = g(w_0 + \sum_i w_i x_i^j)$$

# Multilayer network of sigmoid units

- Advantage: Can produce highly non-linear decision boundaries!
- Sigmoid is differentiable, so can use gradient descent



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

Very useful in practice!

# Hidden Layer

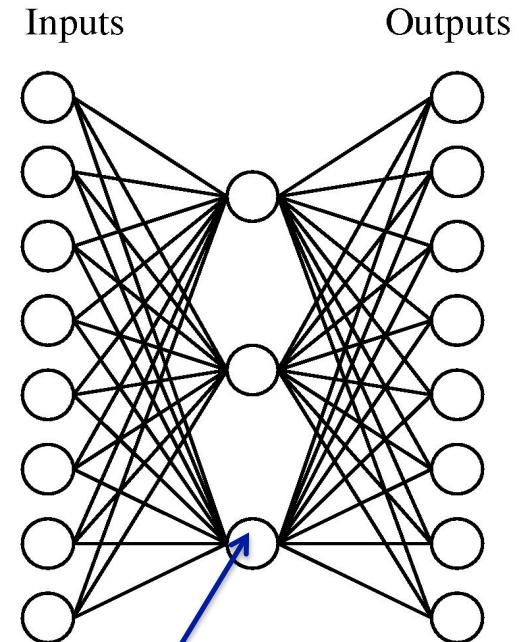
- Single unit:

$$out(\mathbf{x}) = g(w_0 + \sum_i w_i x_i)$$

- 1-hidden layer:

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

- No longer convex function!



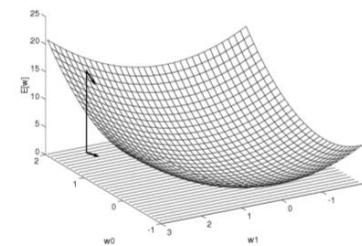
# Gradient Descent to Minimize Squared Error

Goal: Given  $(x_d, t_d)_{d \in D}$  find  $w$  to minimize  $E_D[w] = \frac{1}{2} \sum_{d \in D} (f_w(x_d) - t_d)^2$

## Batch mode Gradient Descent:

Do until satisfied

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



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

## Incremental (stochastic) Gradient Descent:

Do until satisfied

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

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

Note: *Incremental Gradient Descent* can approximate *Batch Gradient Descent* arbitrarily closely if  $\eta$  made small enough

# Gradient Descent in weight space

Goal: Given  $(x_d, t_d)_{d \in D}$  find  $w$  to minimize  $E_D[w] = \frac{1}{2} \sum_{d \in D} (f_w(x_d) - t_d)^2$

This error measure defines a surface over the hypothesis (i.e. weight) space

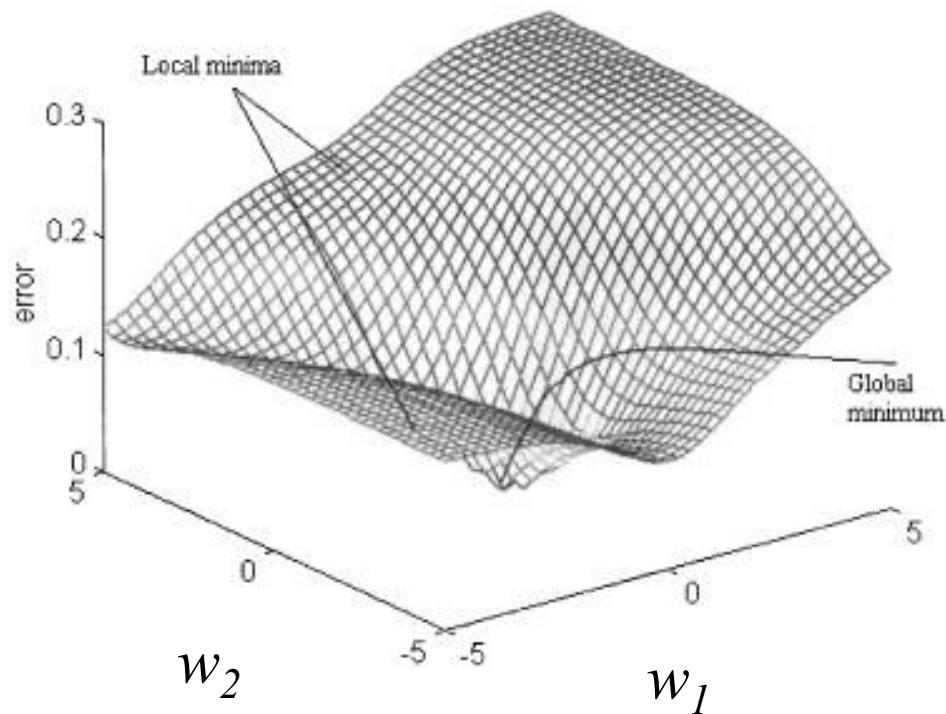


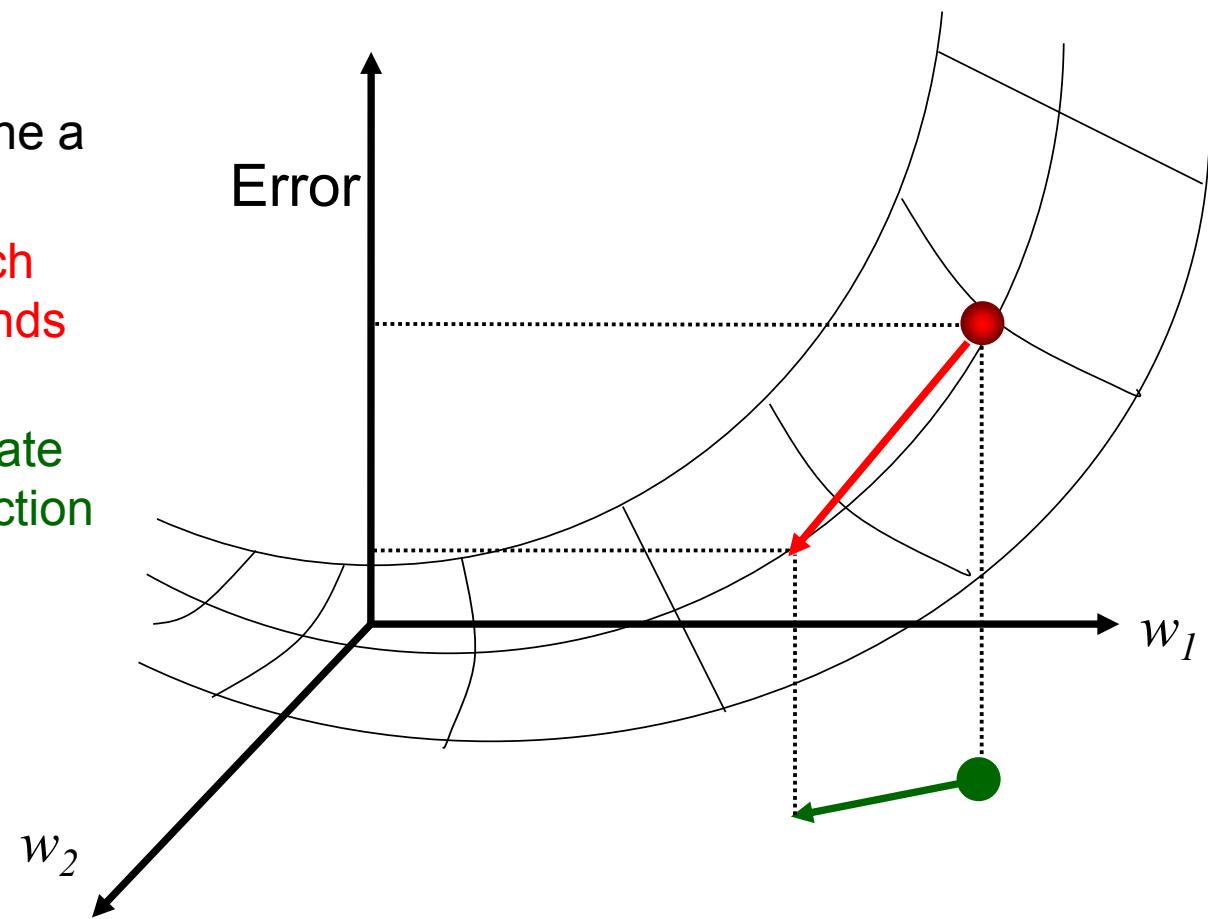
figure from Cho & Chow, *Neurocomputing* 1999

# Gradient Descent in weight space

Gradient descent is an iterative process aimed at finding a minimum in the error surface.

on each iteration

- current weights define a point in this space
- find direction in which error surface descends most steeply
- take a step (i.e. update weights) in that direction



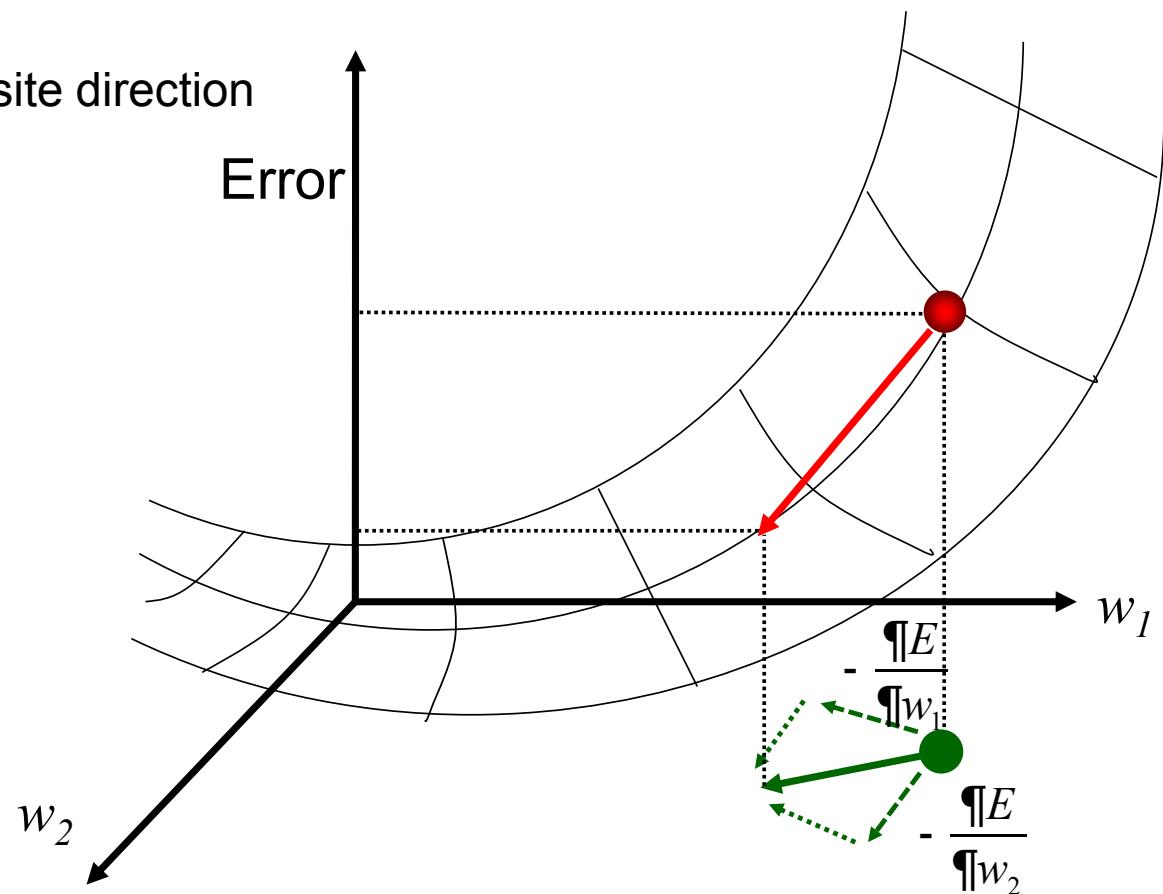
# Gradient Descent in weight space

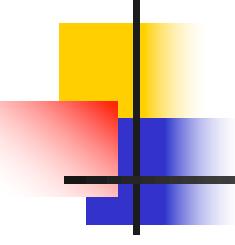
Calculate the gradient of  $E$ :  $\nabla E(\mathbf{w}) = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \square, \frac{\partial E}{\partial w_n} \right]$

Take a step in the opposite direction

$$\Delta \mathbf{w} = -h \nabla E(\mathbf{w})$$

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





## Taking derivative: chain rule

---

Recall the chain rule from calculus

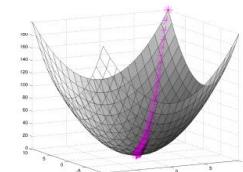
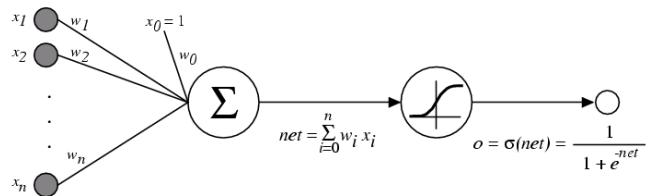
$$y = f(u)$$

$$u = g(x)$$

$$\frac{\frac{dy}{dx}}{u} = \frac{\frac{dy}{du}}{\frac{du}{dx}}$$

# Gradient Descent for the Sigmoid unit

Given  $(x_d, t_d)_{d \in D}$  find  $\mathbf{w}$  to minimize  $\sum_{d \in D} (o_d - t_d)^2$



$o_d$  = observed unit output for  $x_d$

$$o_d = \sigma(\text{net}_d); \quad \text{net}_d = \sum_i w_i x_{i,d}$$

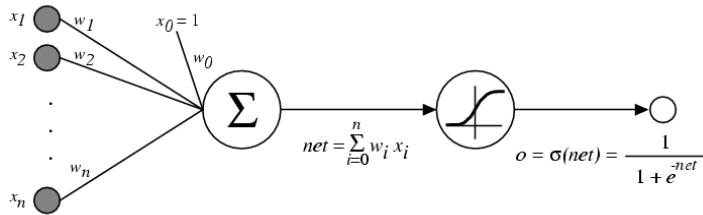
$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 = \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) = \sum_{d \in D}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

But we know:  $\frac{\partial o_d}{\partial \text{net}_d} = \frac{\partial \sigma(\text{net}_d)}{\partial \text{net}_d} = o_d(1 - o_d)$  and  $\frac{\partial \text{net}_d}{\partial w_i} = \frac{\partial (\mathbf{w} \cdot \mathbf{x}_d)}{\partial w_i} = x_{i,d}$

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

# Gradient Descent for the Sigmoid unit

Given  $(x_d, t_d)_{d \in D}$  find  $\mathbf{w}$  to minimize  $\sum_{d \in D} (o_d - t_d)^2$



$o_d$  = observed unit output for  $x_d$

$$o_d = \sigma(\text{net}_d); \quad \text{net}_d = \sum_i w_i x_{i,d}$$

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

$\delta_d$  error term  $t_d - o_d$  multiplied by  $o_d(1 - o_d)$  that comes from the derivative of the sigmoid function

$$\frac{\partial E}{\partial w_i} = - \sum_{d \in D} \delta_d x_{i,d}$$

Update rule:  $w \leftarrow w - \eta \nabla E[w]$

# Gradient Descent for 1-hidden layer

$$\frac{\partial \ell(W)}{\partial w_k}$$

$$\ell(W) = \frac{1}{2} \sum_j [y^j - \text{out}(\mathbf{x}^j)]^2$$

Dropped  $w_0$  to make derivation simpler

$$\text{out}(\mathbf{x}) = g \left( \sum_{k'} w_{k'} g \left( \sum_{i'} w_{i'}^{k'} x_{i'} \right) \right)$$

$$v_k^j = g \left( \sum_{i'} w_{i'}^{k'} x_{i'} \right)$$

$$\frac{\partial \ell(W)}{\partial w_k} = \sum_{j=1}^m -[y^j - \text{out}(\mathbf{x}^j)] \frac{\partial \text{out}(\mathbf{x}^j)}{\partial w_k}$$

$$\text{out}(\mathbf{x}) = g \left( \sum_{k'} w_{k'} v_k^j \right)$$

$$\frac{\partial \text{out}(\mathbf{x})}{\partial w_k} = v_k^j g' \left( \sum_{k'} w_{k'} v_k^j \right)$$



Gradient for last layer same as the single node case, but with hidden nodes  $v$  as input!

# Gradient Descent for 1-hidden layer

$$\frac{\partial \ell(W)}{\partial w_i^k}$$

$$\ell(W) = \frac{1}{2} \sum_j [y^j - \text{out}(\mathbf{x}^j)]^2$$

$$\text{out}(\mathbf{x}) = g \left( \sum_{k'} w_{k'} g \left( \sum_{i'} w_{i'}^{k'} x_{i'} \right) \right)$$

Dropped  $w_0$  to make derivation simpler

$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x))g'(x)$$

$$\frac{\partial \ell(W)}{\partial w_i^k} = \sum_{j=1}^m -[y - \text{out}(\mathbf{x}^j)] \frac{\partial \text{out}(\mathbf{x}^j)}{\partial w_i^k}$$

$$\frac{\partial \text{out}(\mathbf{x})}{\partial w_i^k} = g' \left( \sum_{k'} w_{k'} g \left( \sum_{i'} w_{i'}^{k'} x_{i'} \right) \right) \frac{\partial}{\partial w_i^k} g \left( \sum_{i'} w_{i'}^{k'} x_{i'} \right)$$

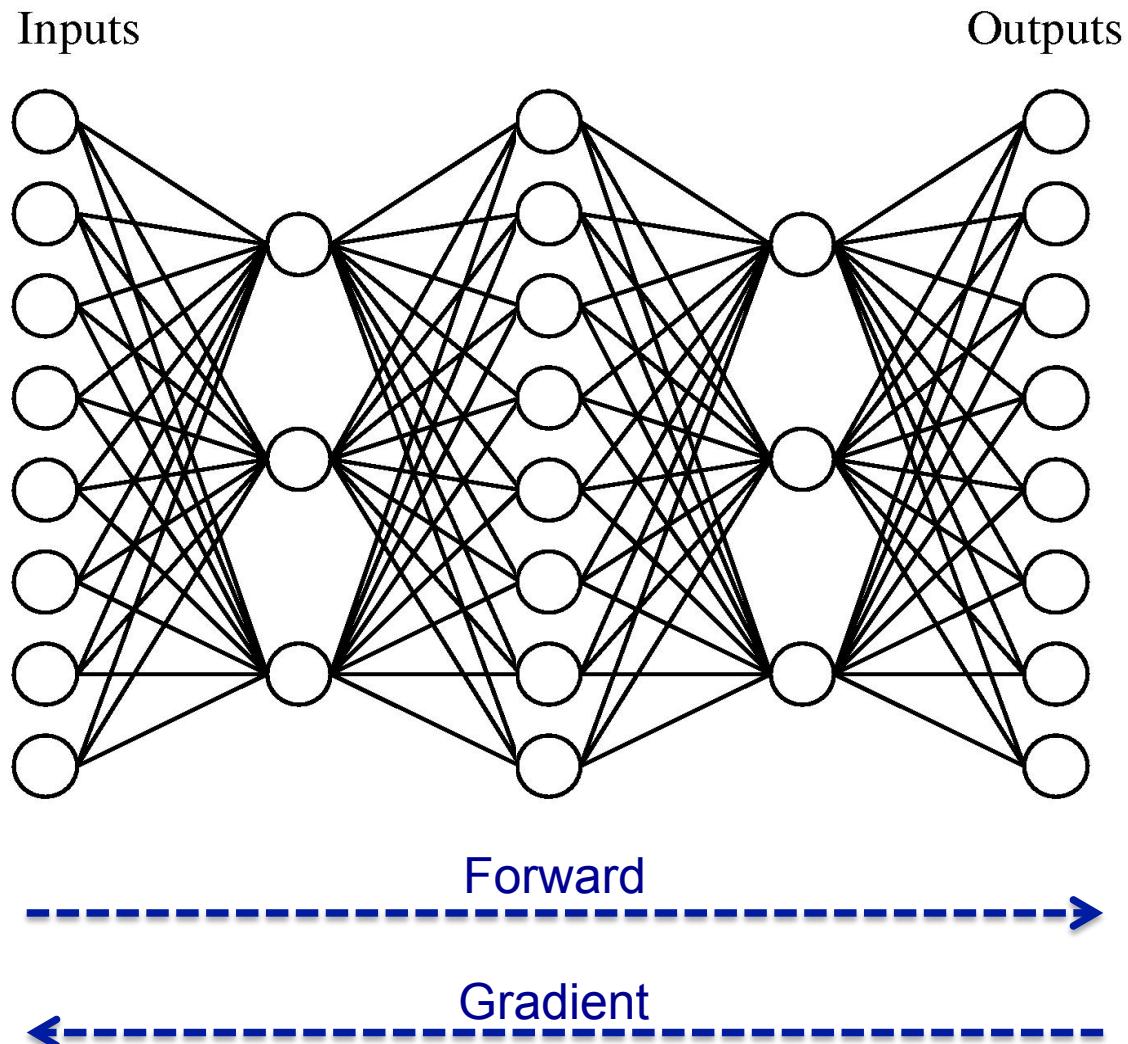
For hidden layer,  
two parts:

- Normal update for single neuron
- Recursive computation of gradient on output layer

# Multilayer Neural Networks

Inference and Learning:

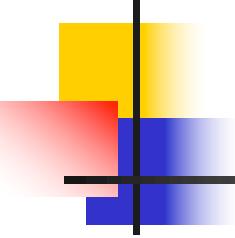
- Forward pass: left to right, each hidden layer in turn
- Gradient computation: right to left, propagating gradient for each node



# Forward propagation: prediction

- Recursive algorithm
- Start from input layer
- Output of node  $V_k$  with parents  $U_1, U_2, \dots$ :

$$V_k = g \left( \sum_i w_i^k U_i \right)$$



# Back-propagation - learning

- Just gradient descent!!!
- Recursive algorithm for computing gradient
- For each example
  - Perform forward propagation
  - Start from output layer
    - Compute gradient of node  $V_k$  with parents  $U_1, U_2, \dots$
    - Update weight  $w_i^k$
    - Repeat (move to preceding layer)

# Backpropagation Algorithm

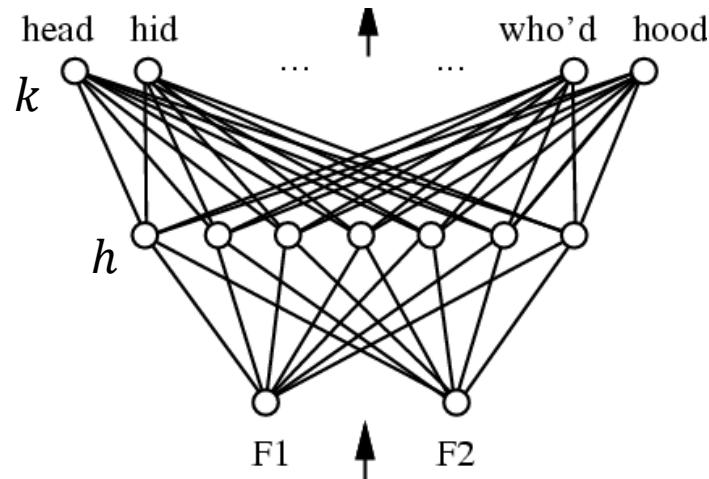
## Incremental/stochastic gradient descent

Initialize all weights to small random numbers.

**Until satisfied, Do:**

- **For each training example  $(x, t)$  do:**
  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  $h$ :
$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$$
  4. Update each network weight  $w_{i,j}$ 
$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

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



$o$  = observed unit output

$t$  = target output

$x$  = input

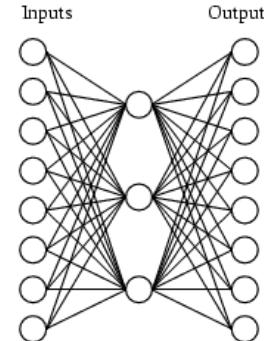
$x_{i,j}$  =  $i$ th input to  $j$ th unit

$w_{ij}$  = wt from  $i$  to  $j$

# Example

## Example data for NN with hidden layer

A target function:



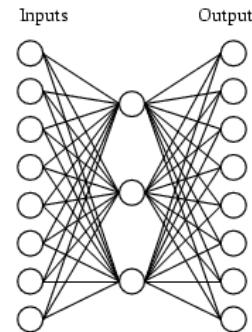
Input	Output
10000000	→ 10000000
01000000	→ 01000000
00100000	→ 00100000
00010000	→ 00010000
00001000	→ 00001000
00000100	→ 00000100
00000010	→ 00000010
00000001	→ 00000001

Can this be learned??

# Example

## Learned weights for hidden layer

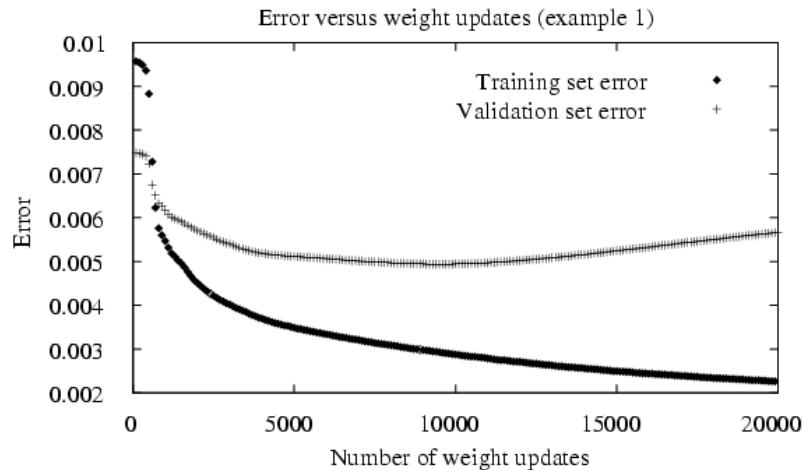
A network:



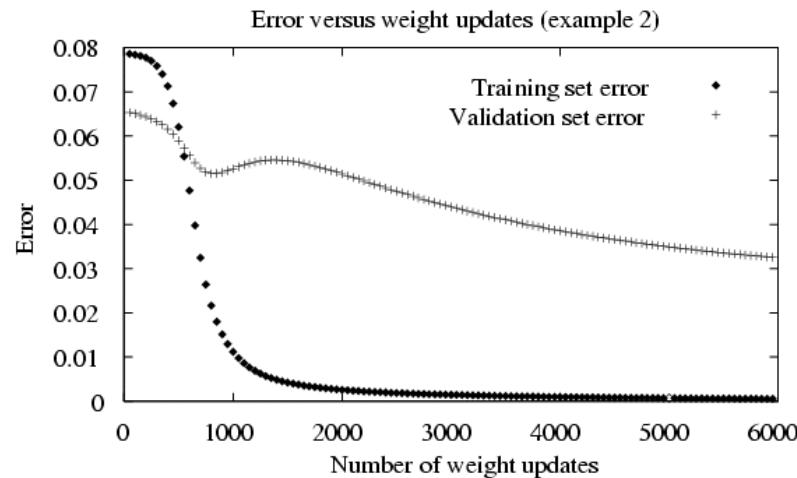
Learned hidden layer representation:

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

# Overfitting in ANN

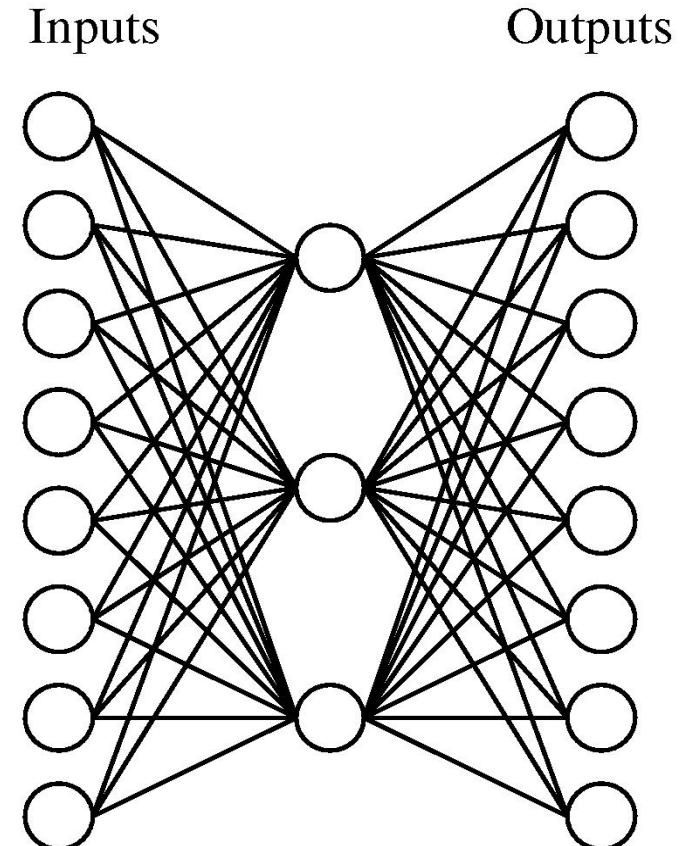


- Validation/generalization error first decreases, then increases.
- Weights tuned to fit the idiosyncrasies of the training examples that are not representative of the general distribution.
- Stop when lowest error over validation set.
- Not always obvious when lowest error over validation set has been reached.



# Overfitting in ANN

- Are NNs likely to overfit?
  - Yes, they can represent arbitrary functions!!!
- Avoiding overfitting?
  - More training data
  - Fewer hidden nodes / better topology
  - Regularization
  - Early stopping



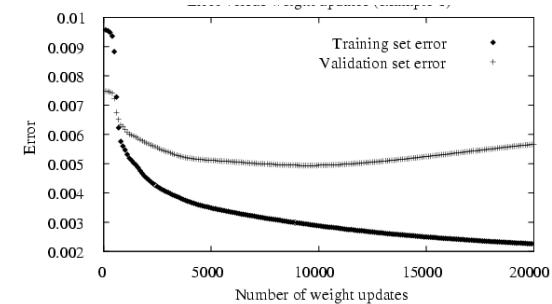
# Dealing with Overfitting

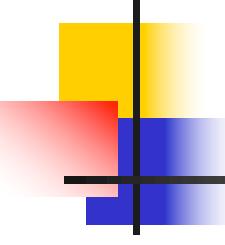
Our learning algorithm involves a parameter

$n$ =number of gradient descent iterations

How do we choose  $n$  to optimize future error?

- Separate available data into training and validation set
- Use training to perform gradient descent
- $n \leftarrow$  number of iterations that optimizes validation set error

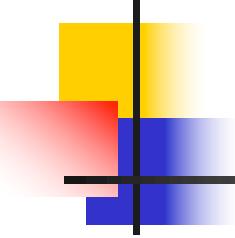




# Dealing with Overfitting

---

- Regularization techniques
  - norm constraint
  - dropout
  - batch normalization
  - data augmentation
  - early stopping
  - ...



# Expressive Capabilities of ANNs

Boolean functions:

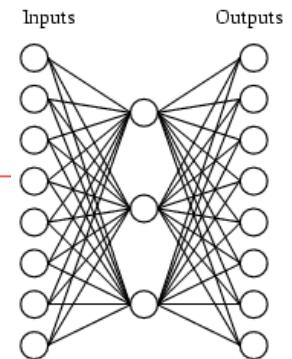
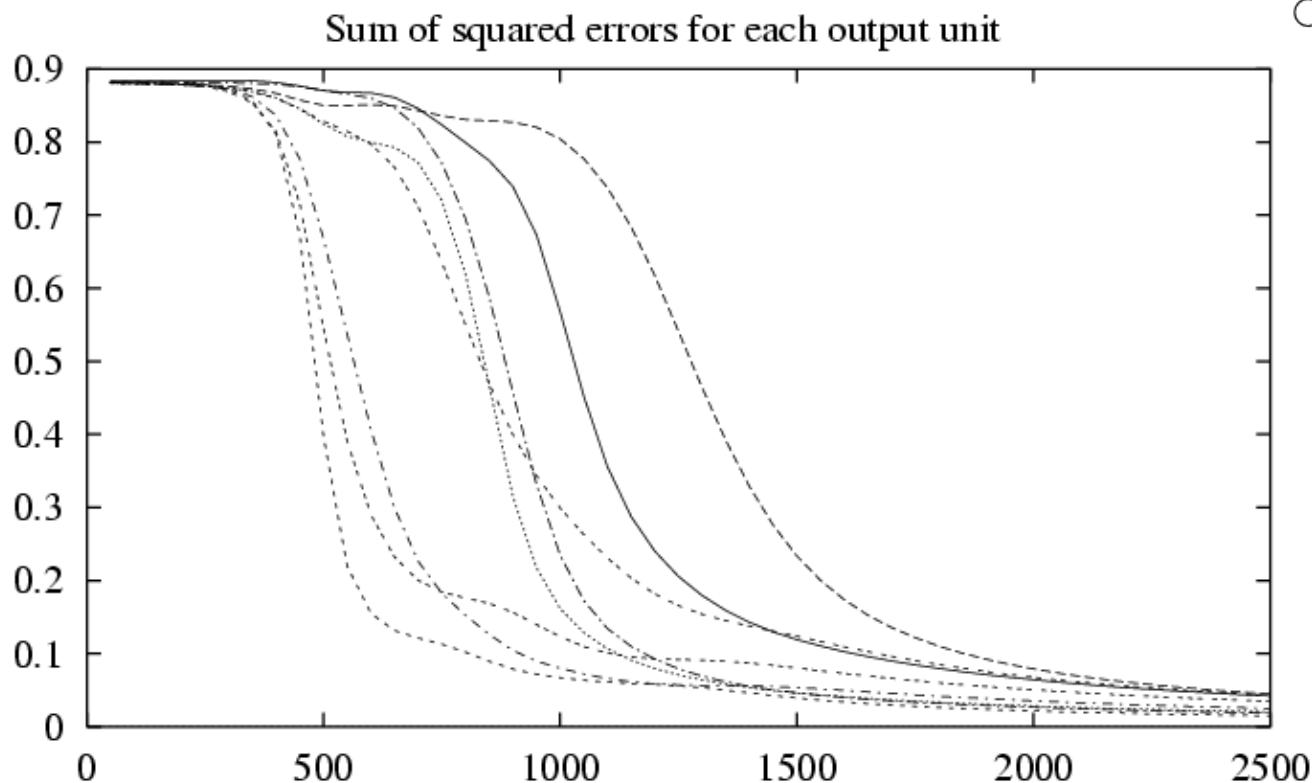
- Every Boolean function can be represented by a network with a single hidden layer
- But might require exponential (in number of inputs) hidden units

Continuous functions:

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

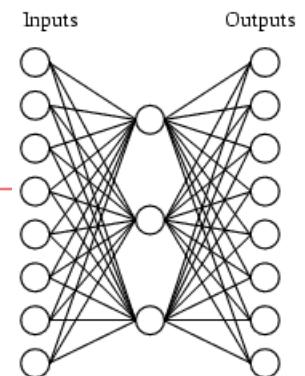
# Training

Training

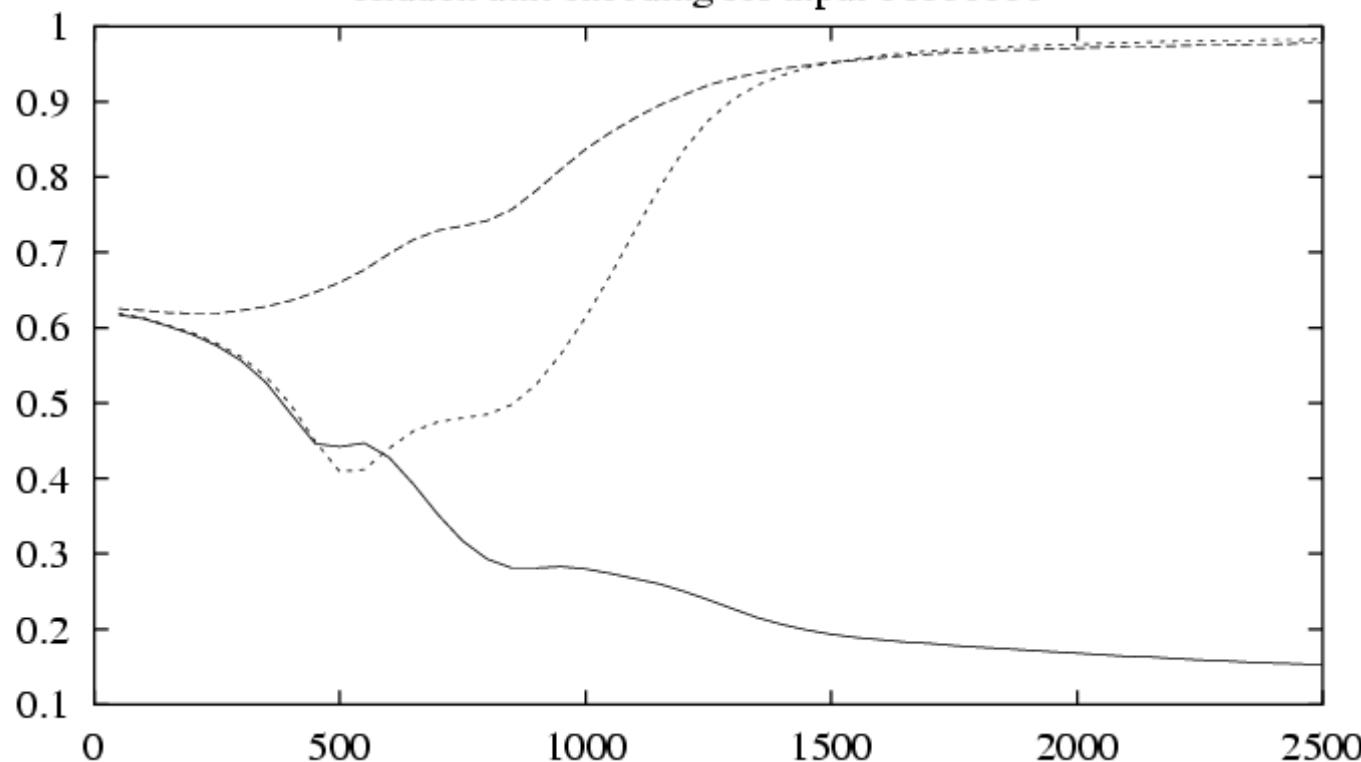


# Training

Training

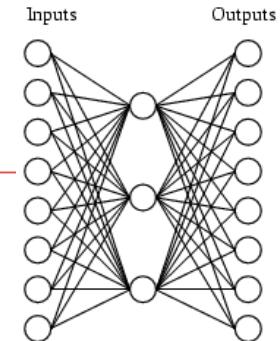


Hidden unit encoding for input 01000000

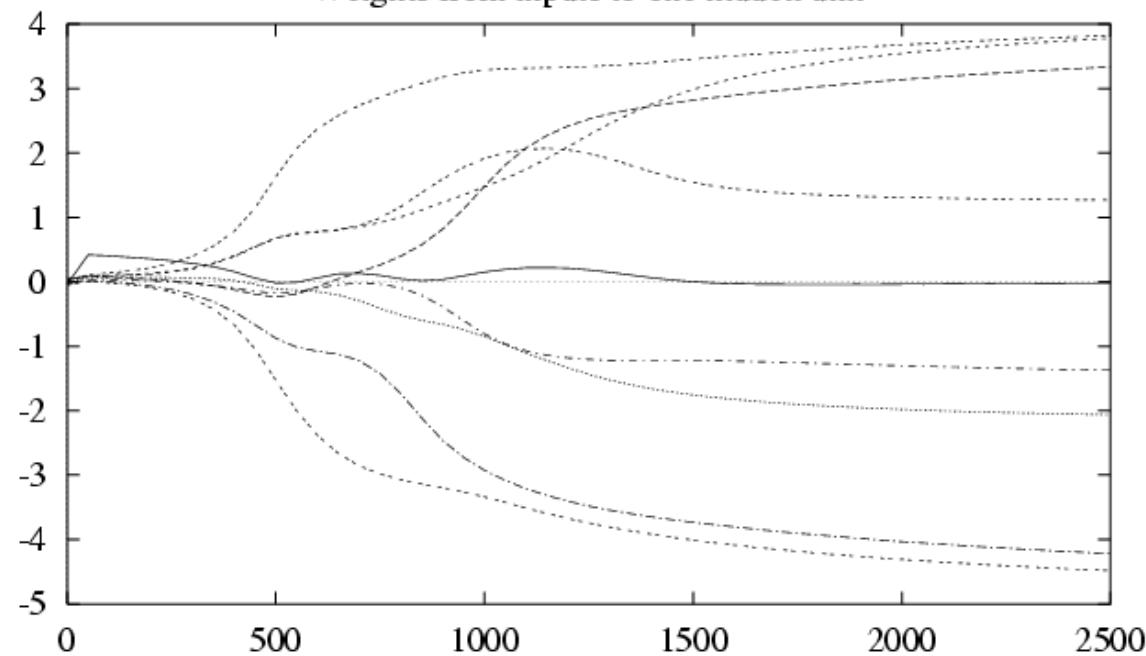


# Training

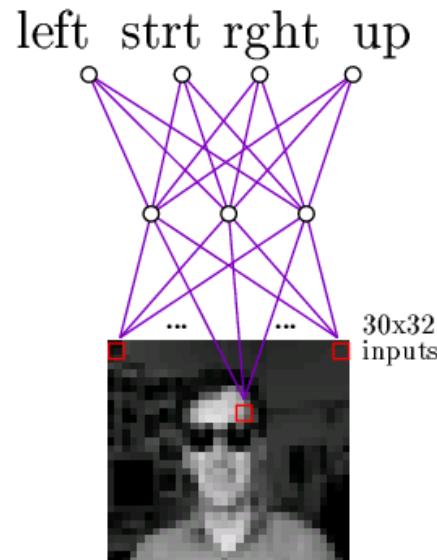
Training



Weights from inputs to one hidden unit



# Neural Nets for Face Recognition

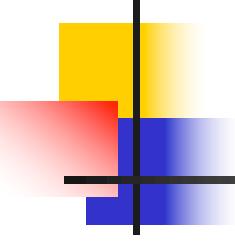


<http://www.cs.cmu.edu/~tom/faces.html>



Typical input images

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



# Artificial Neural Networks: Summary

- Highly non-linear regression/classification
  - Vector valued inputs and outputs
  - Potentially millions of parameters to estimate
  - Actively used to model distributed computation in the brain
  - Hidden layers learn intermediate representations
- 
- Stochastic gradient descent, local minima problems
  - Overfitting and how to deal with it.