CSCI 5090/7090- Machine Learning

Spring 2018

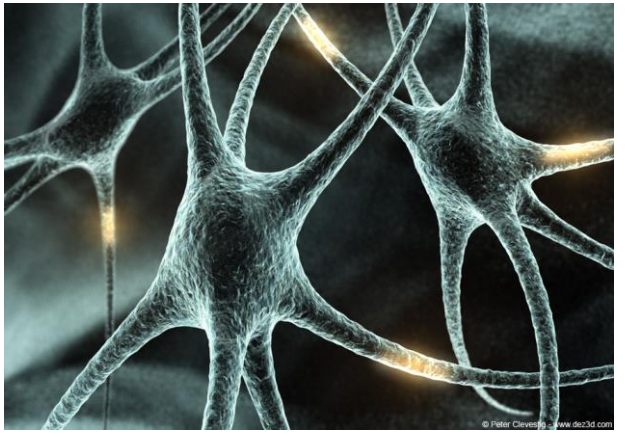
Mehdi Allahyari
Georgia Southern University

Neural Networks

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

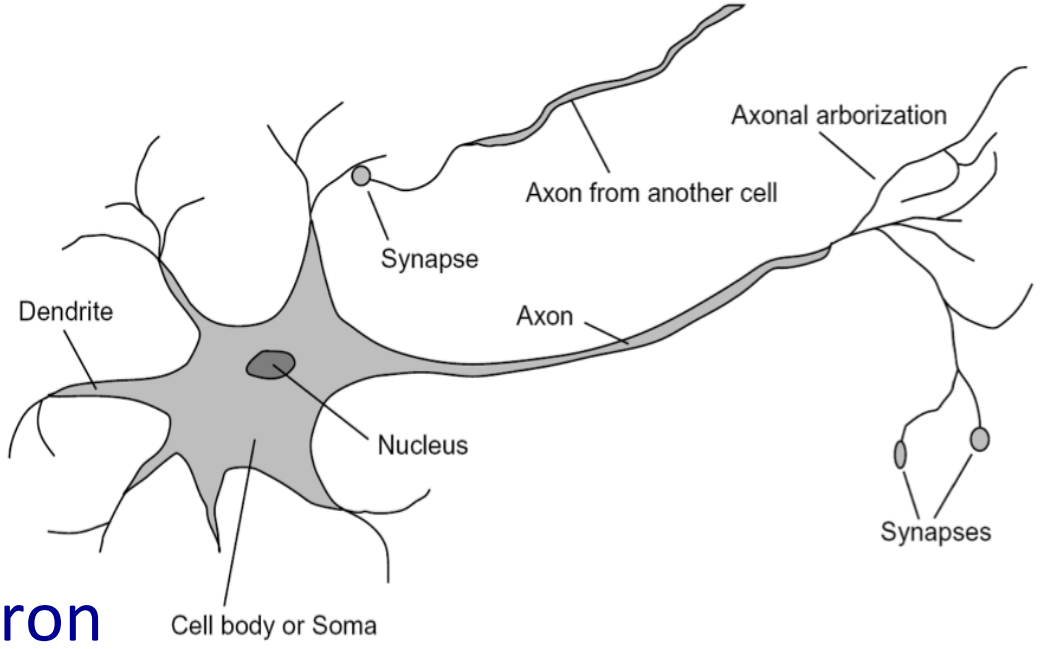
Artificial Neural Network (ANN)

- 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
  - ~ 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

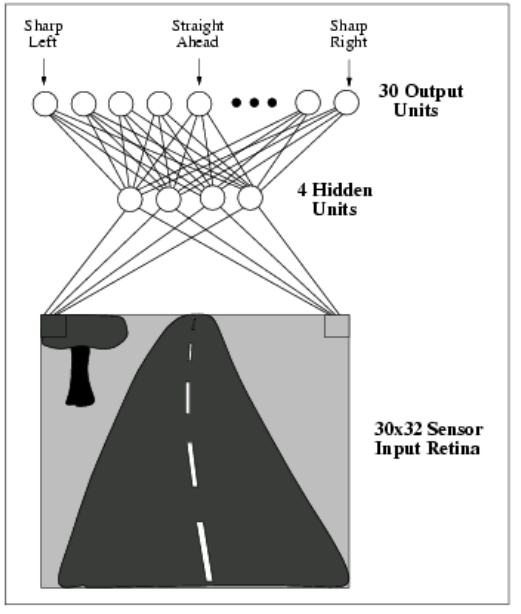
Examples:

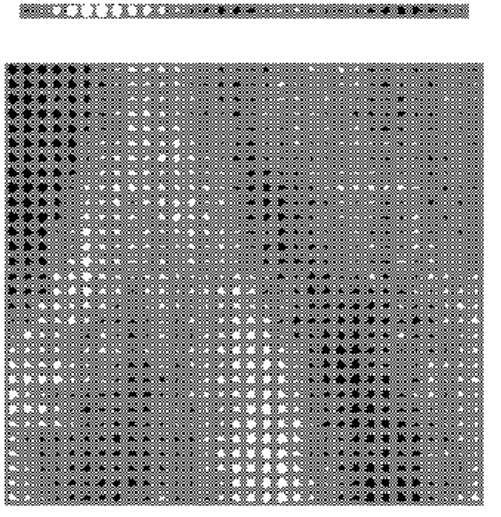
- Speech phoneme recognition [Waibel]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction

Example

ALVINN

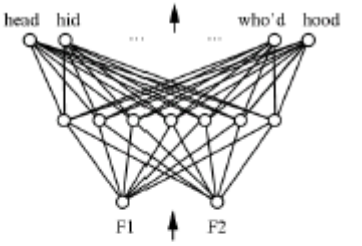
[Pomerleau 1993]





Artificial Neural Networks to learn f: X → Y

- f_w typically a non-linear function, f_w: X → Y



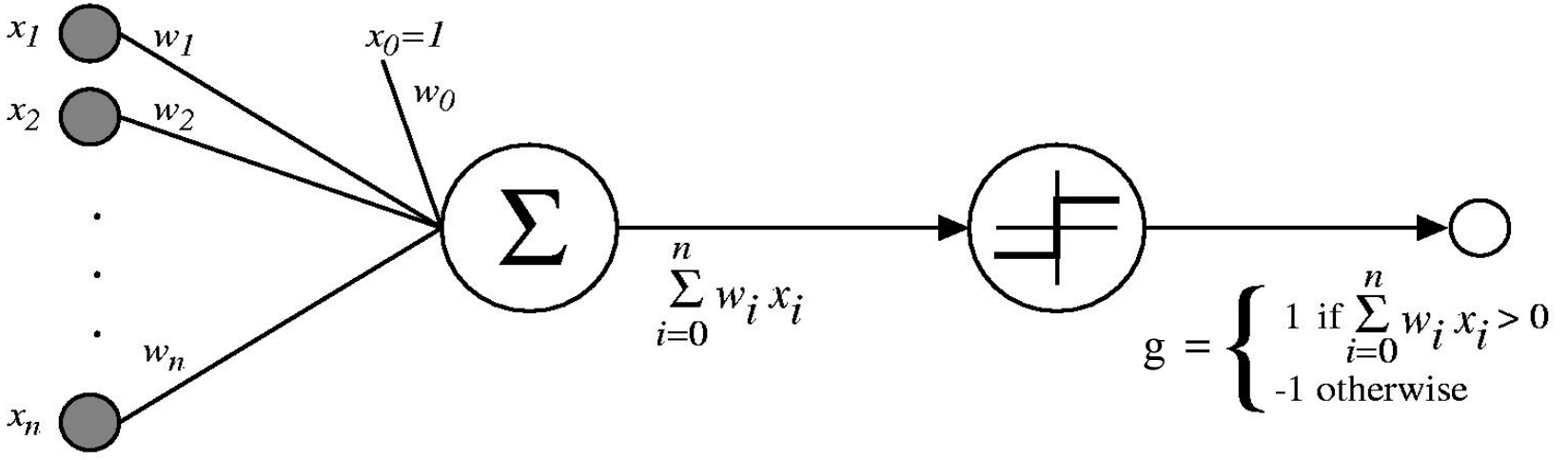
- X feature space: (vector of) continuous and/or discrete vars
- Y ouput space: (vector of) continuous and/or discrete vars
- f_w *network* of basic units

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!

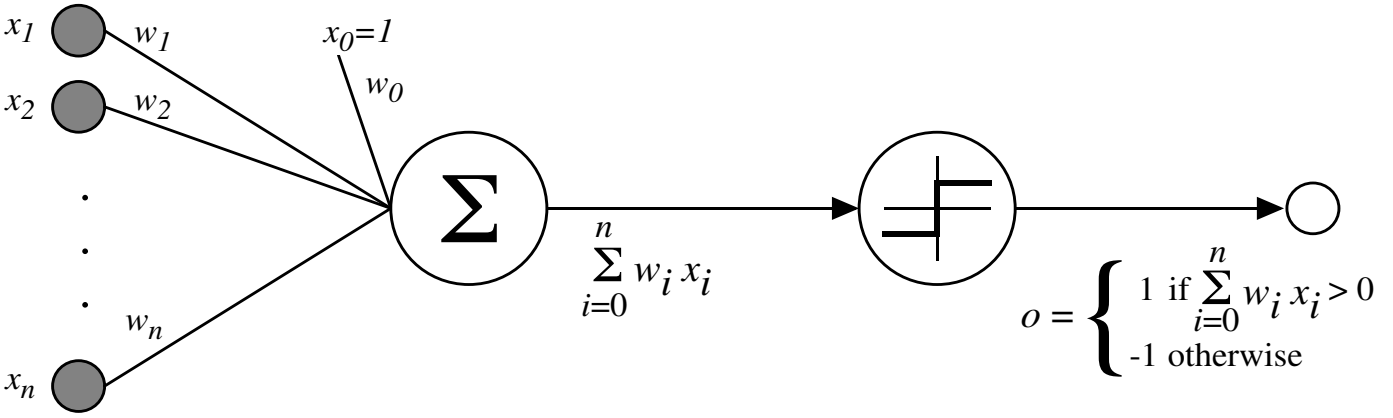
Perceptron as a Neural Network



This is one neuron:

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

Perceptron



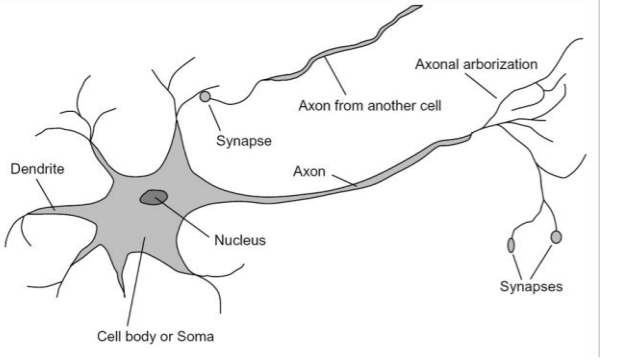
$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

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

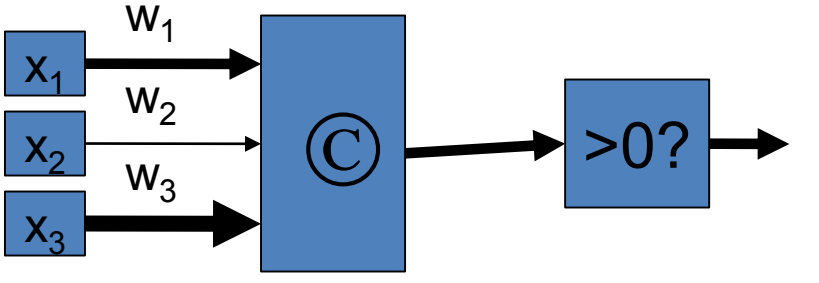
Linear Classifiers

- 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"

x

BIAS	:	1
free	:	1
money	:	1
...		

w

BIAS	:	-3
free	:	4
money	:	2
...		

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

w•x > 0 ➔ 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

```
BIAS  :  1
free  :  1
money :  1
...
```

w

```
BIAS  : -3
free  :  4
money :  2
...
```

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

w•x > 0 ➔ 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
- η 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

$$y = sign(w \cdot x^i)$$

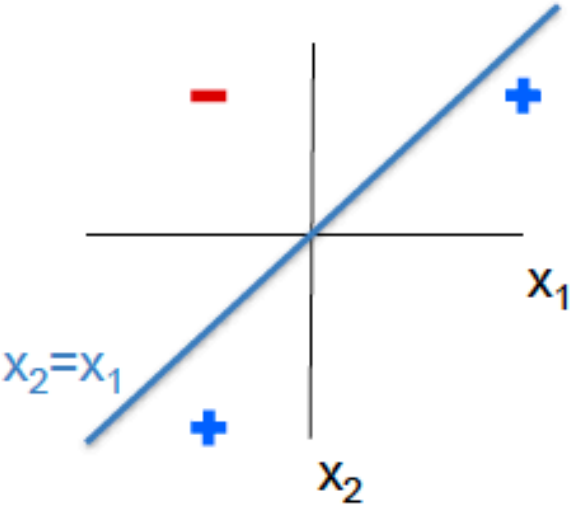
      - sign(x) is +1 if x>0, else -1
    - If correct (i.e., y=y^i), no change!
    - If wrong: update

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

- For t=1..T, i=1..n:
  - $y = sign(w \cdot x^i)$
  - 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]•[3,2] = 0, sign(0)=-1
- w = [0,0] + [3,2] = [3,2]

t=1,i=2

- [3,2]•[-2,2]=-2, sign(-2)=-1

t=1,i=3

- [3,2]•[-2,-3]=-12, sign(-12)=-1
- w = [3,2] + [-2,-3] = [1,-1]

t=2,i=1

- [1,-1]•[3,2]=1, sign(1)=1

t=2,i=2

- [1,-1]•[-2,2]=-4, sign(-4)=-1

t=2,i=3

- [1,-1]•[-2,-3]=1, sign(1)=1

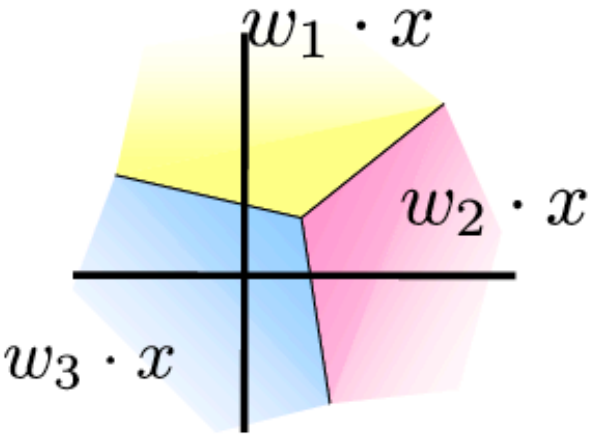
Converged!!!

- $y=w_1x_1+w_2x_2 \rightarrow y=x_1+-x_2$
- So, at $y=0 \rightarrow x_2=x_1$

Multiclass Decision Rule

Example: y is {1,2,3}

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



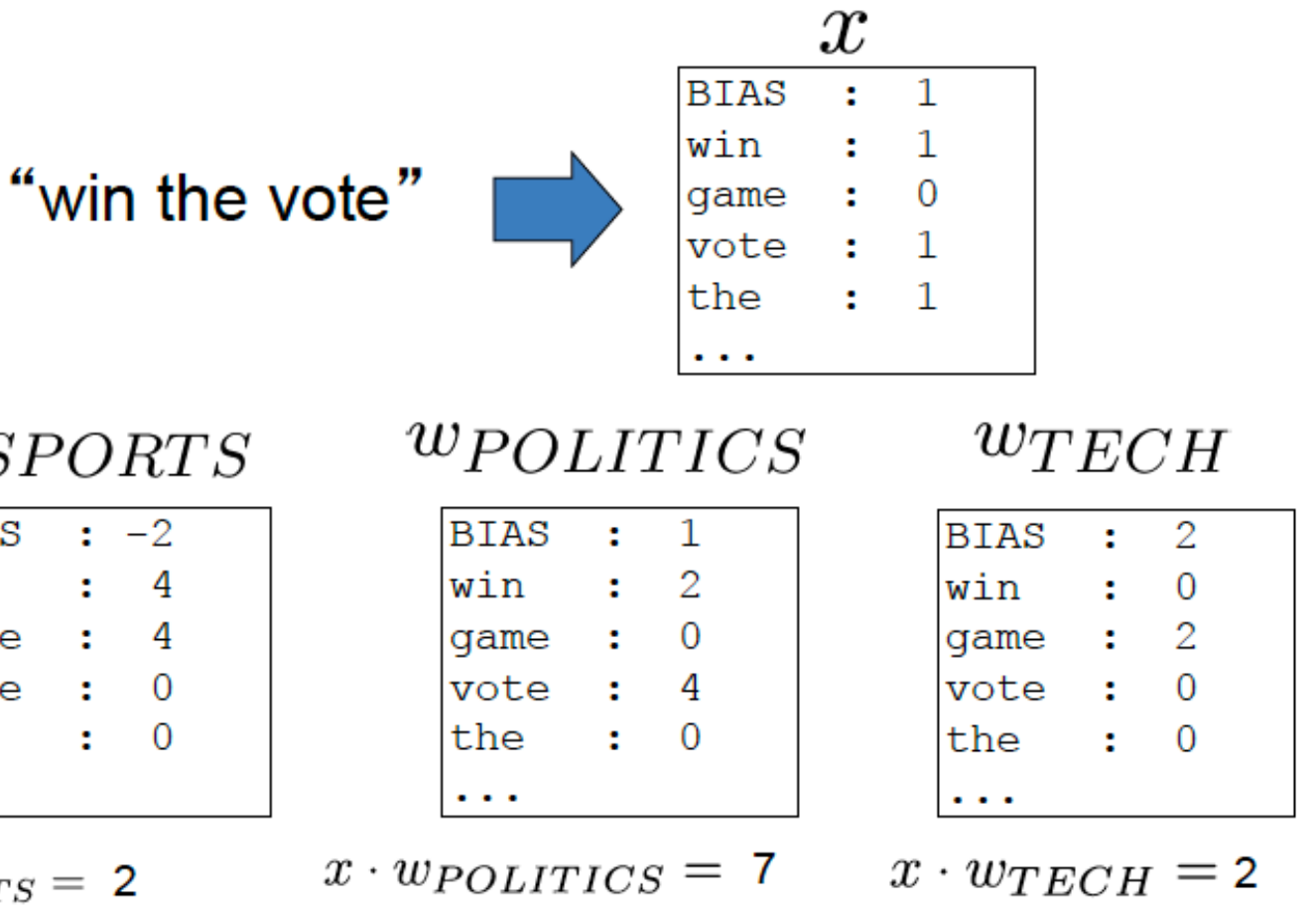
- If we have more than two classes:
  - Have a weight vector for each class: w_y
  - Calculate an activation for each class

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

  - Highest activation wins

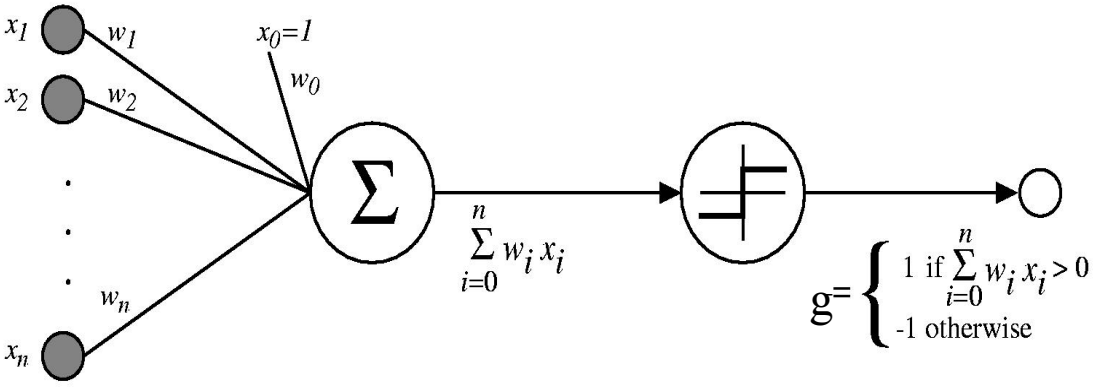
$$y^* = \arg\max_y (\text{activation}_w(x, y))$$

Example


"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
...
$w_{POLITICS}$
BIAS : 1
win : 2
game : 0
vote : 4
the : 0
...
w_{TECH}
BIAS : 2
win : 0
game : 2
vote : 0
the : 0
...
$x \cdot w_{SPORTS} = 2$
$x \cdot w_{POLITICS} = 7$
$x \cdot w_{TECH} = 2$
POLITICS wins!!!

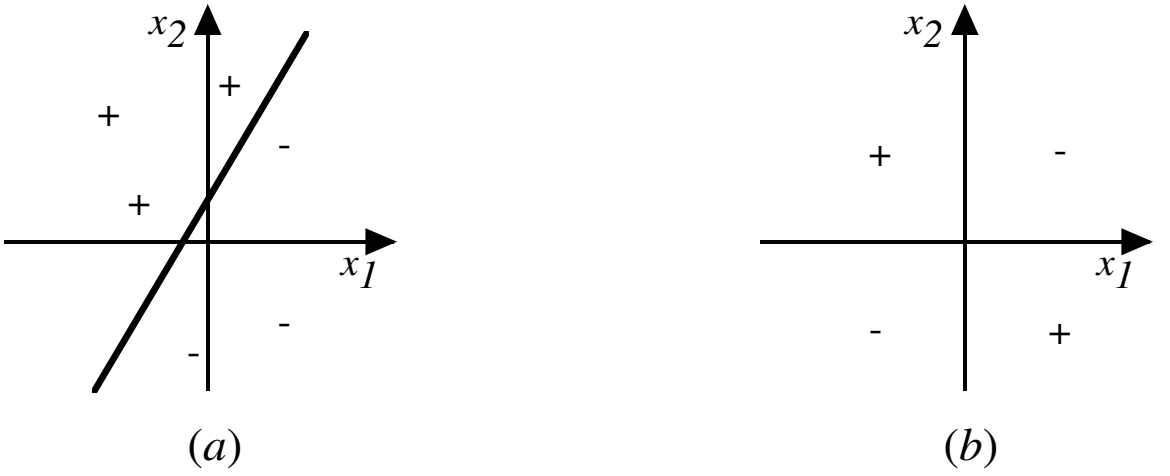
Perceptron, linear classification

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?
  - $0.5 + x_1 + \ldots + x_n$
  - $(-n+0.5) + x_1 + \ldots + x_n$
- Can learn majority?
  - $(-0.5*n) + x_1 + \ldots + x_n$
- What are we missing? The dreaded XOR!, etc.

Decision Surface of a Perceptron

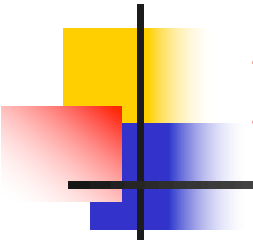


Represents some useful functions

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

But some functions not representable

- 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 η sufficiently small

Going beyond linear classification

Solving the XOR problem

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

$v_1 = (x_1 \wedge \neg x_2)$

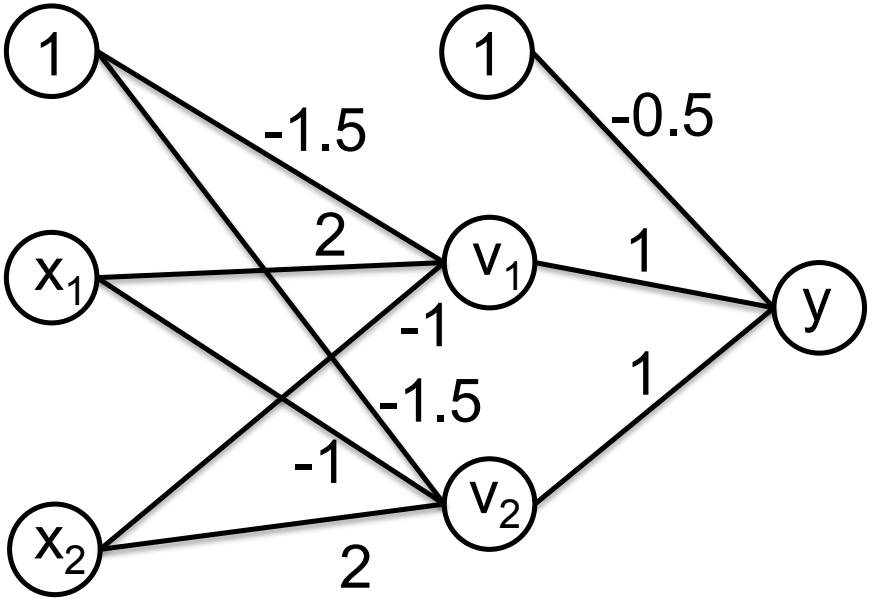
$= -1.5+2x_1-x_2$

$v_2 = (x_2 \wedge \neg x_1)$

$= -1.5+2x_2-x_1$

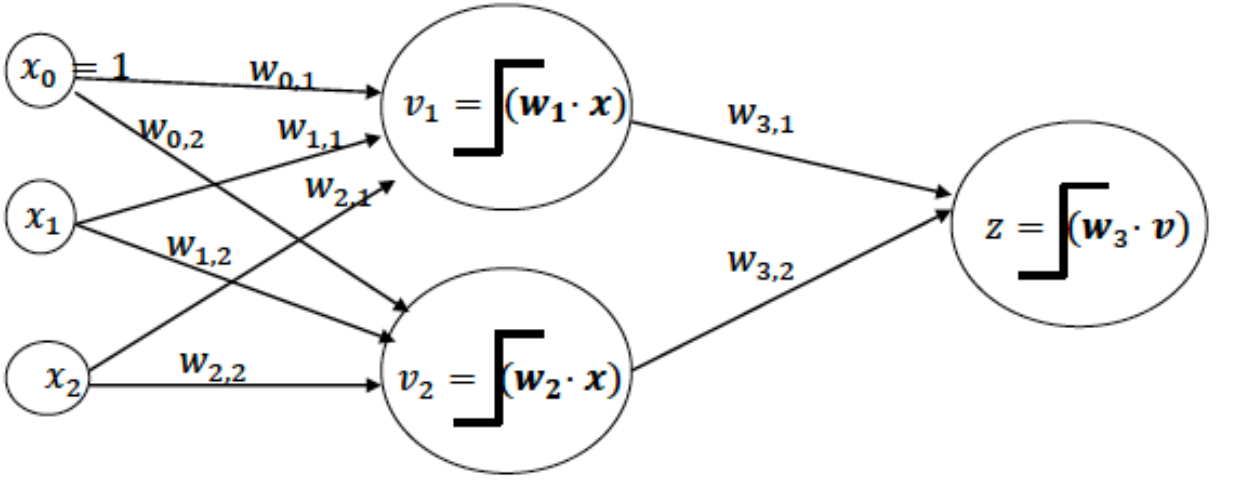
$y = v_1 \vee v_2$

$= -0.5+v_1+v_2$



Multilayer network of Perceptron units

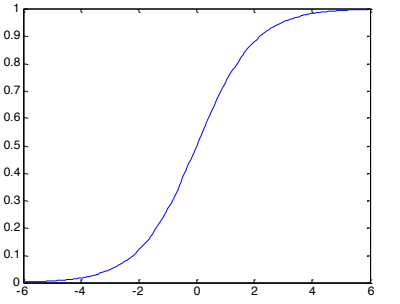
- Advantage: Can produce highly non-linear decision boundaries!



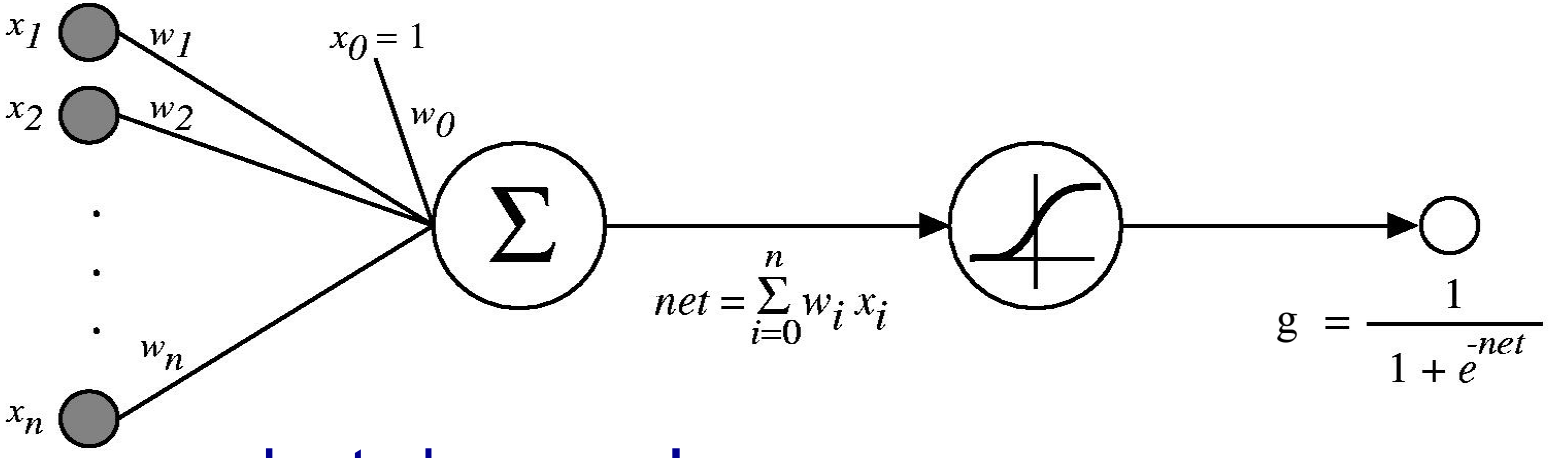
Threshold function: $\sqrt{x} = 1$ if x is positive, 0 if x is negative.

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

Sigmoid Neuron



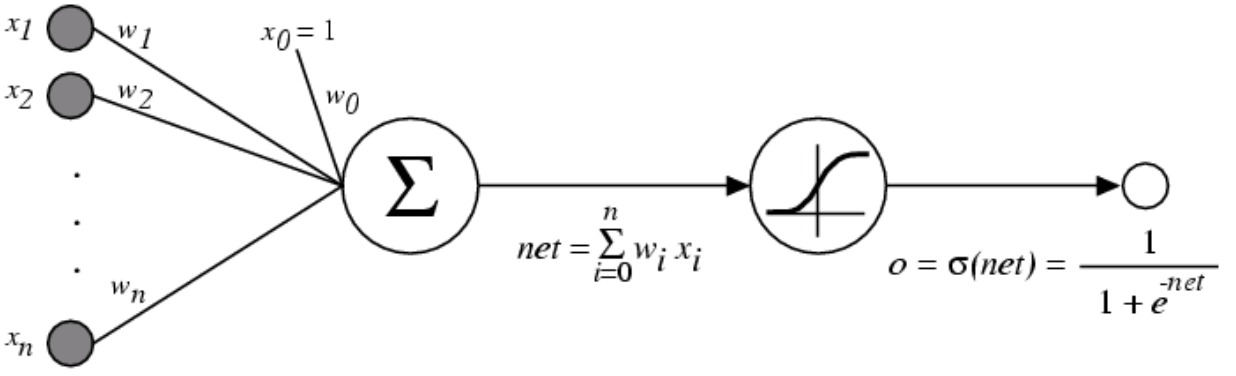
$$g(w_0 + \sum_i w_i x_i) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}}$$



Just change g!

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

The Sigmoid Unit



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

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

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

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

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

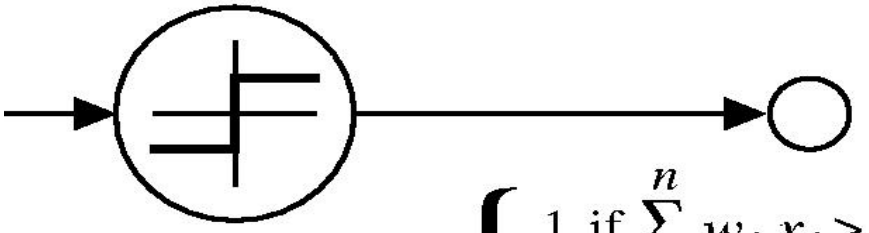
$$\frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x_i^j) = x_i^j \frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x_i^j) = x_i^j g'(w_0 + \sum_i w_i x_i^j)$$

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

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

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

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

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

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

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

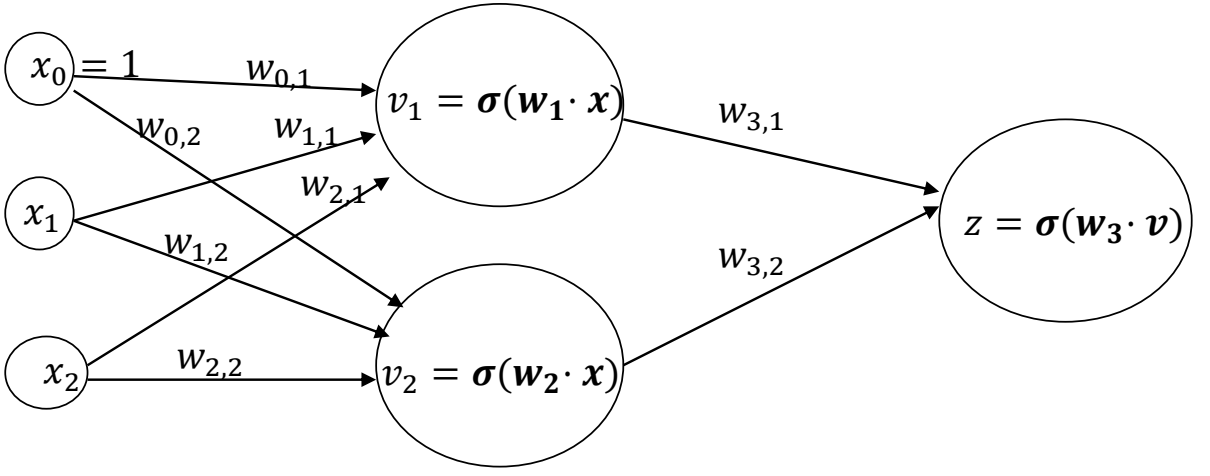
Multilayer network of sigmoid units

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

Input layer

Hidden layer

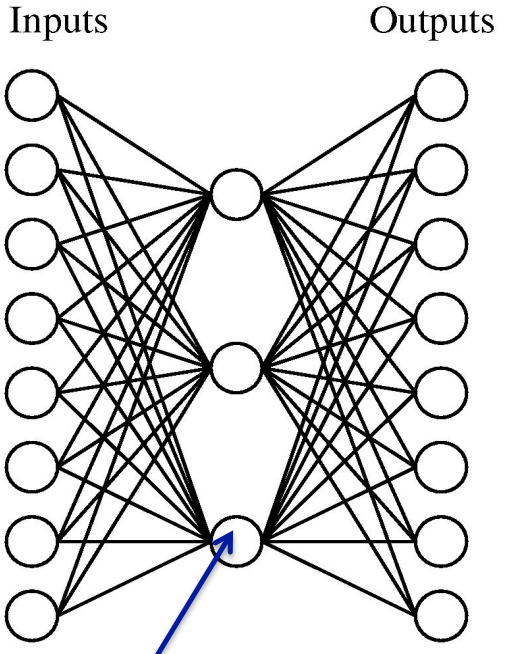
Output layer



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

Very useful in practice!

Hidden Layer



- Single unit:

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

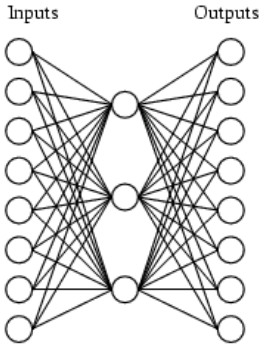
- 1-hidden layer:

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

- No longer convex function!

Example

Example data for NN with hidden layer



A target function:

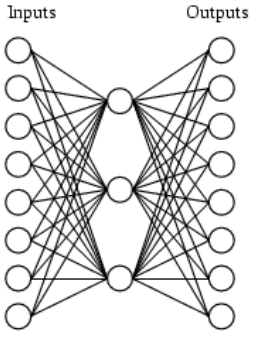
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

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

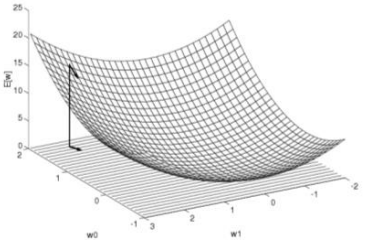
Gradient Descent to Minimize Squared Error

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

Batch mode Gradient Descent:

Do until satisfied

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



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

Incremental (stochastic) Gradient Descent:

Do until satisfied

- For each training example d in D

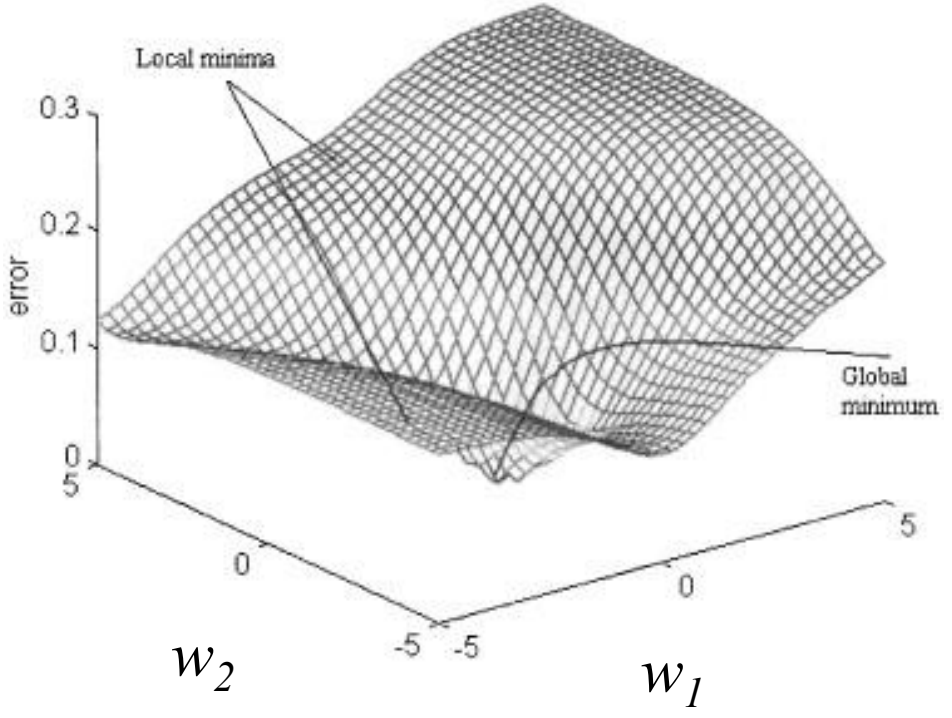
1. Compute the gradient $\nabla E_d[\boldsymbol{w}]$
2. $\mathbf{w} \leftarrow \boldsymbol{w} - \eta \nabla E_d[\boldsymbol{w}]$

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

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

Gradient Descent in weight space

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



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

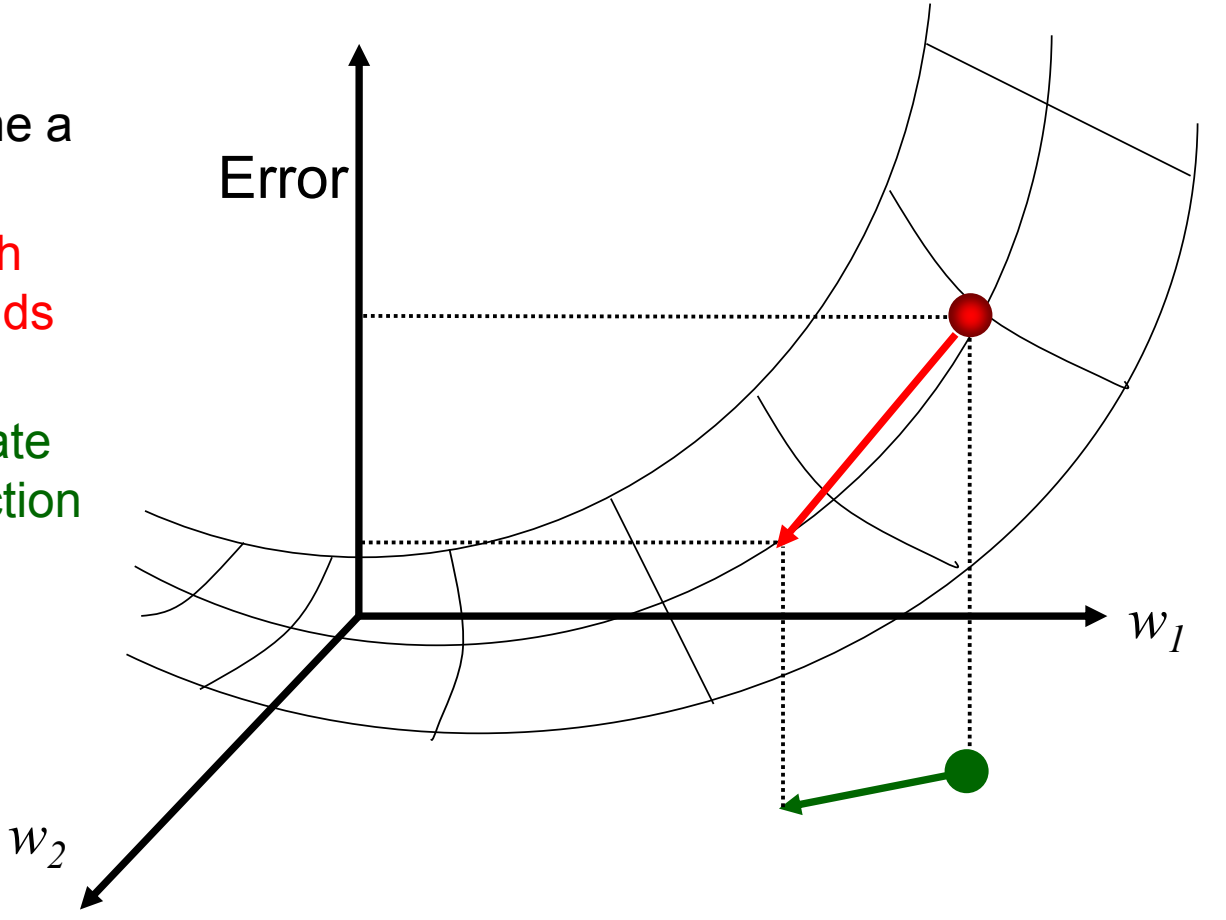
figure from Cho & Chow, *Neurocomputing* 1999

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



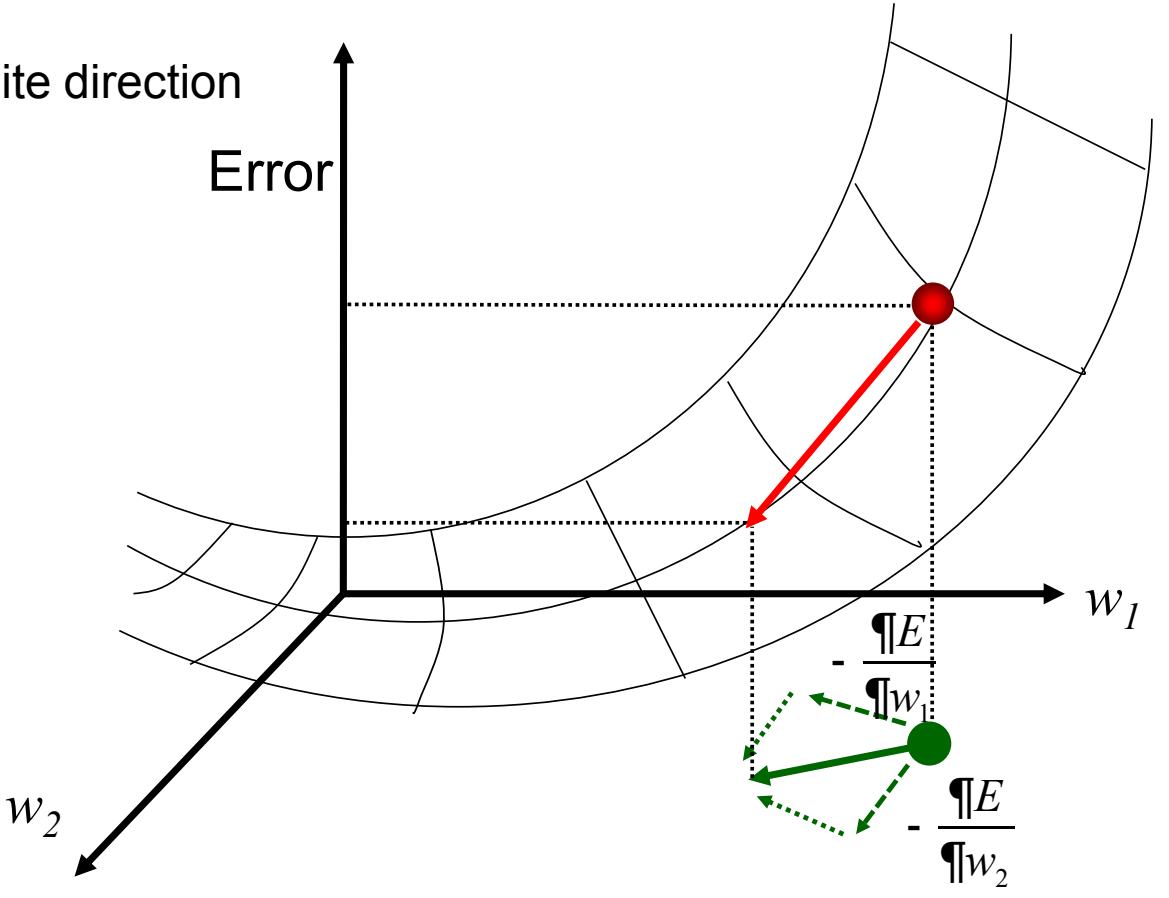
Gradient Descent in weight space

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

Take a step in the opposite direction

$$\Delta \boldsymbol{w} = -\eta \nabla E(\boldsymbol{w})$$

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



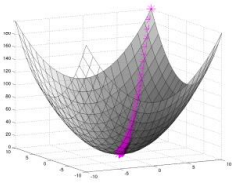
Taking derivative: chain rule

Recall the chain rule from calculus

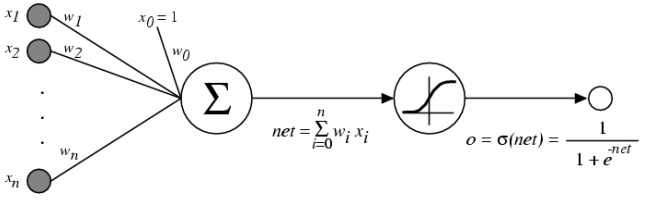
$$y = f(u)$$
$$u = g(x)$$

$$\frac{\P y}{\P x} = \frac{\P y}{\P u}\frac{\P u}{\P x}$$

Gradient Descent for the Sigmoid unit



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



o_d = observed unit output for x_d

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

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 = \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) = \sum_{d \in D} (t_d - o_d) \left(-\frac{\partial o_d}{\partial w_i} \right)$$

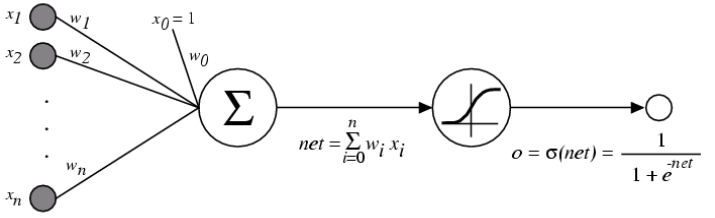
$$= -\sum_{d \in D} (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}$$

But we know: $\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$ and $\frac{\partial net_d}{\partial w_i} = \frac{\partial (\boldsymbol{w} \cdot \boldsymbol{x_d})}{\partial w_i} = x_{i,d}$

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 **w** to minimize $\sum_{d \in D} (o_d - t_d)^2$



o_d = observed unit output for x_d

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

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

δ_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-hiddent layer

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

Dropped w_0 to make derivation simpler

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

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

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

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

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

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

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

Gradient Descent for 1-hiddent layer

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

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

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

Dropped w_0 to make derivation simpler

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

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

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

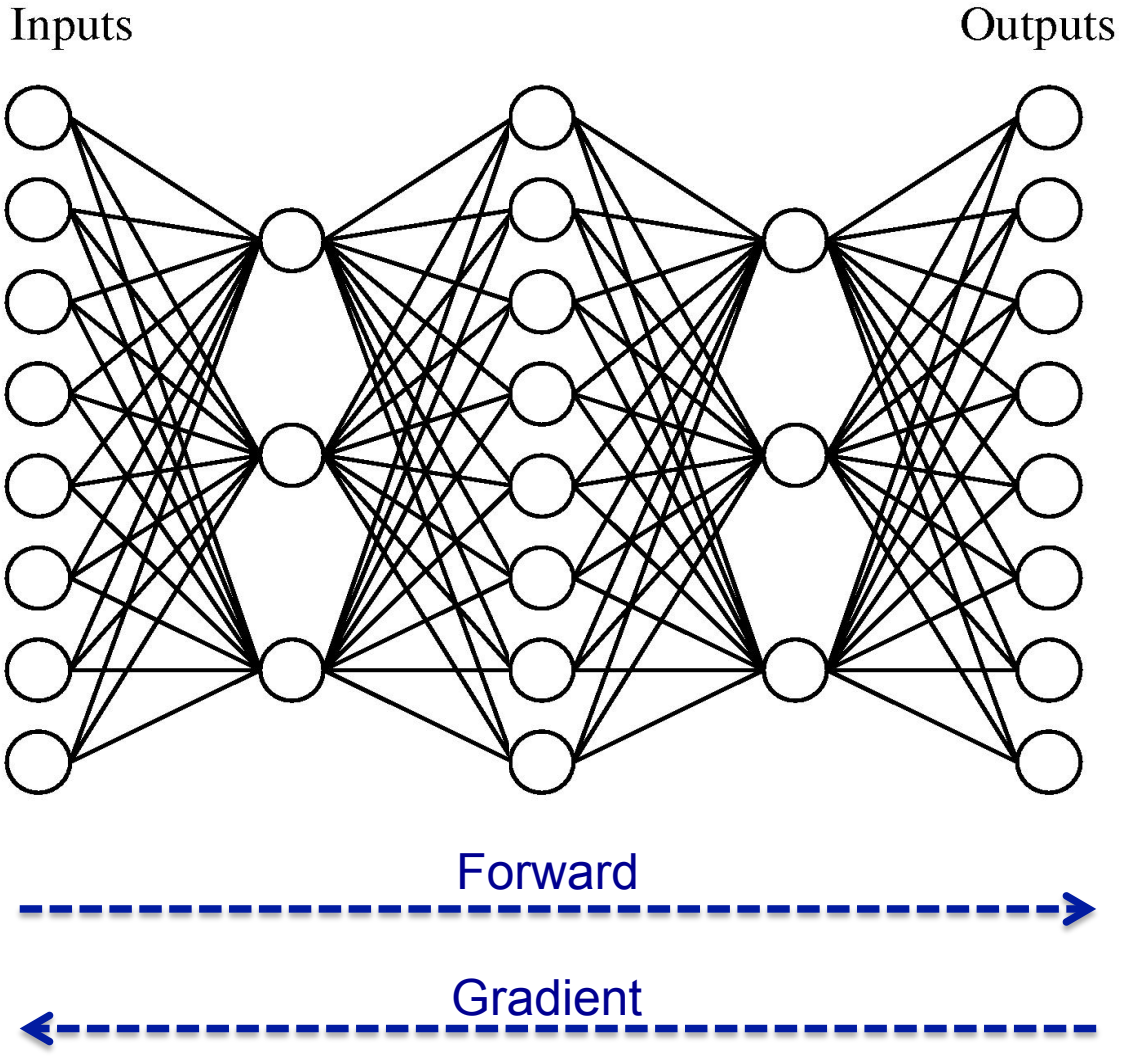
For hidden layer, two parts:

- 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 $U_1, U_2, \ldots$:

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

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 $U_1, U_2, \ldots$
    - Update weight w_i^k
    - Repeat (move to preceding layer)

Back-propagation - pseudocode

Initialize all weights to small random numbers

- Until convergence, do:
  - For each training example x,y:
    1. Forward propagation, compute node values V_k
    2. For each output unit o (with labeled output y):

$$\delta_o = V_o(1-V_o)(y-V_o)$$

    3. For each hidden unit h:

$$\delta_h = V_h(1-V_h) \sum_{k \text{ in output}(h)} w_{h,k}\delta_k$$

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

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

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:

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

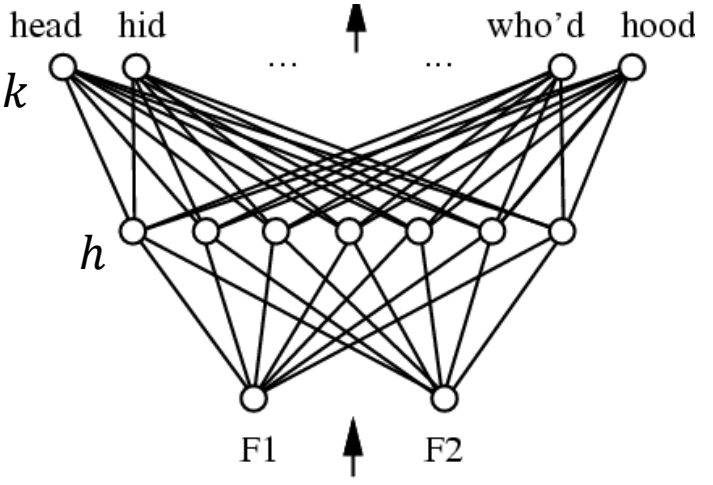
  3. For each hidden unit h:

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

  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

$x_{i,j}$ = ith input to jth unit

w_{ij} = wt from i to j