BMI-203: Biocomputing Algorithms
Lecture 5: Optimization I

Ajay N. Jain, PhD

Associate Professor, Cancer Research Institute and Dept. of Laboratory Medicine

University of California, San Francisco

ajain@cc.ucsf.edu

http://jainlab.ucsf.edu

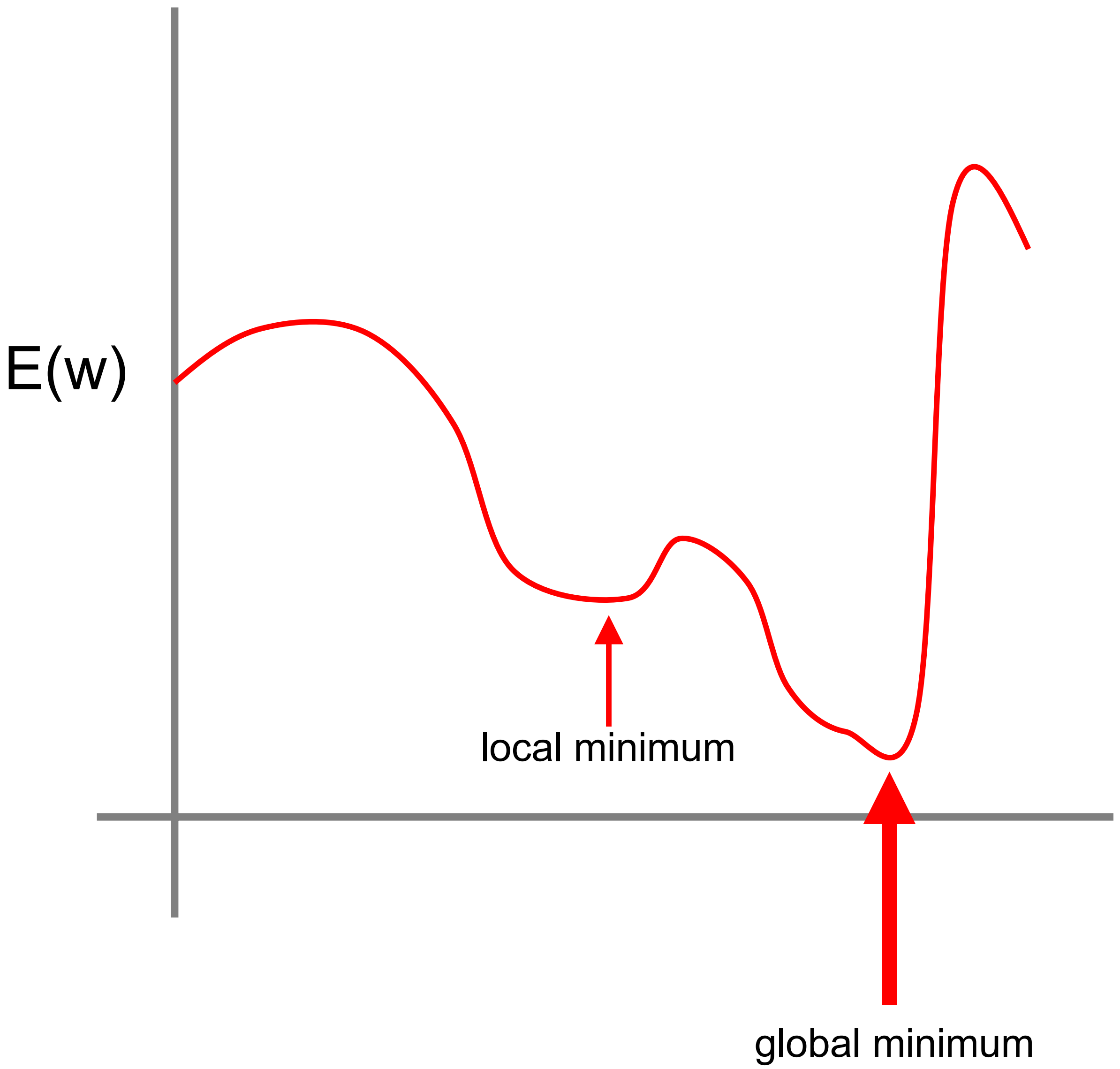
Outline

- Optimization introduction
  - General statement of the optimization problem
  - Problems that embed aspects of optimization
  - Optimization smorgasbord
- Gradient descent
- Preceptrons and neural networks
- Gradient descent in neural networks
- Homework (due May 4th)

What is “optimization”?

- Optimization is the name given to the field of study concerned with finding the **values** for sets of independent variables that **minimize** (or maximize) some **function**
- A good reference is Numerical Recipes in C (Chapter 10):
  http://www.ma.utexas.edu/documentation/nr/bookcpdf.html
- It gives the hairy details of many optimization methods plus code to implement them in many cases.

Local vs. global minima



What problems embed optimization?

- Many tasks in biocomputing
  - Finding the lowest energy state of a molecule
  - Finding the optimal orientation and conformation of a molecule docked to a receptor
  - Determining the optimal alignment of two sequences subject to some local similarity function (DP solution)
- Essentially all machine-learning and pattern recognition problems
  - All pattern classifiers can be formally described as complex functions
  - Most have some parameters that need to be estimated
  - Neural networks, genetic algorithms, Bayesian classifiers, etc… implement optimization strategies
  - Lectures II-III on Optimization will discuss machine learning

Optimization algorithms

- Stochastic
  - Random walk
  - Monte Carlo
  - Genetic Algorithms

- Non-stochastic
  - Need no gradient
  - Need the first derivative
  - Need the second derivative too!

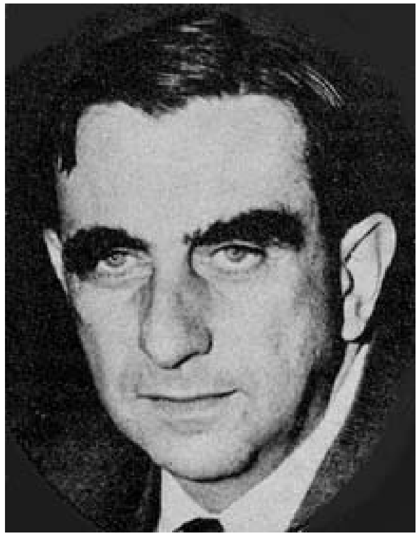
Random Walk

- Given a function in some n-dimensional space, find its (global) minimum
- Pick a dimension at random and take a step
- Evaluate the function
- Reject if new value is not better than old
- Repeat until frustrated (some number of steps yield no improvement)
- **Problem**: You can get “wedged”

Monte Carlo

- Proposed in 1953 by Metropolis (with Teller)
- Instead of always accepting a step if it is better, we reject such a step with a certain probability (delta is below is "cost" of poor step)

$$e^{-\Delta E / kT}$$



Simulated Annealing

- Proposed in 1983 by Kirkpatrick, Gelatt, and Vecchi
- Small tweak: we vary T to make a “cooling schedule”
- In the early part of search, we choose high T (high prob of making bad step), then we reduce T

$$e^{-\Delta E / kT}$$

Genetic Algorithms

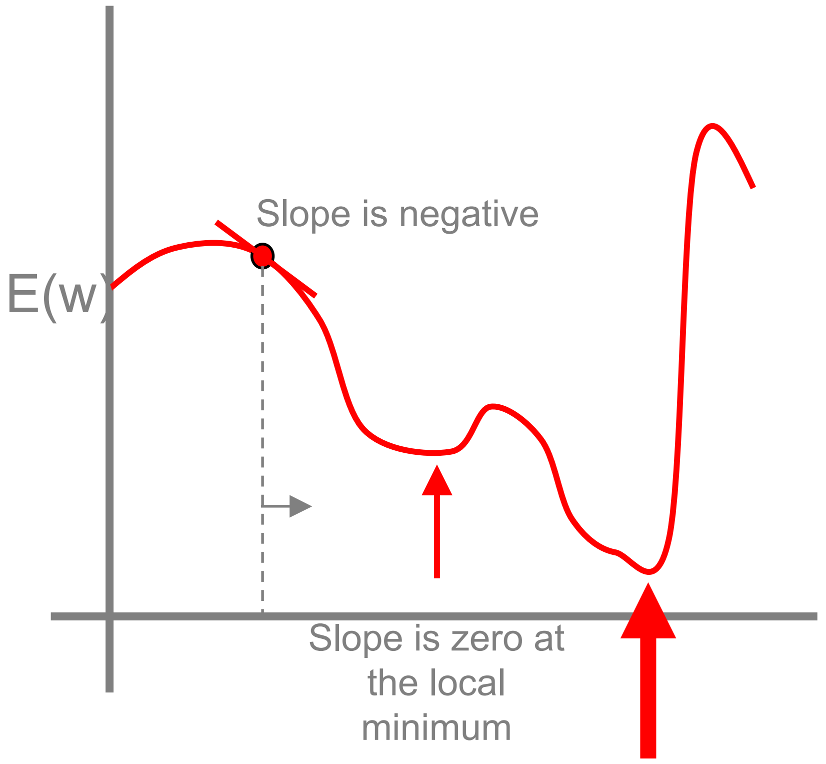
- We construct a representation of our function space where operators such as crossover and mutation make sense
- The fitness of individuals is our function
- We formally define our population operators
- We simulate the evolution of a population so as to extremize the function

Random vs. not

- I am not a big fan of stochastic optimization algorithms
- However, they can be very useful as generic solutions to optimization and can be used to prove feasibility
- In the case of non-stochastic optimization, we will focus on the case where we need the first derivative, but not the second
  - Remember, this is a **HUGE** field, and it will often pay you hugely to be clever about optimization

Gradient-based optimization: gradient descent

- Local optimization method
  - Will find a local minimum
  - However, may get stuck in a local minimum
  - Requires efficient method for computing derivative (and that one exists!)
- Basic algorithm
  - Start with some set of parameters w_i
  - Compute change of E w/r/t each w_i, where E is the function to optimize
  - Modify weights according to the direction of the gradient

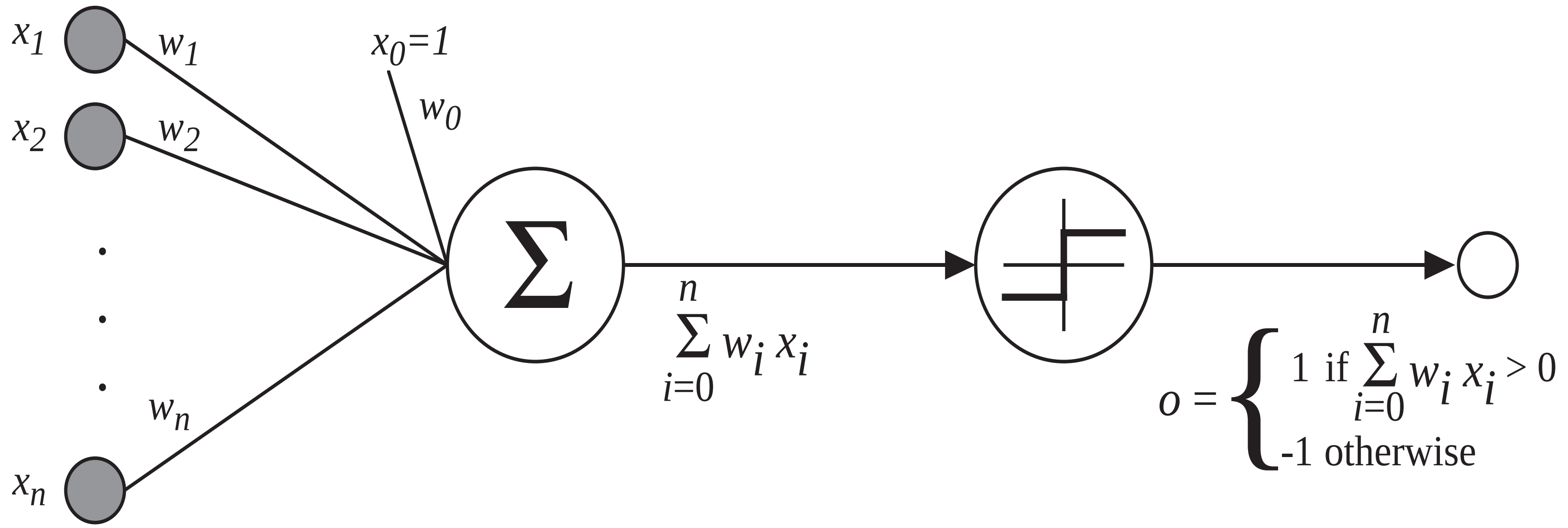


$$w_i = w_i - \eta \frac{\partial E}{\partial w_i}$$

Artificial Neural Networks: Just function optimization

- “Neural Nets” are just a class of functions
- They happen to have some nice theoretical properties
- They also happen to have some nice practical properties: principal among them is that their parameters can be estimated by gradient-descent
- We will discuss these as an in-depth example of optimization
  - History: the “perceptron”
  - ANN’s as cool functions
  - The backpropagation algorithm: gradient descent

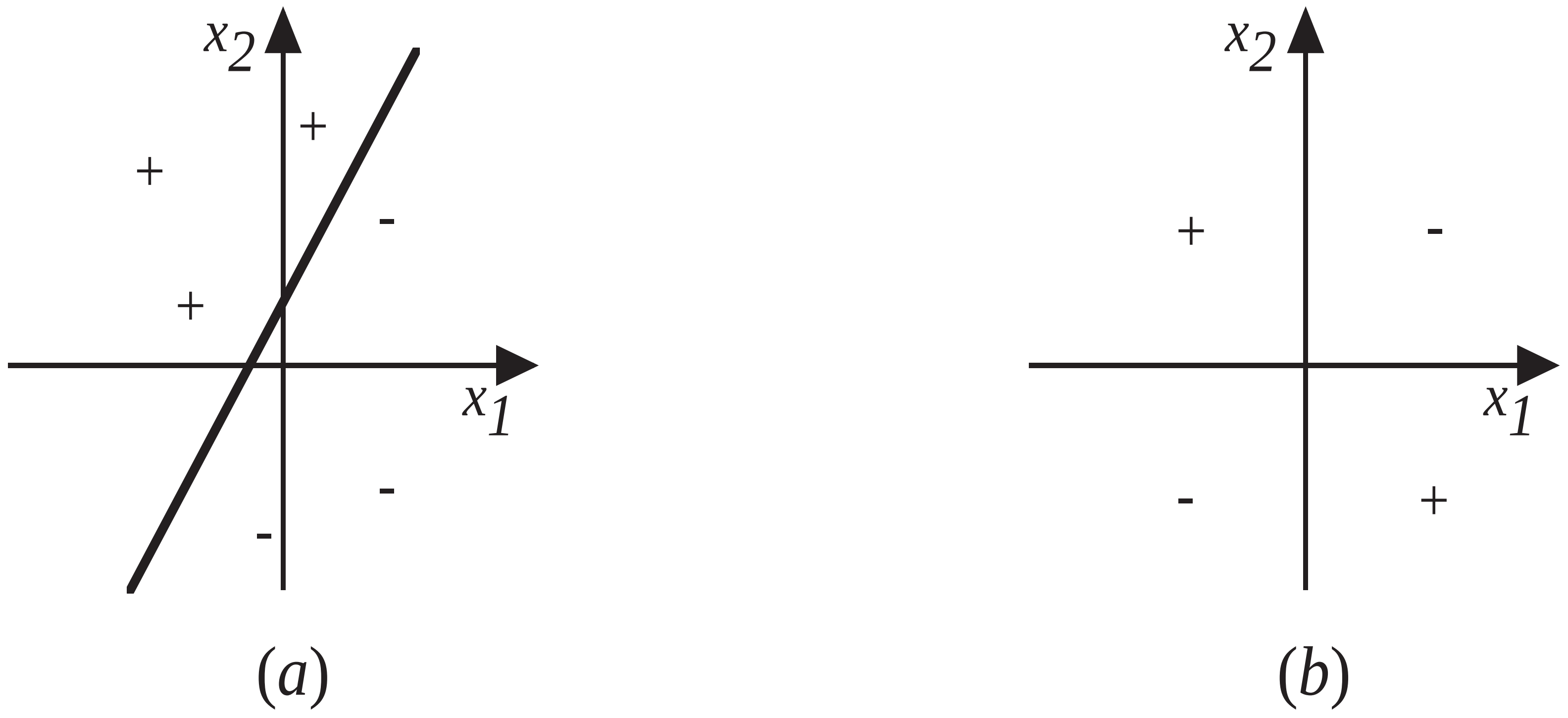
The perceptron: a linear function


x_1
w_1
x_2
w_2
$x_0=1$
w_0
x_n
w_n
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

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

The perceptron encodes a line in a high-dimensional space. For machine-learning, it can solve simple problems: those that are linearly separable.

The decision surface of a perceptron permits it to solve linearly separable problems



Minsky and Papert killed “neural nets” in 1969 for about 15 years by showing that perceptrons couldn’t solve simple problems like (b).

The perceptron training rule is just gradient descent

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

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

D is the set of training examples. E defines a standard error function. Our job is to minimize the error over the training examples.

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\left(\tfrac{1}{2}\sum_d (t_d - o_d)^2 \right)$$

$$= \left(\tfrac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \right)$$

$$= \tfrac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$

$$= \sum_d (t_d - o_d)\left(-\frac{\partial}{\partial w_i} o_d \right)$$

$$\frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d)(-x_{i,d})$$

Perceptron training rule

$$\frac{\partial E}{\partial w_i} = \sum_d \left(t_d - o_d\right)\left(-x_{i,d}\right)$$

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

Does the training rule make sense?

- When we have the "right" answer for an example ($t_d = o_d$)
  - Contribution to gradient is 0
  - No effect on any weights
- When a particular input is 0
  - Contribution for the particular weight is 0
- If output is too low
  - And $x_{i,d}$ is positive, then contribution to error gradient is negative
  - The effect on the weight change will be positive

$$\frac{\partial E}{\partial w_i} = \sum_d \left(t_d - o_d\right)\left(-x_{i,d}\right)$$

Training algorithm

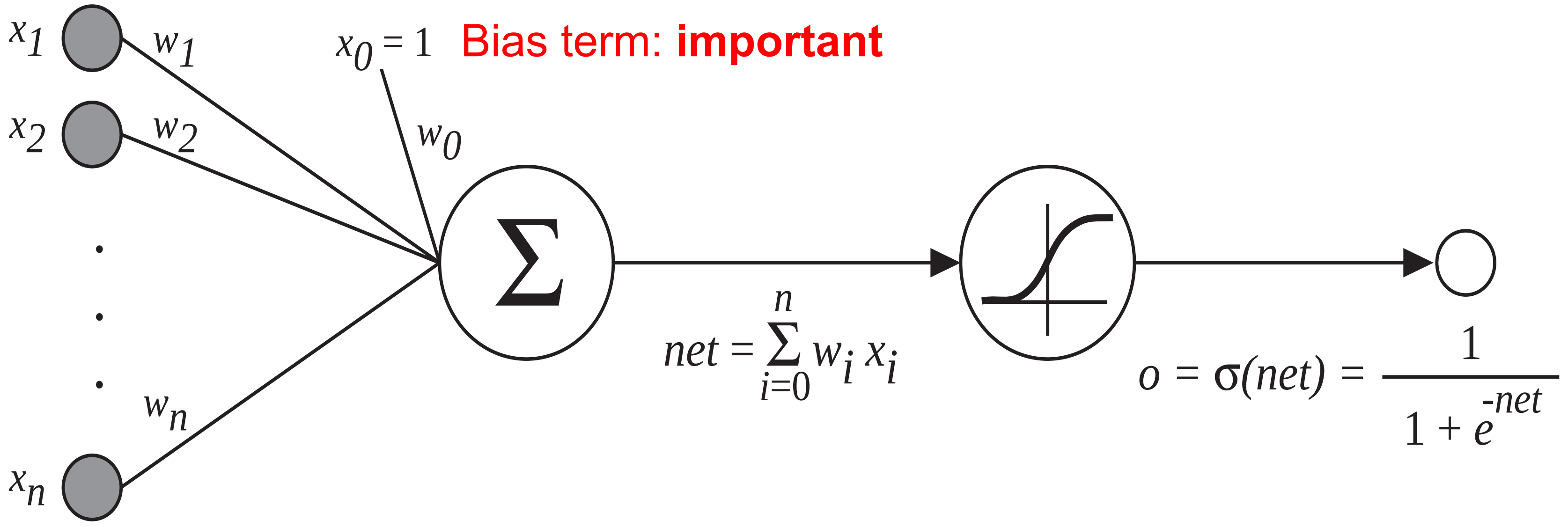
- Given a set of training examples ($x_{1\dots n}$,t) where we have a vector of input values and a target
- Initialize each w_i to a small random value
- Until we terminate, do
  - Initialize each Δw_i to zero
  - For each training example, do
    - Compute the output o with the current weights
    - For each weight

      $$\Delta w_i \leftarrow \Delta w_i + \eta (t - o) x_i$$
  - For each weight, modify it:

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

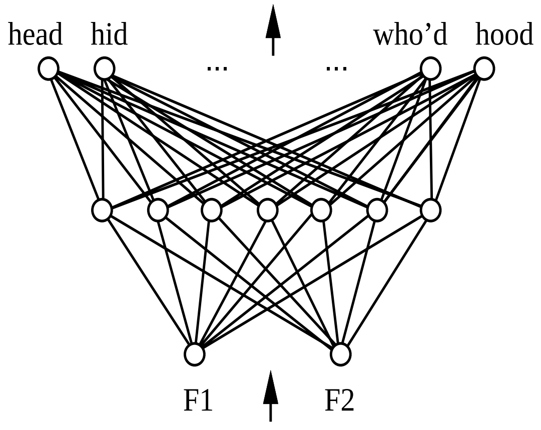
- The perceptron training algorithm is guaranteed to succeed if
  - Training examples are linearly separable
  - Learning rate is sufficiently small
- Notes
  - Multiple linear units are always re-representable as a single linear unit
  - Very few problems are linearly separable
  - Can perform incremental gradient descent (modify weights after each example): sometimes converges faster, allows for skipping examples that are "close enough"

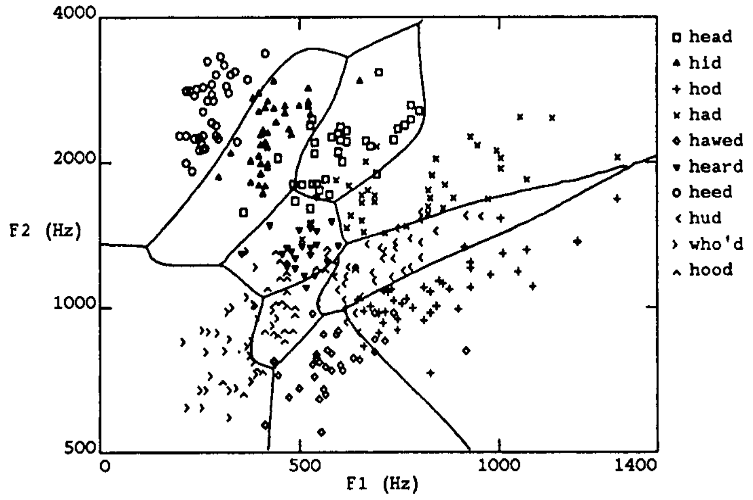
The sigmoid unit: non-linearity generates functional complexity, true ANNs



$$net = \sum_{i=0}^{n} w_i x_i$$

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





The sigmoid unit: error gradient

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\left(\tfrac{1}{2}\sum_d (t_d - o_d)^2\right)$$

$$= \left(\tfrac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2\right)$$

$$= \tfrac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$

$$= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right)$$

$$= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial net_d}\frac{\partial net_d}{\partial w_i}\right)$$

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

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

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

$$\frac{\partial net_d}{\partial w_i} = x_{i,d}$$

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

The sigmoid unit: error gradient

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\left(\tfrac{1}{2}\sum_d (t_d - o_d)^2\right)$$

$$= \left(\tfrac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2\right)$$

$$= \tfrac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$

$$= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right)$$

$$= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial net_d}\frac{\partial net_d}{\partial w_i}\right)$$

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

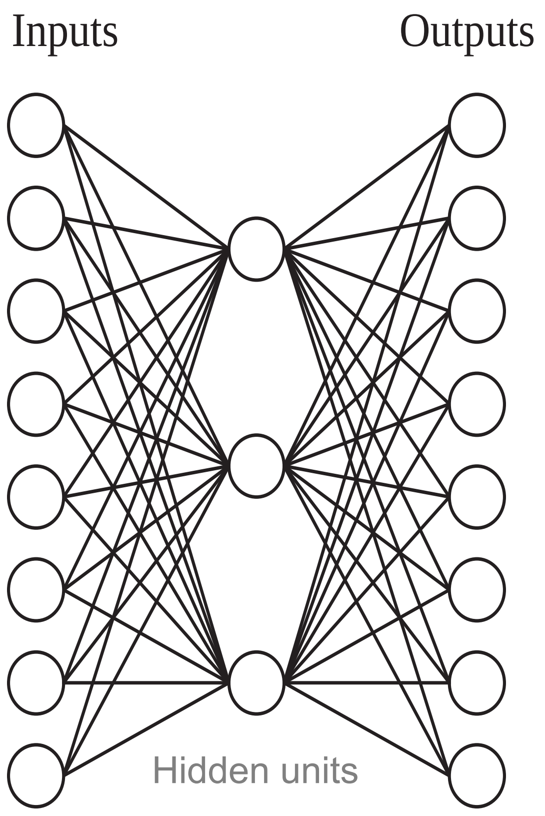
$$\frac{d}{dx}\sigma(x) = \sigma(x)\left(1-\sigma(x)\right)$$

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

$$\frac{\partial net_d}{\partial w_i} = x_{i,d}$$

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

Multi-layer networks: Including a “hidden” layer of network nodes



- These networks encode very powerful functions
  - Every boolean mapping can be represented by a network with a single hidden layer (may require an exponential number of hidden units relative to inputs)
  - Every bounded continuous function can be approximated with arbitrarily small error by a network with one hidden layer
  - Any function can be approximated to arbitrary accuracy by a network with two hidden layers
- How do we compute gradients all the way through?

We define an intermediate value for all units The change in error w/r/t “my input”: δ

- For an output unit, we are going to directly compute how the change in its input affects the error
- For a hidden unit, in order to figure out its effect on network error, we need to figure out how it affects the behavior of the units to whom it connects

For an output unit k

$$\delta_k = \frac{\partial E}{\partial net_k} = \frac{\partial}{\partial net_k}\left(\tfrac{1}{2}\sum_d \left(t_{d,k} - o_{d,k}\right)^2\right)$$

$$= \left(\tfrac{1}{2}\sum_d \frac{\partial}{\partial net_{d,k}}\left(t_{d,k} - o_{d,k}\right)^2\right)$$

$$= \sum_d \left(\left(t_{d,k} - o_{d,k}\right)\frac{\partial}{\partial net_{d,k}}\left(t_{d,k} - o_{d,k}\right)\right)$$

$$= \sum_d \left(\left(t_{d,k} - o_{d,k}\right)\left(-\frac{\partial o_{d,k}}{\partial net_{d,k}}\right)\right)$$

$$= \sum_d \left(t_{d,k} - o_{d,k}\right)\left(o_{d,k}\right)\left(1 - o_{d,k}\right)$$

For a hidden unit h

- For a hidden unit, we need to figure out how our output affects our output units' influence on error
- At right is for a single training example (to omit the d subscripts)

$$\delta_h = \frac{\partial E}{\partial net_h} = \frac{\partial E}{\partial o_h}\frac{\partial o_h}{\partial net_h}$$

$$= \left(\sum_{k \in outputs} \frac{\partial E}{\partial net_k}\frac{\partial net_k}{\partial o_h} \right) \frac{\partial o_h}{\partial net_h}$$

$$= \left(\sum_{k \in outputs} \delta_k \frac{\partial net_k}{\partial o_h} \right) \frac{\partial o_h}{\partial net_h}$$

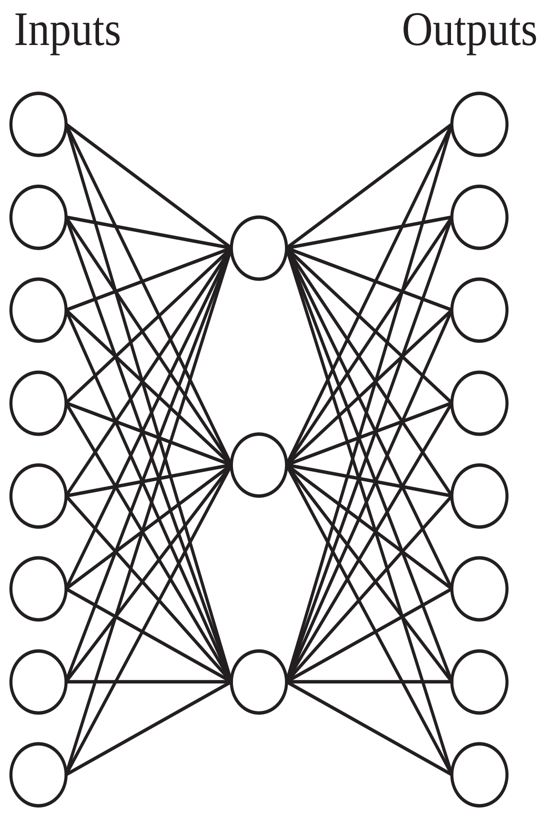
$$= \left(\sum_{k \in outputs} \delta_k \frac{\partial net_k}{\partial o_h} \right) (o_h)(1 - o_h)$$

$$= \left(\sum_{k \in outputs} \delta_k \frac{\partial}{\partial o_h} \left(\sum_{i \in inputs} w_{i,k} o_i \right) \right) (o_h)(1 - o_h)$$

$$= \left(\sum_{k \in outputs} \delta_k w_{h,k} \right) (o_h)(1 - o_h)$$

So now we can compute deltas for each unit

- We first compute deltas for the output units
- We then move one layer back and compute deltas for these units
- We continue through all of the layers
- Notes:
  - Networks with cycles of connections require special training regimes
  - The recursive back-calculation is where “back-propagation” comes from



So, now we can compute gradient for any weight

- For a particular training example, we compute the contribution to the gradient for weight $w_{i,j}$
  - First, compute the delta for unit j
  - Then, simply multiply it by the output unit i

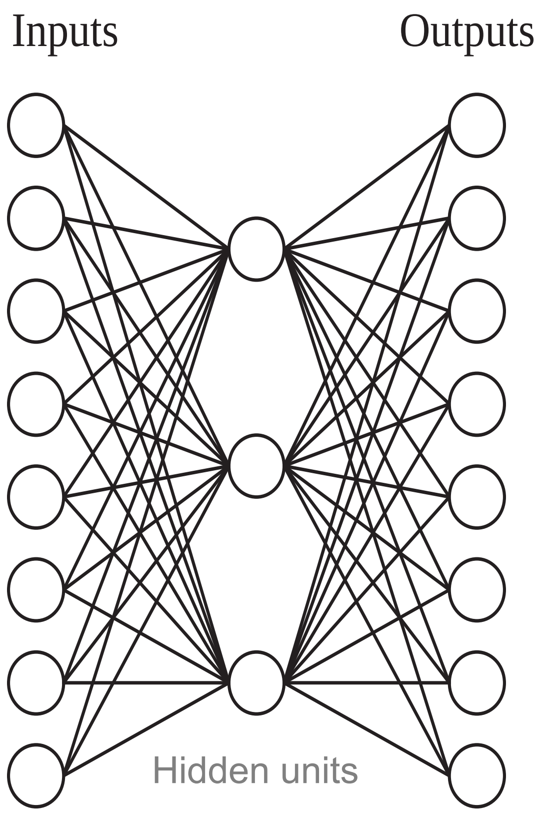
- We will follow the opposite direction of the gradient

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial net_j} \frac{\partial net_j}{\partial w_i}$$

$$= \delta_j \frac{\partial net_j}{\partial w_i}$$

$$= \delta_j o_i$$

Backpropagation Algorithm: On-line learning version (weight updates each example)



Remember: All units have a connection to a "bias unit" with constant output of 1.0

- Initialize weights to small random numbers.
- Until we are satisfied, do
  - For each training example
    - Input the training example to the input units and compute the network outputs (forward propagation)
    - For each output unit k

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

    - For each hidden unit h

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

    - Update each network weight from unit i to unit j

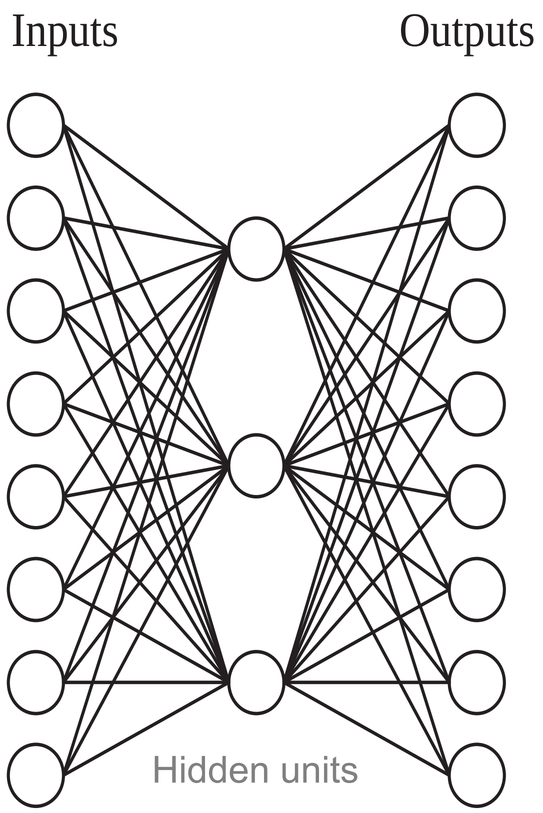
$$\Delta w_{i,j} = \eta\delta_j o_i$$

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

Networks of sigmoidal units trained via backpropagation have nice properties

- We’re performing gradient descent over the entire weight vector
- The training algorithm easily generalizes to arbitrary directed graphs
- Usually doesn’t have a problem with local minima
- Training can be slow, but using a trained network is usually very fast
- Network architecture, constraints, and functional form can be designed to suit the properties of particular problems, leading to classifiers of very high predictive performance

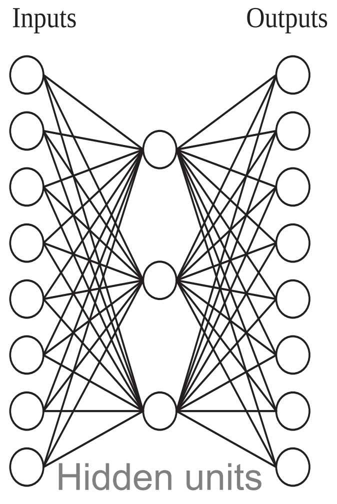
Example: An 8x3x8 encoder

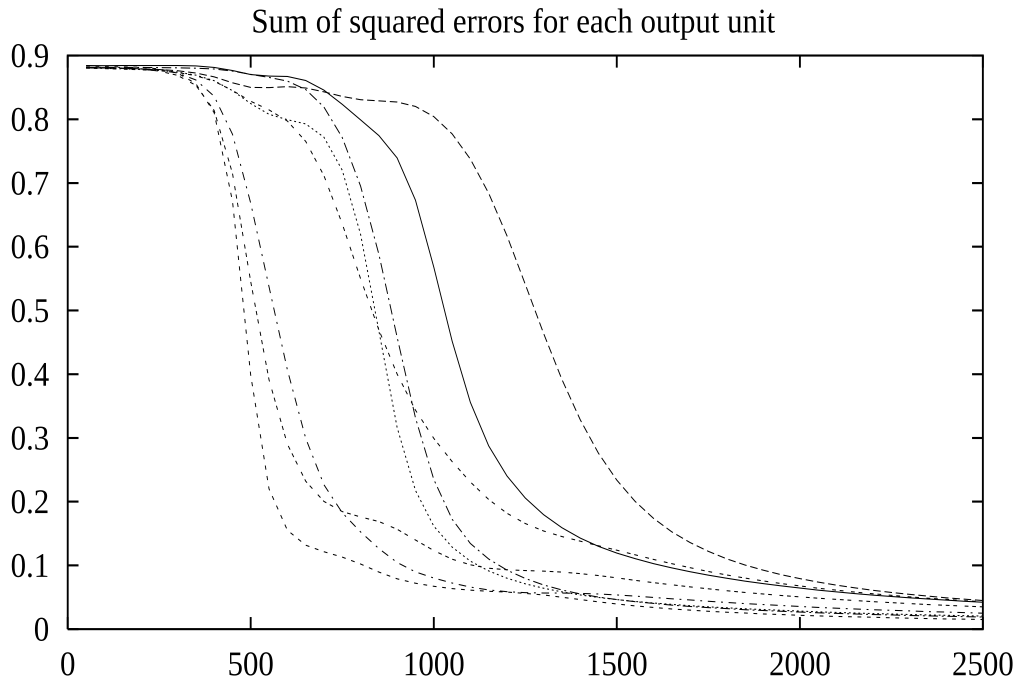


Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

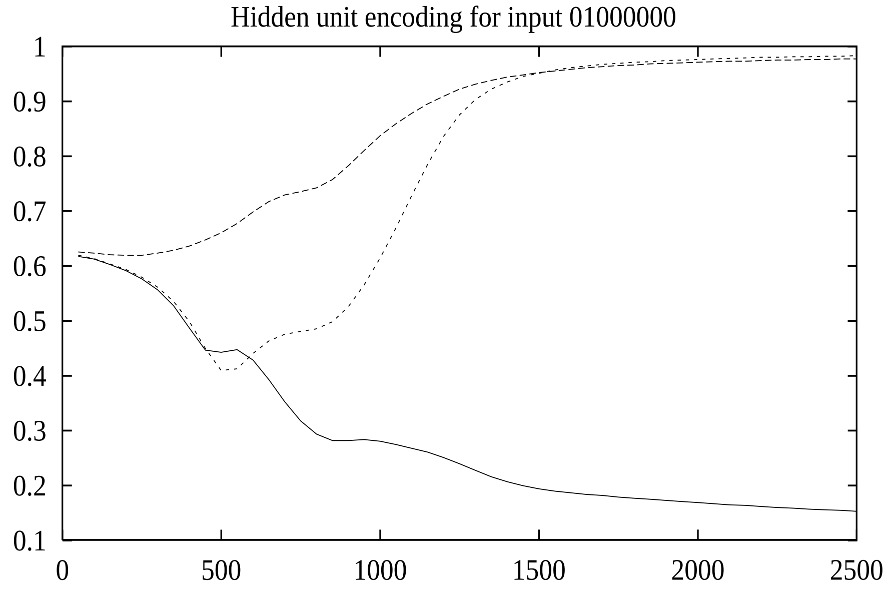
The hidden layer learns a binary representation

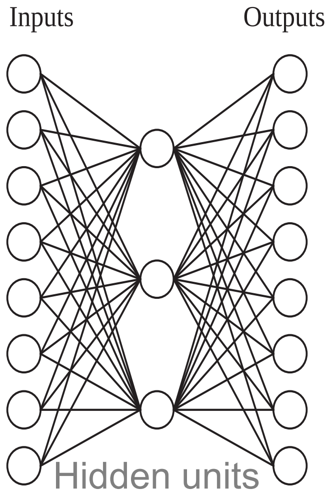
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

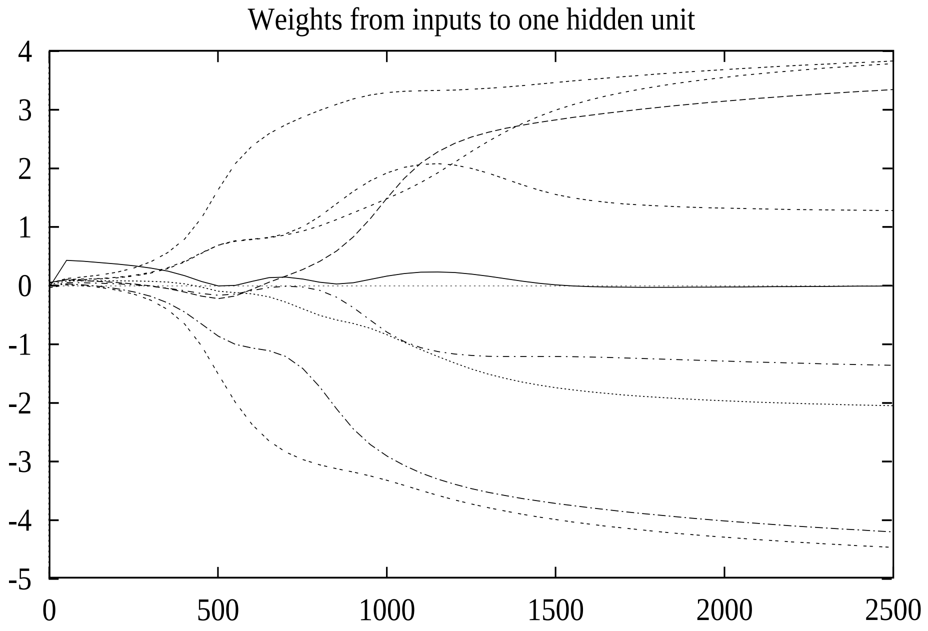




The representation and weights evolve smoothly over training







Homework 5

- Write a program that implements a neural network
  - Input: network architecture
    - Number of inputs
    - Number of hidden units
    - Number of output units
    - Assume fully connected network
  - Input: learning rate
  - Input: training data
    - One example per line as follows:
    - Input: 01000000 Output: 01000000
  - Output
    - Initial weights, Final weights
    - Final output values for each training example (also output the training example)
    - Final total error (sum of squared error for all output units over all examples)
- Run your neural network on the 8x3x8 encoder problem used as an example in this lecture
- You will need to choose a learning rate and run your network until it is able to correctly generate the binary encoding for each input example (i.e. all 1's should be > 0.5 and all 0's should be < 0.5)
- What to turn in:
  - A single file (text or pdf)
    - Program output
    - Program listing
  - Email answers to ajain@cc.ucsf.edu
  - Homework is due 5/4/04