Neural Networks

Instructor: Jesse Davis

Slides from: Pedro Domingos, Ray Mooney, David Page, Jude Shavlik

Announcements

- Homework 1 has been graded
- Homework 2 is due now
- Homework 3 is available online
- Lecture notes are available online

Outline

- Homework 1 review
- Perceptron
- Multilayer neural networks

Problem 1: Results

- MAE ~ 0.695, RMSE ~ 0.884
- Baselines:
  - User's average: MAE = 0.79, RMSE = 0.99
  - Average rating (3.0): MAE = 0.90, RMSE = 1.08
  - Random guessing: MAE = 1.42, RMSE = 1.76
- Difference between best baselines and CF is seemingly small but translates to lots of $$.

Problem 1: Optimizations

- Logical:
  - Cache the mean prediction for each user
  - Sort test records by user ID, cache Pearson's coefficients for use for subsequent users (remember: w(a,i) = w(i,a))
  - For each user, keep a sorted list of rated movies – allows O(n) identification of common movies between two users
- Technical:
  - Multithread
  - Eliminate as much output as possible
  - Avoid typecasting (e.g., don't store Objects in maps)
  - Use floats instead of doubles
  - Use retail builds

Problem 2a: Solution

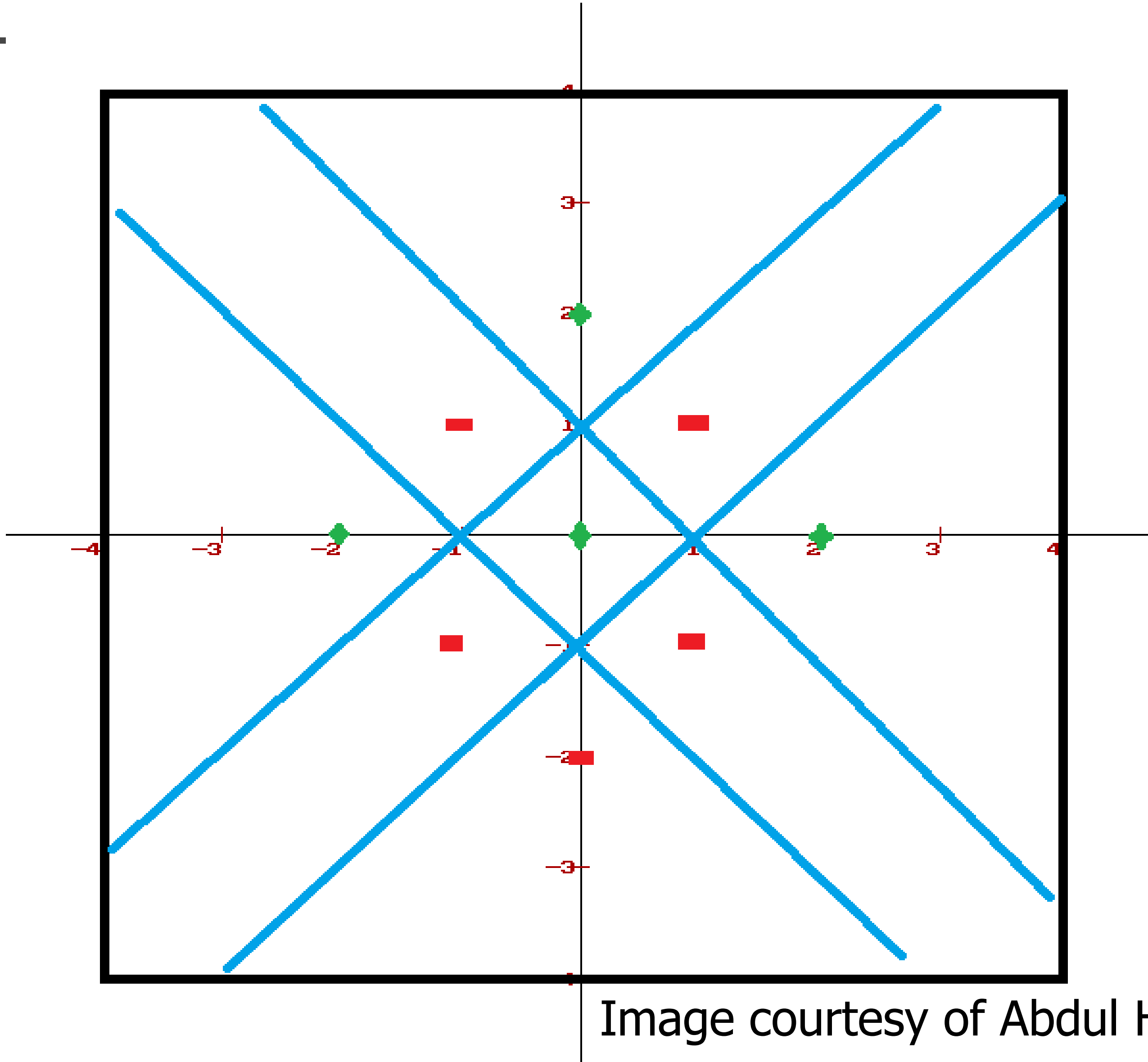
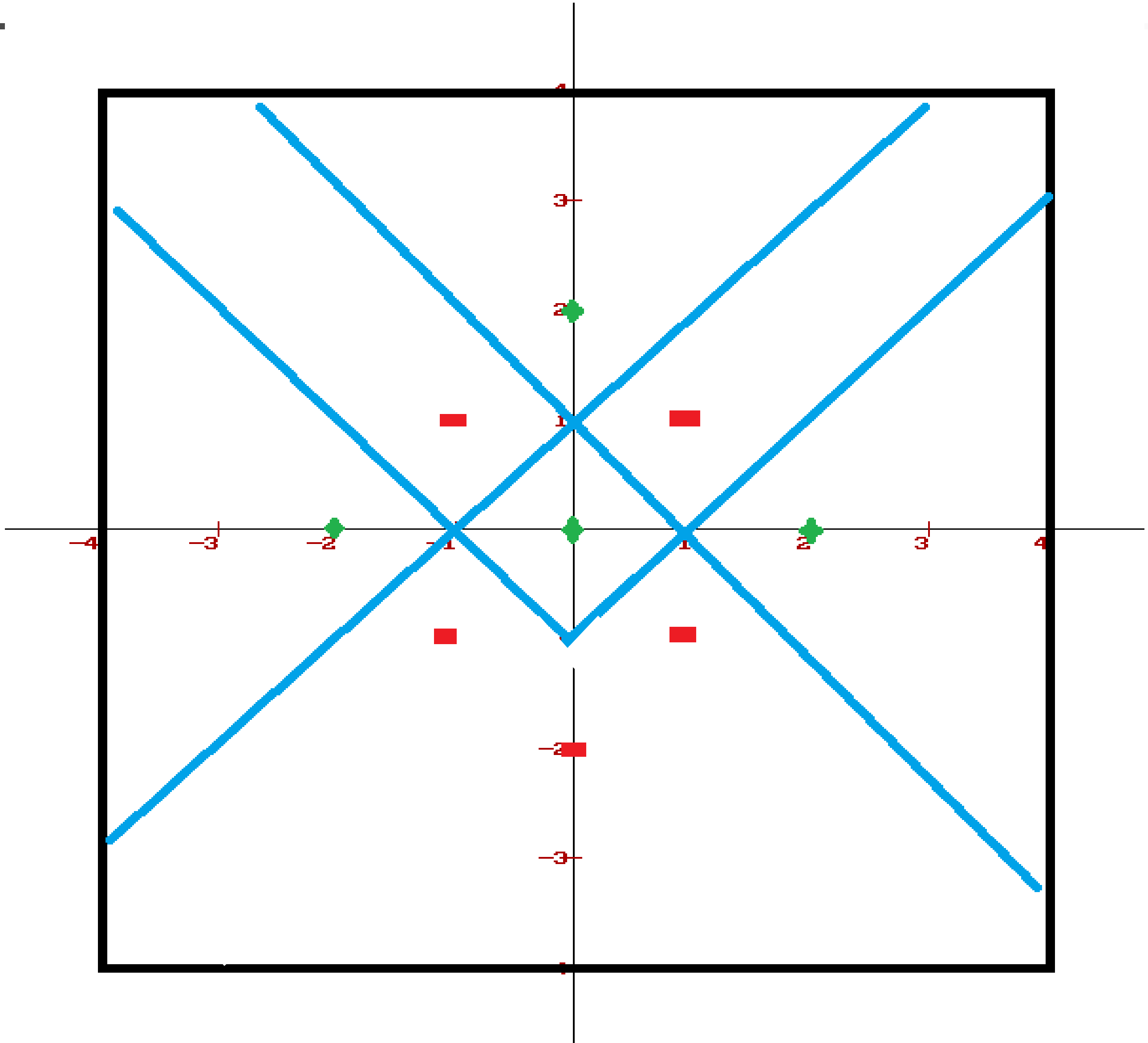


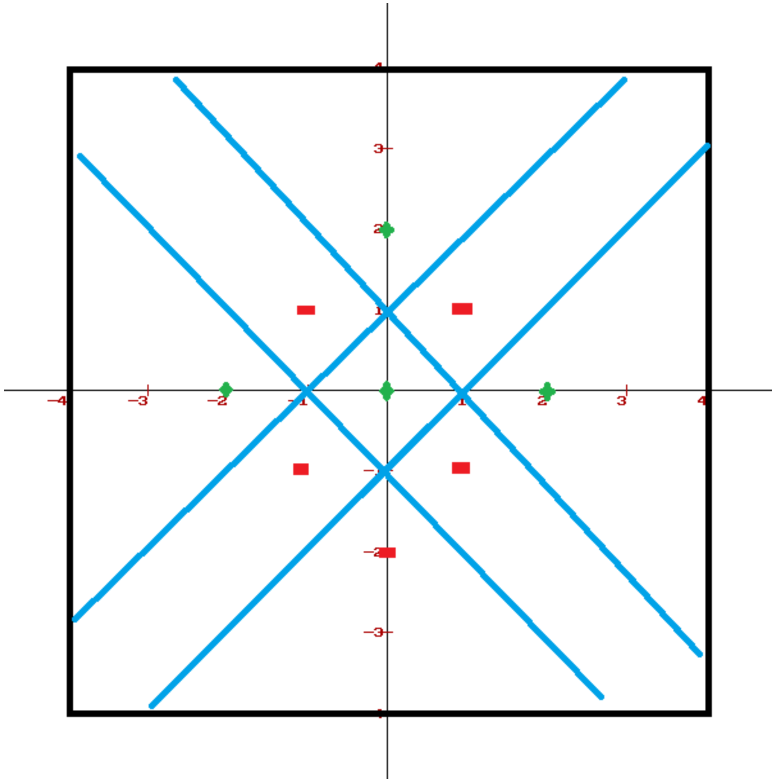
Image courtesy of Abdul Hyee Waqas

Problem 2a: Common Mistake



Problem 2b: Solution

Point	True Label	Predicted Label	Result
(0,2)	+	-	error
(-1,1)	-	+	error
(1,1)	-	+	error
(-2,0)	+	-	error
(0,0)	+	-	error
(2,0)	+	-	error
(-1,-1)	-	+	error
(1,-1)	-	+	error
(0,-2)	-	-	correct

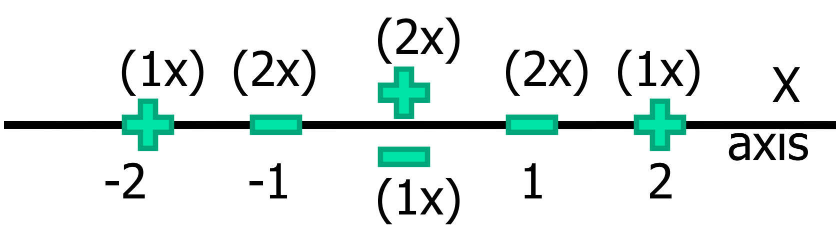


Error = 8/9

Problem 2c: Simulate Backward Elimination

- Step 1: Try eliminating Y

Point	True Label	Predicted Label	Result
(0,.)	+	-	error
(-1,.)	-	conflict	error
(1,.)	-	conflict	error
(-2,.)	+	-	error
(0,.)	+	-	error
(2,.)	+	-	error
(-1,.)	-	conflict	error
(1,.)	-	conflict	error
(0,.)	-	+	error



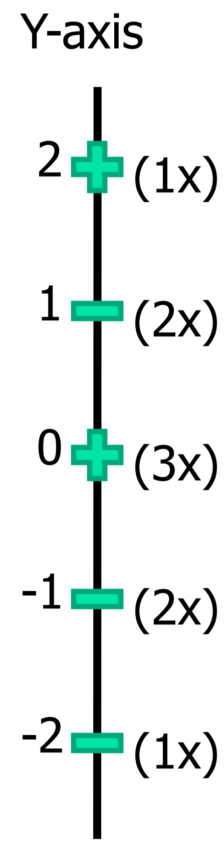
Error = 1

Problem 2c: Simulate Backward Elimination

- Step 2: Try eliminating X

Point	True Label	Predicted Label	Result
(.,2)	+	-	error
(.,1)	-	+	error
(.,1)	-	+	error
(.,0)	+	+	correct
(.,0)	+	+	correct
(.,0)	+	+	correct
(.,-1)	-	conflict	error
(.,-1)	-	conflict	error
(.,-2)	-	-	correct

Error = 5/9



Problem 2c: Simulate Backward Elimination

- Step 3: Decide which feature to drop (if any)

Drop X

Problem 2c: Simulate Backward Elimination

- Step 4: Try eliminating Y (again!)

Point	True Label	Predicted Label	Result
(.,.)	+	conflict	error
(.,.)	-	conflict	error
(.,.)	-	conflict	error
(.,.)	+	conflict	error
(.,.)	+	conflict	error
(.,.)	+	conflict	error
(.,.)	-	conflict	error
(.,.)	-	conflict	error
(.,.)	-	conflict	error

Error = 1

(4x) +

− (5x)

Problem 2c: Simulate Backward Elimination

- Step 5: Decide which feature to drop (if any)

Can’t drop anything else. Stop.

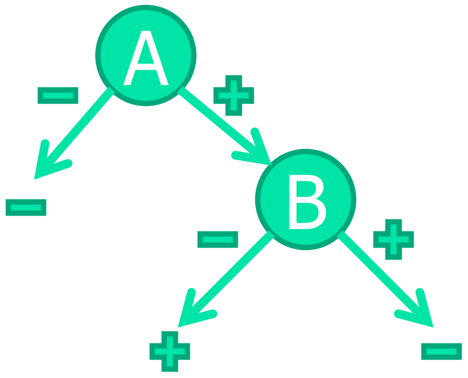
Only X gets eliminated.

Problem 2c: Common Mistakes

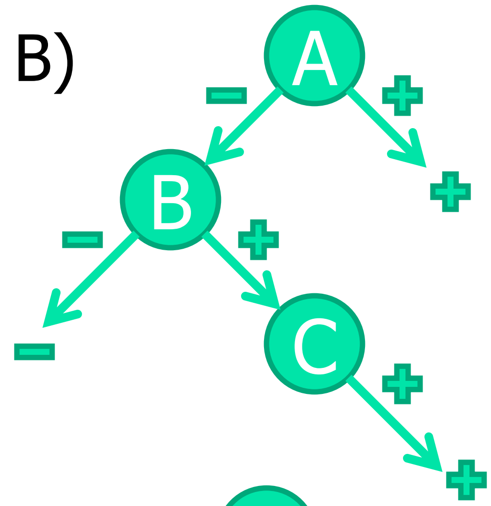
- Forgetting to consider dropping Y after X is dropped
  - Counterintuitive in this case, but B.E. does it.
- Assuming that with no features, all points get the same label (+ or -)
  - You could do that, but this is a hack.
- Assuming that different 3-NN sets *always* yield different predictions, resulting in conflicts (errors)
- Considering elimination of {X}, then {Y}, then {X,Y}
  - B.E. doesn't consider all feature subsets – it eliminates one feature at a time

Problem 3: Solutions

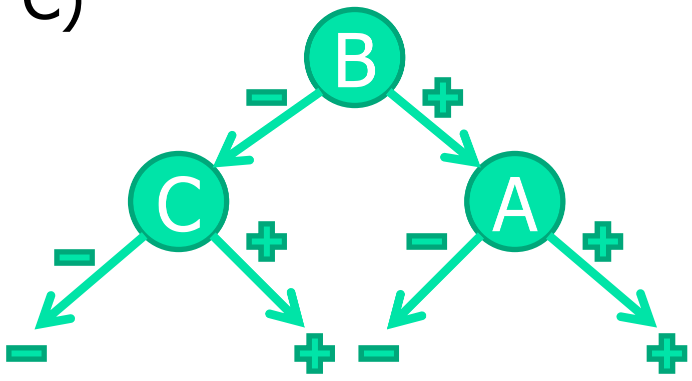
A)



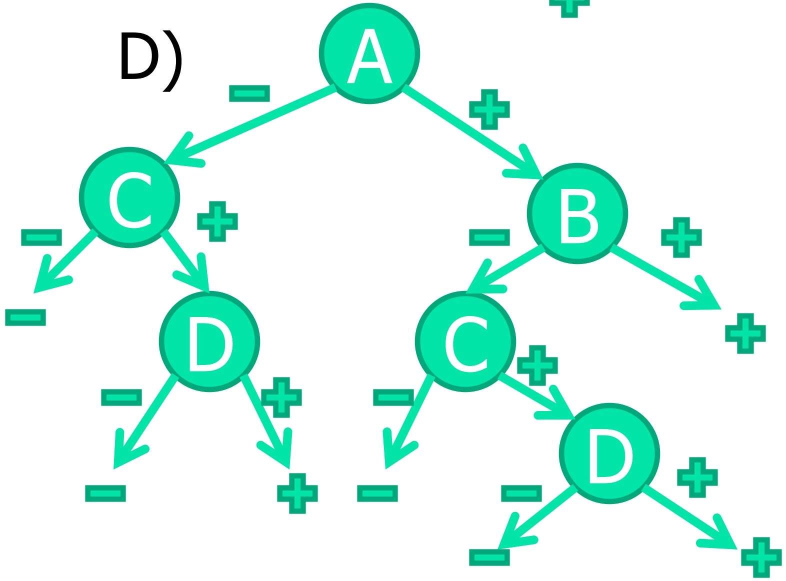
B)



C)

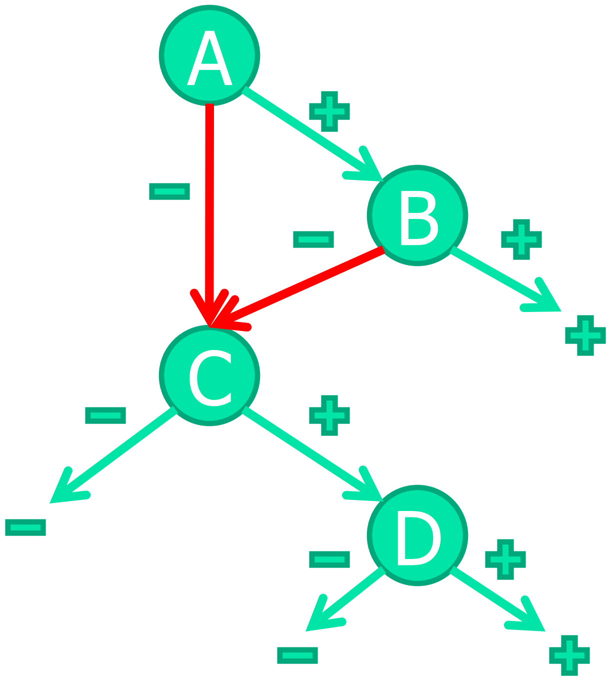


D)



Problem 3: Common Mistake

Can't do this – the result is not a tree!

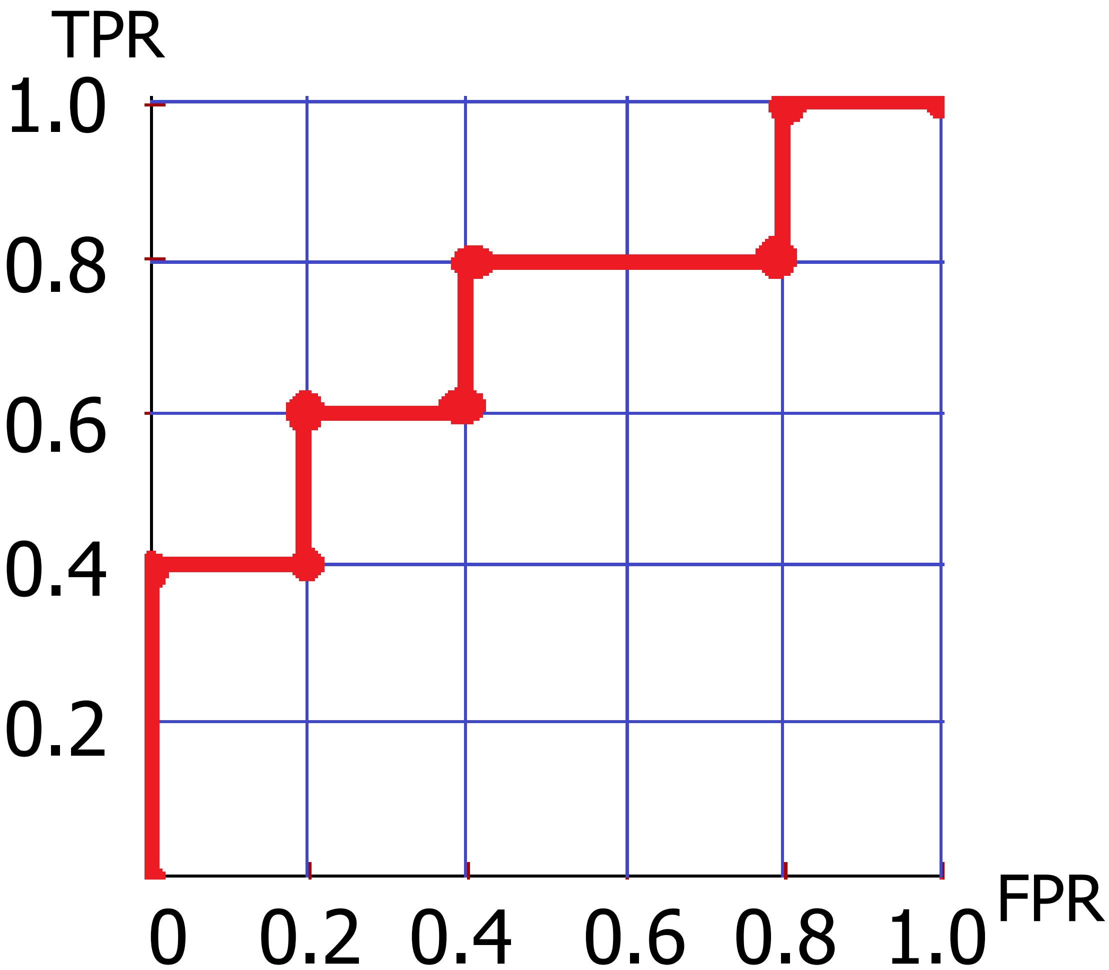


Also, finding identical subtrees in practice is very hard, and standard tree-learning algorithms don't do it.

Problem 4: Solution

- A) 1
  - # of positive and negative examples are equal
- B) 0
  - Given a2=true, # of positive and negative examples are equal, and same for a2 = false
  - Thus, knowing a2 doesn't reduce entropy in any way

Problem 5: Solution



Problem 5: Advice

- When building plots, pay attention to axes' scales and ranges
  - E.g., for the ROC, both axes should be on the same scale – they have the same units of measurement
  - When plotting probabilities, set axes' ranges to [0, 1.0] – extending them past 1.0 (e.g., to 1.2) doesn't make sense
- A little care will make your plots look *much* more convincing and professional

Outline

- Homework 1 review
- Perceptron
- Multilayer neural networks

Neural Networks

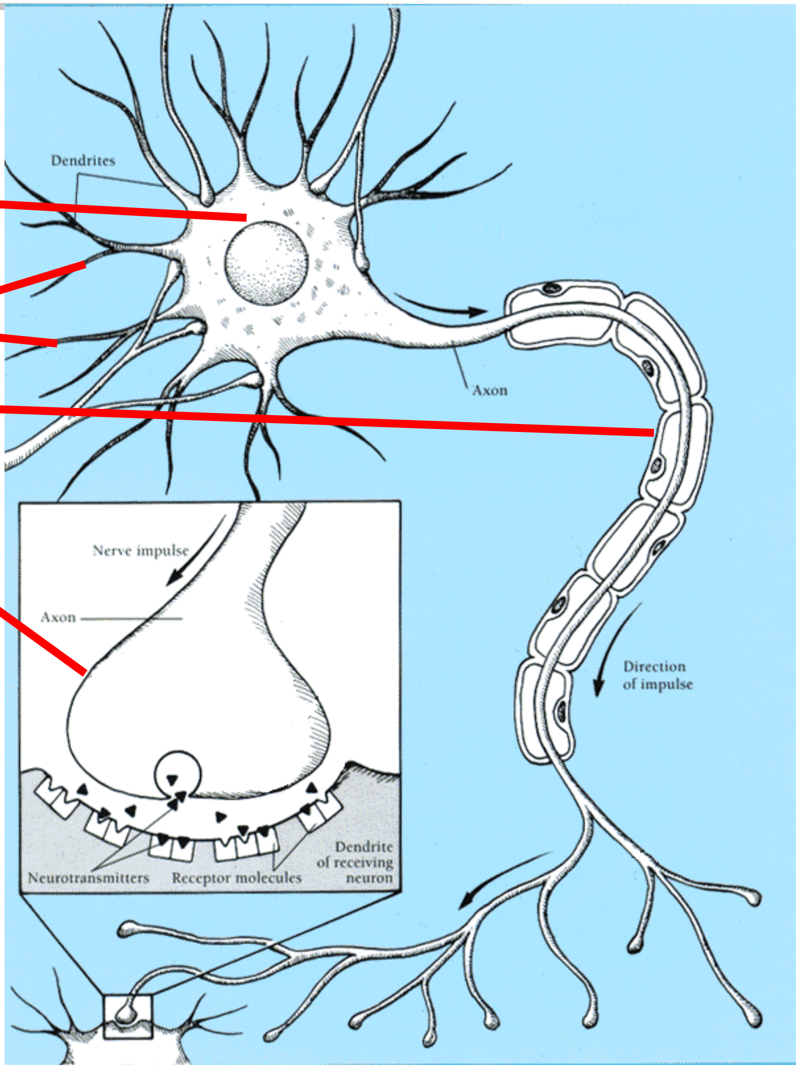
- Analogy to biological neural systems, the most robust learning systems we know
- Attempt to understand natural biological systems through computational modeling
- Massive parallelism allows for computational efficiency
- Intelligent behavior as an "emergent" property
  - Large number of simple units
  - Combine output of simple units
  - As opposed to explicitly encoded symbolic rules and algorithms

Neural Network Learning

- Learning approach based on modeling adaptation in biological neural systems
- Perceptron: Initial algorithm for learning simple neural networks (single layer) developed in the 1950’s
- Backpropagation: More complex algorithm for learning multi-layer neural networks developed in the 1980’s

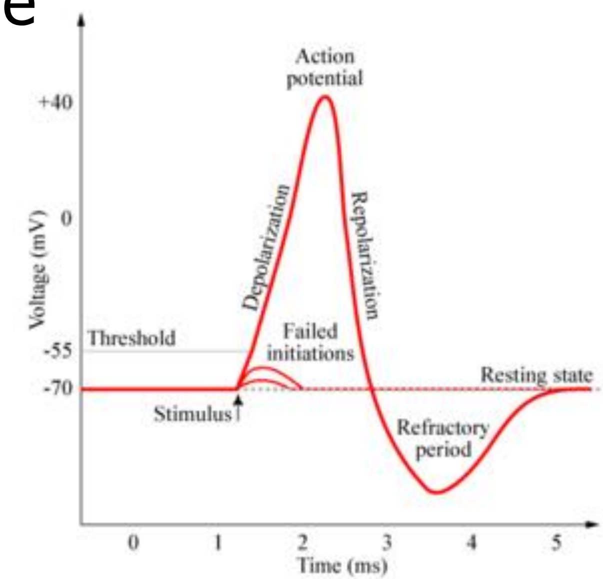
Real Neurons

- Cell structures
  - Cell body
  - Dendrites
  - Axon
  - Synaptic terminals



Neural Communication

- Electrical potential across cell membrane exhibits spikes called action potentials
- Spike originates in cell body, travels down axon, and causes synaptic terminals to release neurotransmitters
- Chemical diffuses across synapse to dendrites of other neurons
- Neurotransmitters: excititory or inhibitory
- If net input of neurotransmitters to a neuron is excititory and exceeds some threshold, it fires an action potential



Real Neural Learning

- Synapses change size and strength with experience
- Hebbian learning: When two connected neurons are firing at the same time, the strength of the synapse between them increases
- “Neurons that fire together, wire together.”

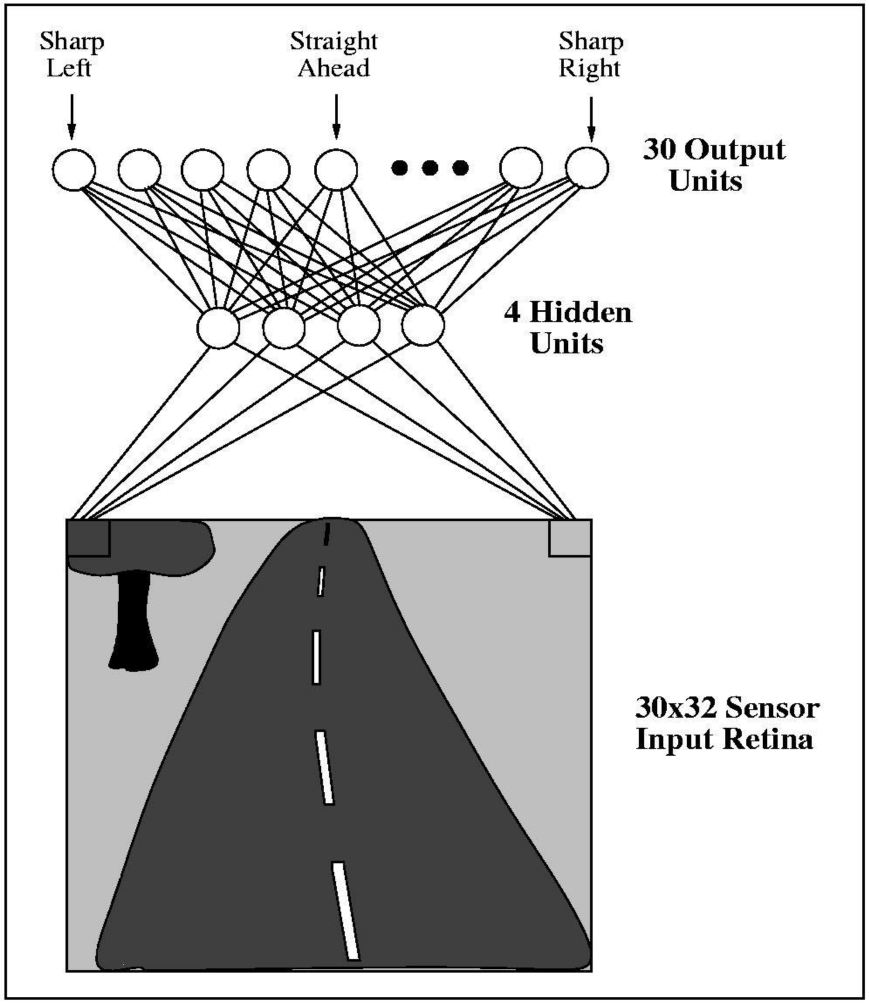
Connectionist Models

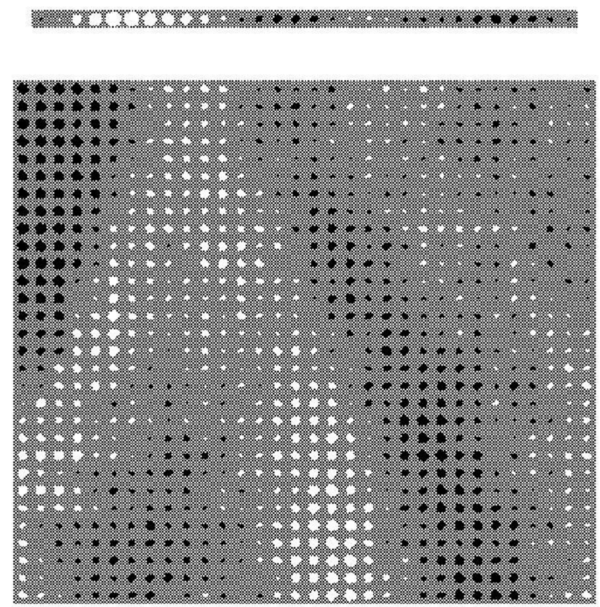
- Consider humans:
  - Neuron switching time ~ 0.001 seconds
  - Number of neurons ~10^{10}
  - Connections per neuron ~ 10^{4-5}
  - Scene recognition ~ 0.1 seconds
  - 100 inference steps seems insufficient to achieve this results

Massive parallel computation!

Properties of Neural Networks

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

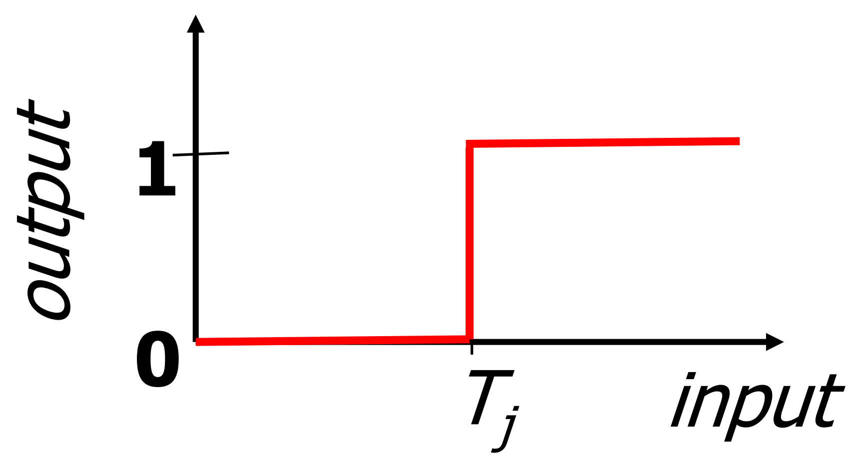

Sharp
Left
Straight
Ahead
Sharp
Right
30 Output
Units
4 Hidden
Units
30x32 Sensor
Input Retina



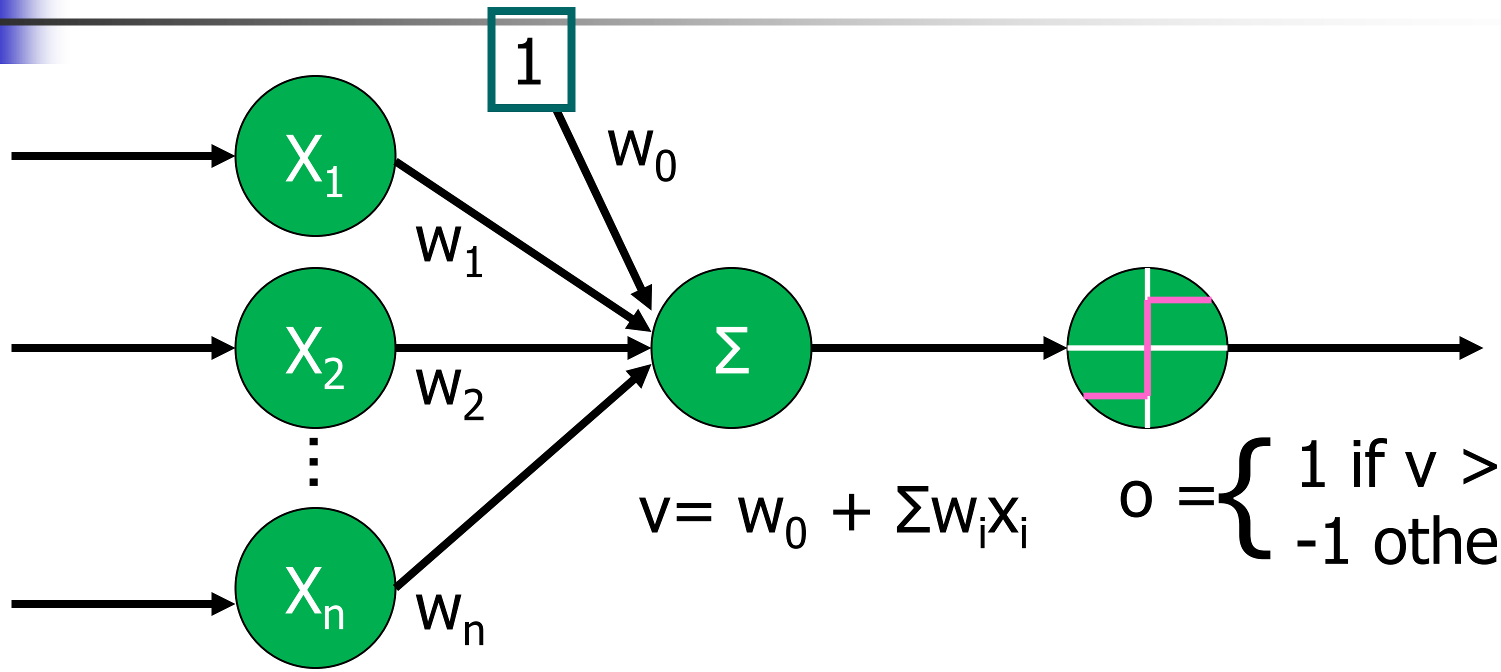
Perceptron Model

Model network as a graph with:

- Cells as nodes
- Synaptic connections as weighted edges
- Neuron fires if sum of inputs exceeds a predefined threshold



Perceptron



$v = w_0 + \Sigma w_i x_i$

$$o = \begin{cases} 1 \text{ if } v > 0 \\ -1 \text{ otherwise} \end{cases}$$

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

Vector Notation

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

Neural Computation

- McCollough and Pitts (1943) showed how such model neurons could compute logical functions and be used to construct finite-state machines.
- Can be used to simulate logic gates:
  - AND: Let all w_{ji} be T_j / n, where n is the number of inputs.
  - OR: Let all w_{ji} be T_j
  - NOT: Let threshold be 0, single input with a negative weight.
- Can build arbitrary logic circuits, sequential machines, and computers with such gates.
- Given negated inputs, two layer network can compute any boolean function using a two level AND-OR network.

Perceptron Training

Given: Set of examples, where we know the desired outputs as well as which inputs are active

Learn: Weights associated with each input such that the correct output is produced for each training example

Learning done by an iterative weight update

Perceptron Training Rule

- Weight update rule: $W_i = W_i + \eta(t_j - o_j)x_{j,i}$
  - W_i is the weight for input i
  - η is the learning rate
  - t_j is the true output for example j
  - o_j is the predicted output for example j
  - $x_{j,i}$ is the value of feature i of example j
- Intuitively, this means
  - If label is correct, do nothing
  - If weights are too high, decrease them
  - If weights are too low, increase them

Perceptron Training Algorithm

Set initial weights to random value
Repeat
    for each example X_j
        compute current output o_j
        compare o_j to t_j
        if necessary, update weights
Until all examples are labeled correctly

Epoch

Linear Separability

Consider a perceptron, its output is

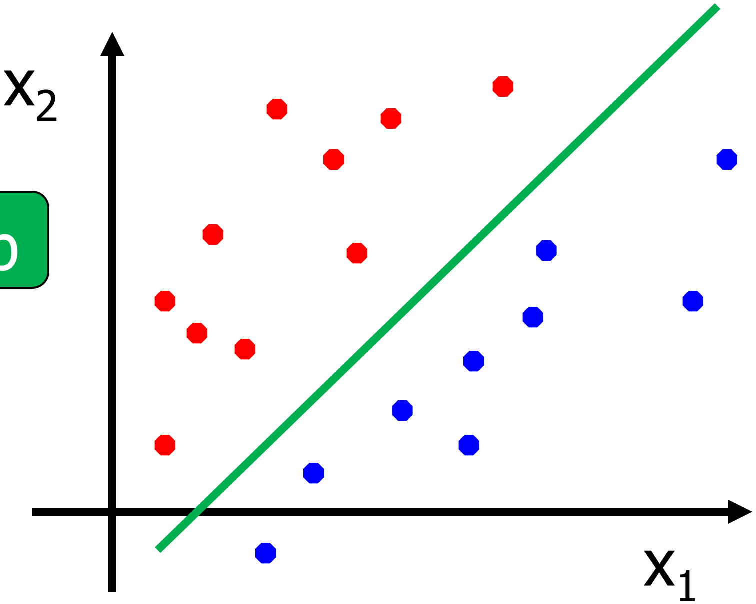
- 1 if $W_1X_1+W_2X_2 + \ldots + W_nX_n > \Theta$
- 0 otherwise

In terms of feature space

$$W_1X_1 + W_2X_2 = \Theta$$

$$X_2 = \frac{\Theta - W_1X_1}{W_2} = \frac{-W_1}{W_2}X_1 + \frac{\Theta}{W_2}$$

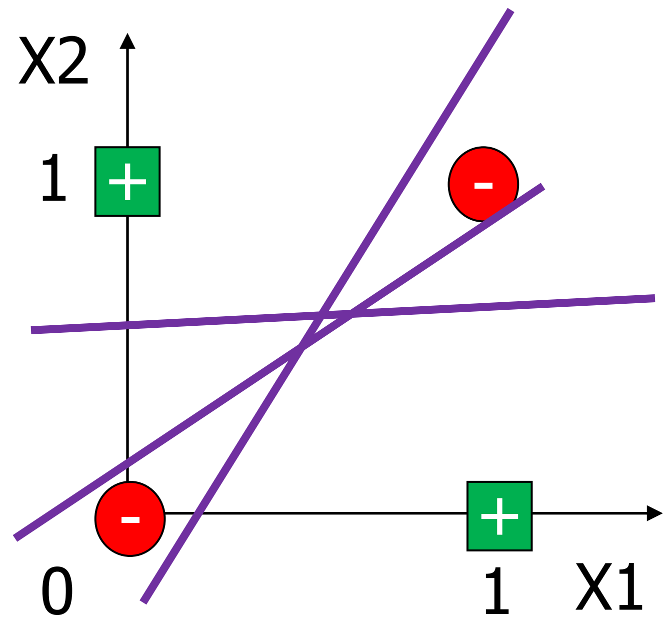
y = mx + b



Hence, can only classify examples if a "line" (hyerplane) can separate them

The XOR Problem

	Input	Output
a)	0 0	-
b)	0 1	+
c)	1 0	+
d)	1 1	-



Not linearly separable!!
Can't correctly classify

Perceptron Convergence Theorem

Perceptron ≡ no Hidden Units

Can prove that weights will converge if

- The set examples is learnable, the perceptron training rule will eventually find the necessary weights
- Learning rate is sufficiently small

Gradient Descent

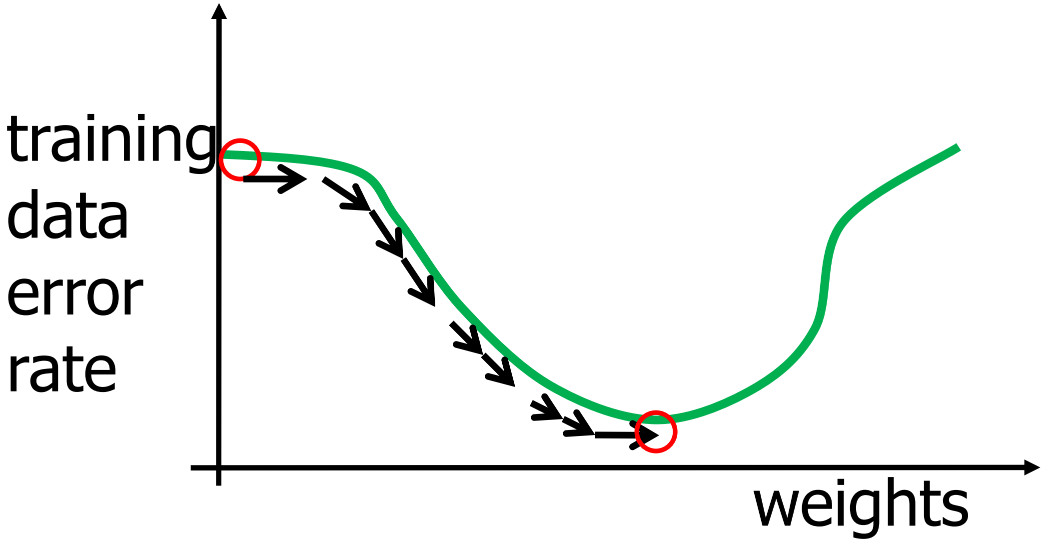
- Recall: $0 = w_0 + w_1x_1 + \ldots + w_nx_n$
- Question: How do we handle noise?
- Idea: Minimize squared error:

$$E(w) = ½ \Sigma (t_i - o_i)^2$$

- Note: i ranges over data points

Perceptron Gradient Descent

- Hypothesis space: Set of weights
- Goal: Minimize the classification error on the training data
- Perceptron does gradient descent to find weights



Derivation

$$\text{Error} \equiv \tfrac{1}{2} * \Sigma(\, t - o \,)^2$$

Network's output

Teacher's answer

$$\Delta W_j \equiv \quad - \eta \; \frac{\partial E}{\partial W_i}$$

$$\frac{\partial E}{\partial W_i} = \Sigma(t-o)\,\frac{\partial\,(t-o)}{\partial W_i} = -\,\Sigma(t-o)\;\frac{\partial o}{\partial W_i}$$

Remember: $o = \vec{W} \cdot \vec{X}$

Continuation of Derivation

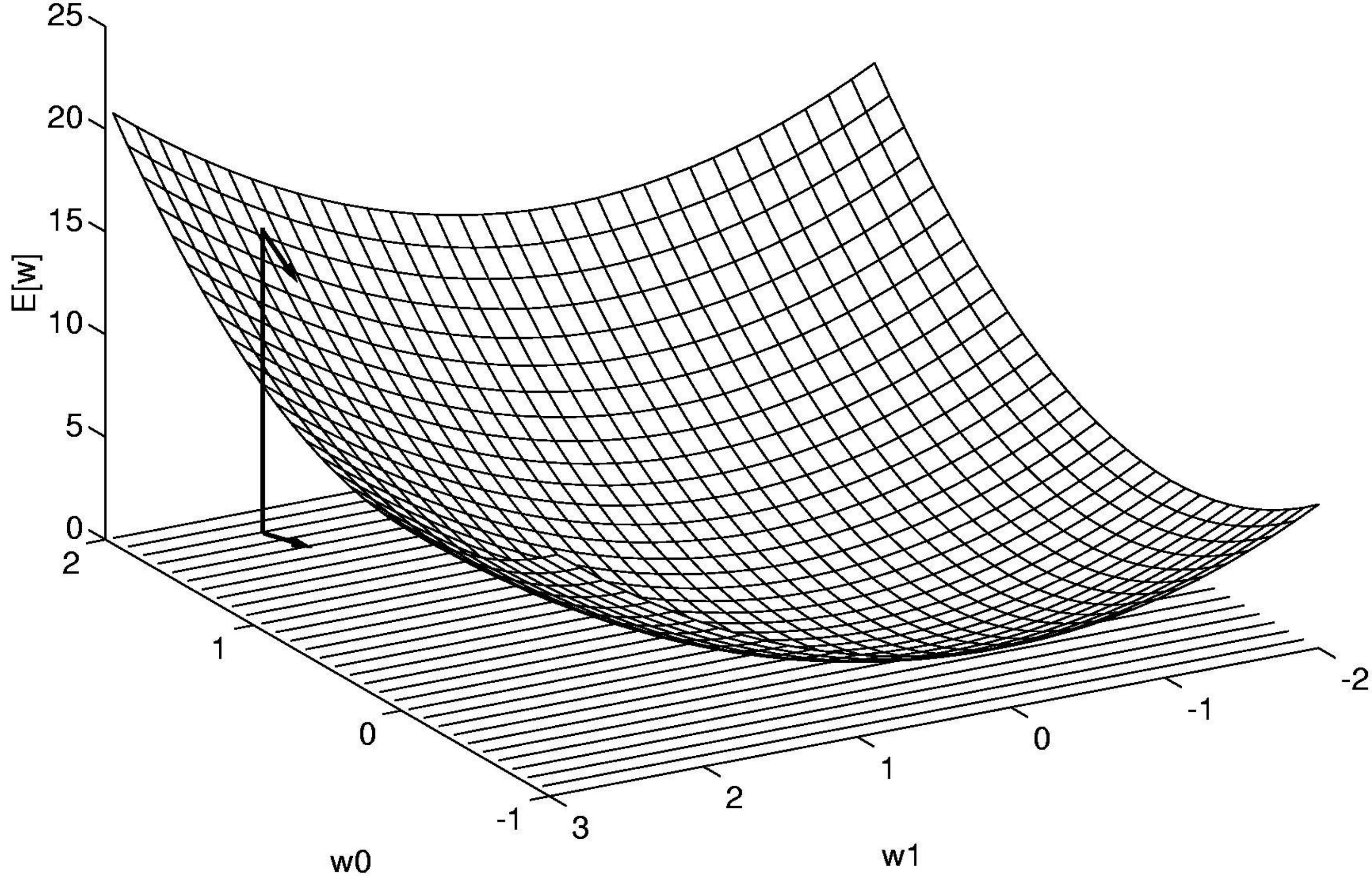
$$\frac{\partial E}{\partial W_i} = -\Sigma(t-o)\frac{\partial(w_i * x_i)}{\partial W_i}$$

Stick in formula for output

$$= -\Sigma(t-o)\, x_i$$

So $\boxed{\Delta W_i = \eta \, \Sigma \, (t-o)\, x_i}$ **The Delta Rule**

Gradient Descent



Batch vs. Incremental Gradient Descent

Batch Mode Gradient Descent:

Do until convergence

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

Incremental Mode Gradient Descent:

Do until convergence

For each training example d in D

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

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

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

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

Perceptron Performance

- Linear threshold functions are restrictive (high bias) but still reasonably expressive; more general than:
  - Pure conjunctive
  - Pure disjunctive
  - M-of-N (at least M of a specified set of N features must be present)
- In practice, converges fairly quickly for linearly separable data.
- Can effectively use even incompletely converged results when only a few outliers are misclassified.
- Experimentally, Perceptron does quite well on many benchmark data sets.

Summary

Perceptron training rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rule uses gradient descent

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

Perceptron Limits

- System obviously cannot learn concepts it cannot represent
- Minksy and Papert (1969) wrote a book analyzing the perceptron and demonstrating many functions it could not learn
- These results discouraged further research on neural nets; and symbolic AI became the dominate paradigm

Naïve Bayes Revisited

- Perceptrons are the simplest neural network
  - Its output is just a function of weighted sum of its inputs

- Perceptrons and logisitic regression are basically the same [see new Mitchell chapter]
  - Several variants of each
  - Similar to SVMs [covered later]

Naïve Bayes and Perceptrons

$$X_{f,v} = \begin{cases} 1 \text{ if feature f has value v} \\ 0 \text{ otherwise} \end{cases}$$

Also note that: $a^0=1$, $a^1=a$

$P(f = v) = P(f = v \mid +) * P(+) + P(f = v \mid -) * P(-)$

(assuming discrete-valued features)

Naïve Bayes and Perceptrons

Test example feature vector

Clever trick: Multiply all conditional probabilities

$$P(+ \mid X_{f,v}) = \frac{\Pi_f \, \Pi_v \, P(f{=}v \mid +)^{X_{f,v}} * P(+)}{[\Pi_f \, \Pi_v \, P(f{=}v \mid +)^{X_{f,v}} * P(+)] + [\Pi_f \, \Pi_v \, P(f{=}v \mid -)^{X_{f,v}} * P(-)]}$$

$$= \frac{1}{1 + \dfrac{[\Pi_f \, \Pi_v \, P(f{=}v \mid -)^{X_{f,v}} * P(-)]}{[\Pi_f \, \Pi_v \, P(f{=}v \mid +)^{X_{f,v}} * P(+)]}}$$

Naïve Bayes and Perceptrons

Note: $\Pi_f \, \Pi_v \, P(f{=}v \mid -)^{X_{f,v}} = e^{\Sigma_{f,v} \log[P(f = v \mid -)^{X_{f,v}}]}$

Rewrite Naïve Bayes equations as:

$$P(+ \mid X_{f,v}) = \frac{1}{1 + \dfrac{[P(-)]}{[P(+)]} \, e^{\Sigma_{f,v} \, \mathrm{Log} \frac{[P(f = v \mid -)]^{X_{f,v}}}{[P(f = v \mid +)]^{X_{f,v}}}}}$$

Naïve Bayes and Perceptrons

weights

$$-W_{f,v} = \text{Log}\ \frac{[P(f = v\ |-)]}{[P(f = v\ |+)]}$$

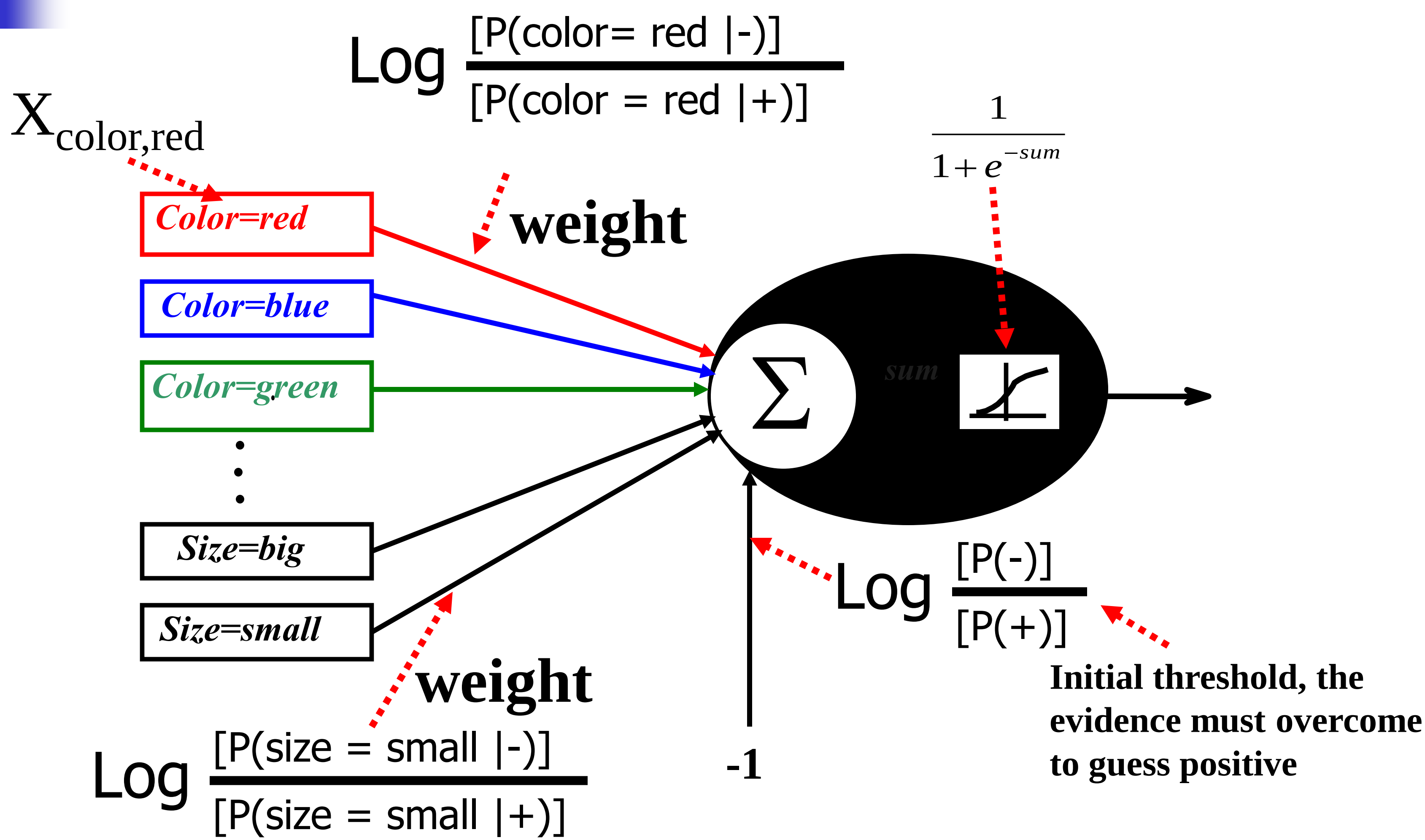
$$-\Theta = \text{Log}\ \frac{[P(-)]}{[P(+)]}$$

Bias/Threshold

Greater ≥ 0.5 if weighted sum of features greater than Θ

$$P(+\ |\ X_{f,v}) = \frac{1}{1 + e^{-[(\Sigma_{f,v}\ w_{f,v} * x_{f,v}) - \Theta]}}$$

Example Encoding of Naïve Bayes as a Perceptron



Naïve Bayes vs. Perceptron

Implement NB as a perceptron w/*sigmoidal* output

- Weights and bias are set by equations on previous slides

Note: Perceptron learner may pick different weights

- Representation is same for NB and perceptron
- Learning algorithm is different
- Many equivalent scoring hypothesis may exist [i.e., separating hyperplanes]

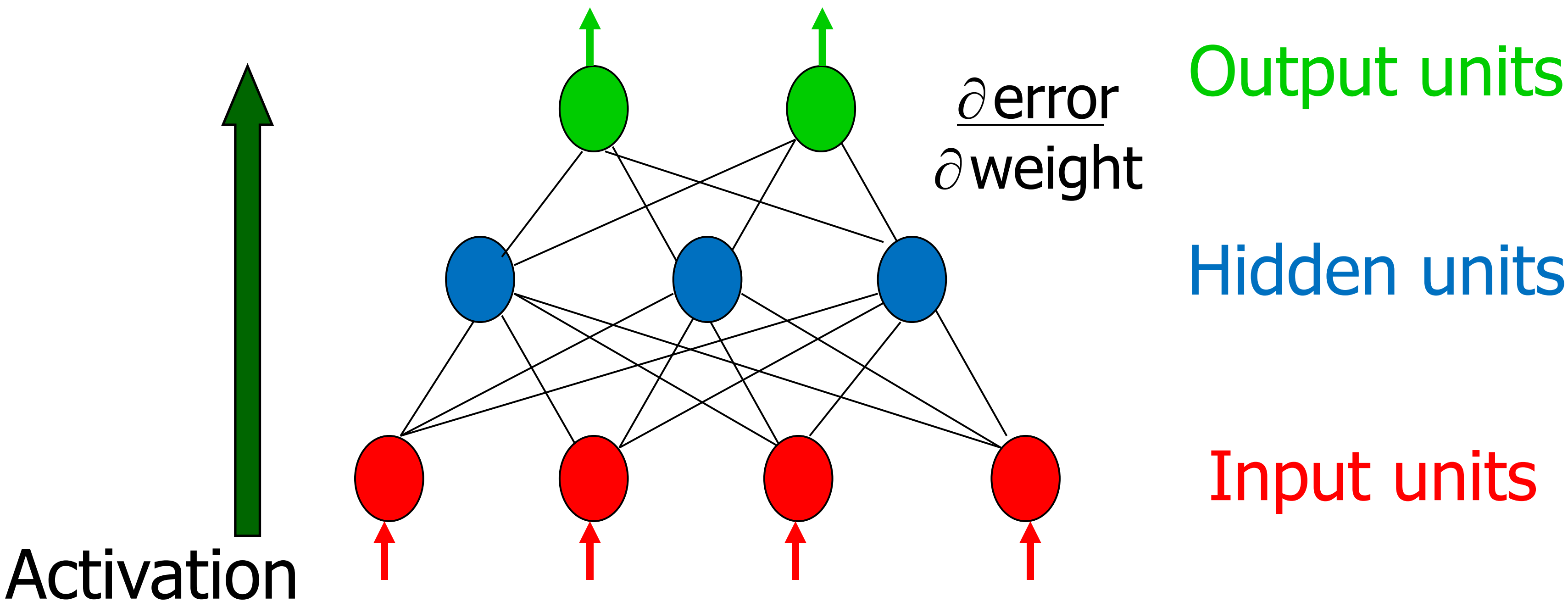
Outline

- Homework 1 review
- Perceptron
- Multilayer neural networks

Multi-Level Neural Networks

- Neural Networks can represent complex decision boundaries
  - Variable size
  - Deterministic
  - Continuous Parameters
- Learning Algorithms for neural networks
  - Local Search. The same algorithm as for sigmoid threshold units
  - Eager
  - Batch or Online

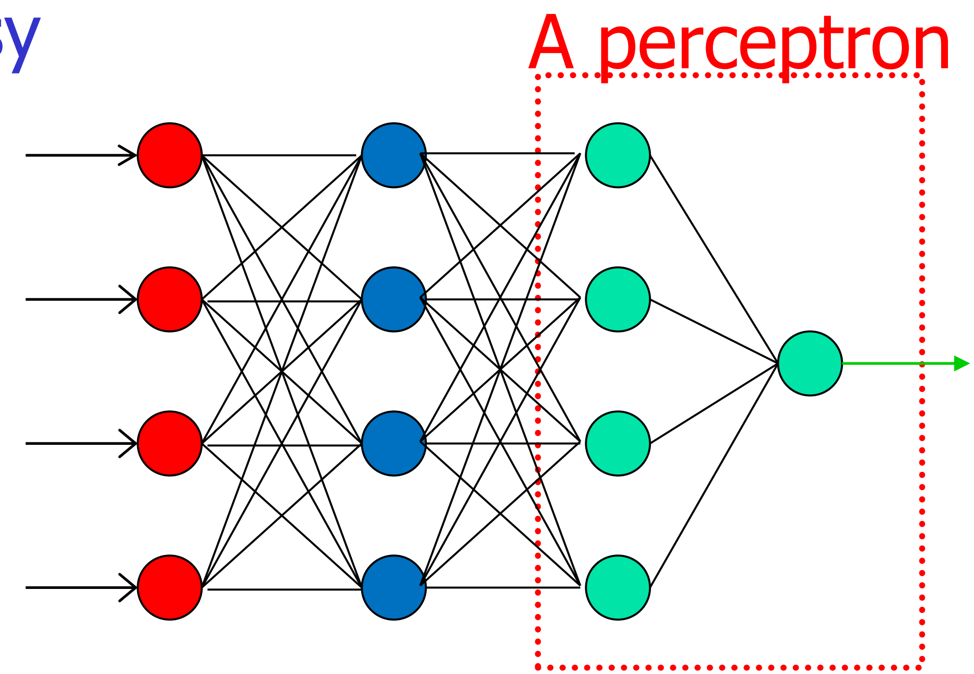
Multi-Level Neural Networks



Hidden Units

One View

Allow a system to create its own internal representation – for which problem solving is easy



Multi-Level Neural Networks

- A typical multi-layer network consists of:
  - Input
  - Hidden
  - Output
  - Typically, each layer is fully connected to next layer
- Activation of neurons feeds forward
- Usually, the network structure (units and interconnections) is specified by the designer
- Learning problem: Find a good set of weights

Multi-Level Neural Networks

- Multi-layer networks can represent arbitrary functions
  - One layer with enough *hidden units* (possibly 2^N for Boolean functions), can record input
  - Single hidden layer: Compute any Boolean function
- The weights determine the function compute
- An effective learning algorithm for such networks was thought to be difficult

Question: How to provide an error signal to the interior units?

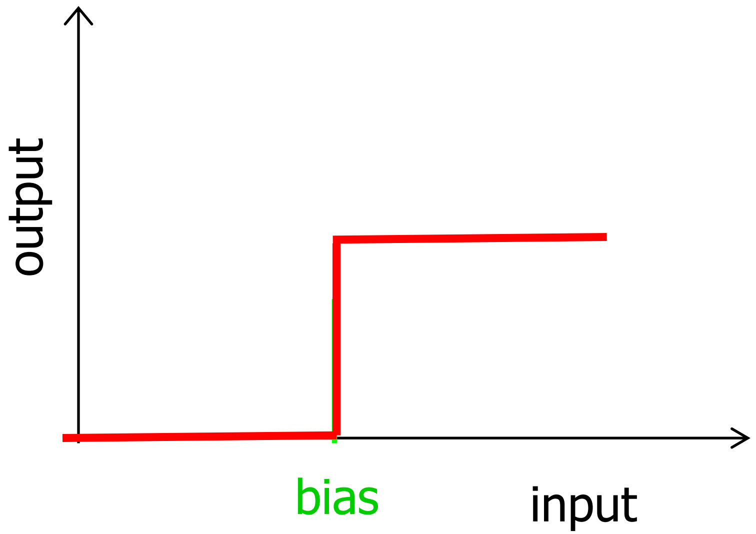
Idea: Still Use Gradient Descent

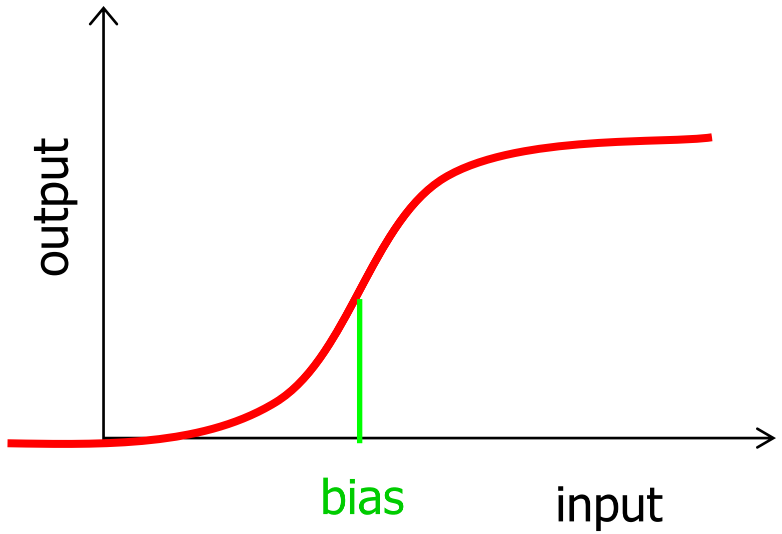
- Despite limitations, gradient descent is works well in practice
- How can we apply it to a multi layer network?
- Gradient descent requires output of a unit to be a differentiable function
- Linear threshold function is not differentiable, so we'll use the sigmoid function

Sigmoid Function

$$\text{output}_j = F(\sum_i \text{weight}_{i,j} \times \text{output}_i)$$

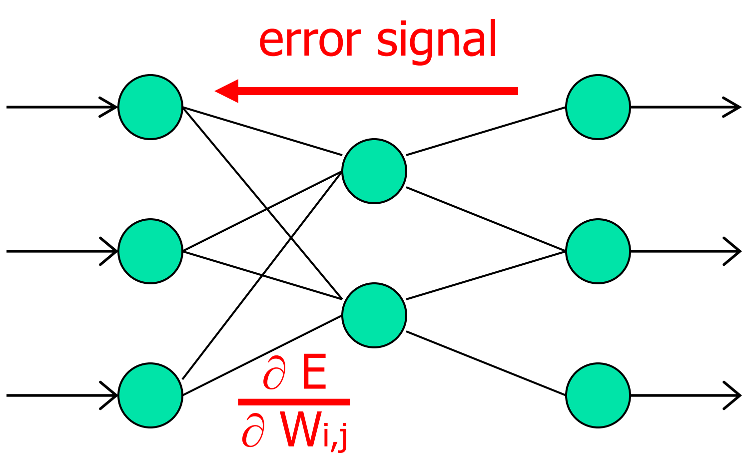
$$F(\text{input}_i) = \frac{1}{1+e^{-(\text{input}_i - \text{bias}_i)}}$$





Backpropogation

- Backpropagation generalizes the perceptron rule
  - Derivation involves partial derivatives
- Rumelhart, Parker, and Le Cun (and Bryson & Ho, 1969 + Werbos, 1974) independently developed (1985) a technique for learning weights of hidden units



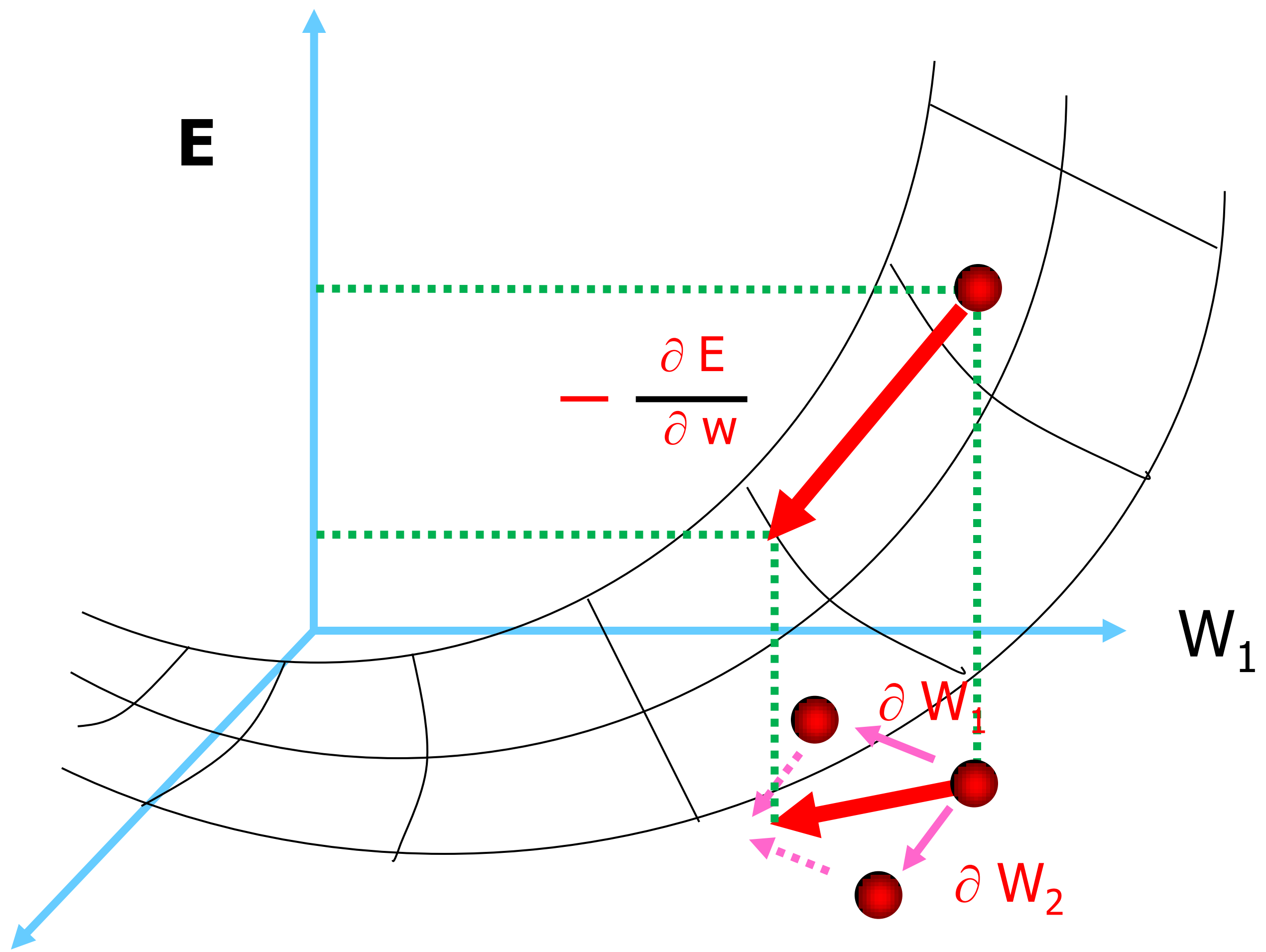
WARNING!

Calculus / Linear Algebra Ahead!!!

Weight Space

- Given a neural-network layout, the weights are free parameters that define a *space*
- Each point in this *Weight Space* specifies a network
- Associated with each point is an *error rate*, **E**, over the training data
- Backprop performs gradient descent in weight space

Gradient Descent Weight Space



Gradient Descent Rule

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \cdots \cdots \cdots, \frac{\partial E}{\partial w_N} \right]$$

Gradient: N+1 dimensional vector (slope in weight space)

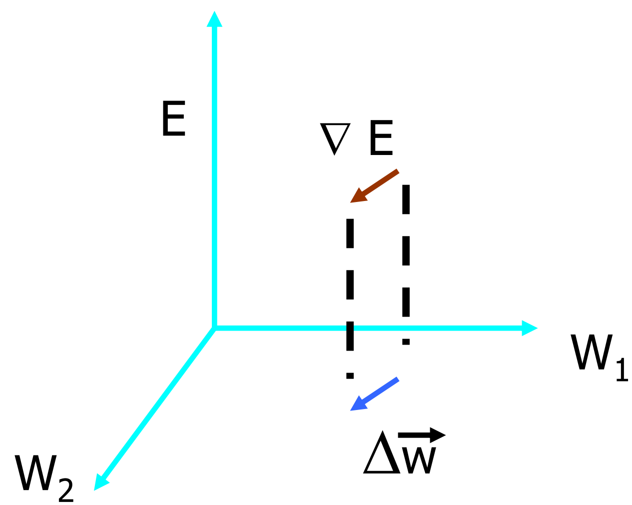
Goal: **Reduce** errors

How: Go "down hill"

Take a finite step in weight space:

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

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



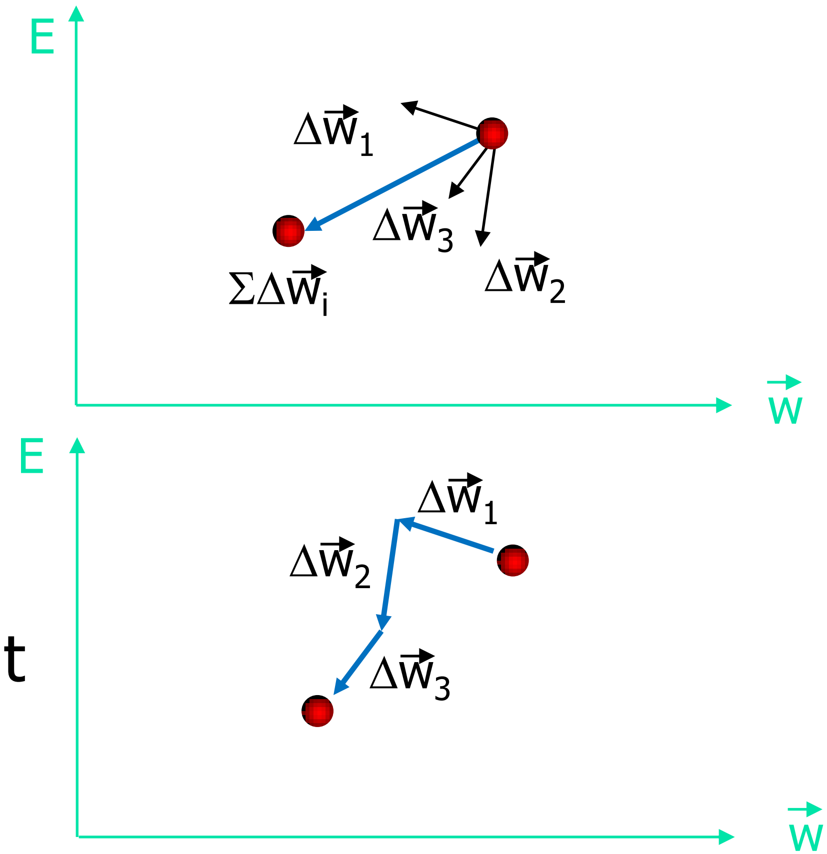
Online vs. Batch Gradient Descent

- Technically, we should look at the error gradient for the entire training set, before taking a step in weight space ("batch" Backprop)
- *However*, as presented, we take a step after each example ("on-line" Backprop)
  - Much faster convergence
  - Can reduce overfitting (since on-line Backprop is "noisy" gradient descent)

Online vs. Batch Gradient Descent

<u>BATCH</u> – add Δw vectors for <u>every</u> training example, <u>then</u> 'move' in weight space.

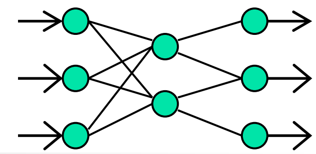
<u>ON-LINE</u> – "move" after <u>each</u> example (aka, *stochastic* gradient descent)



* Final locations in $\vec{w}$ space need not be the same for BATCH and ON-LINE

* Note $\Delta w_{i,BATCH} \neq \Delta w_{i,\ ON\text{-}LINE}$, for $i > 1$

BP Calculations



Assume one layer of hidden units (std. topology)

1. Error $\equiv \frac{1}{2} \Sigma$ ($\text{Teacher}_i - \text{Output}_i$) ^{2}
2. $= \frac{1}{2} \Sigma(\text{Teacher}_i - F([\Sigma W_{i,j} \times \text{Output}_j])^2$
3. $= \frac{1}{2} \Sigma(\text{Teacher}_i - F([\Sigma W_{i,j} \times F(\Sigma W_{j,k} \times \text{Output}_k)]))^2$

<u>Determine</u>

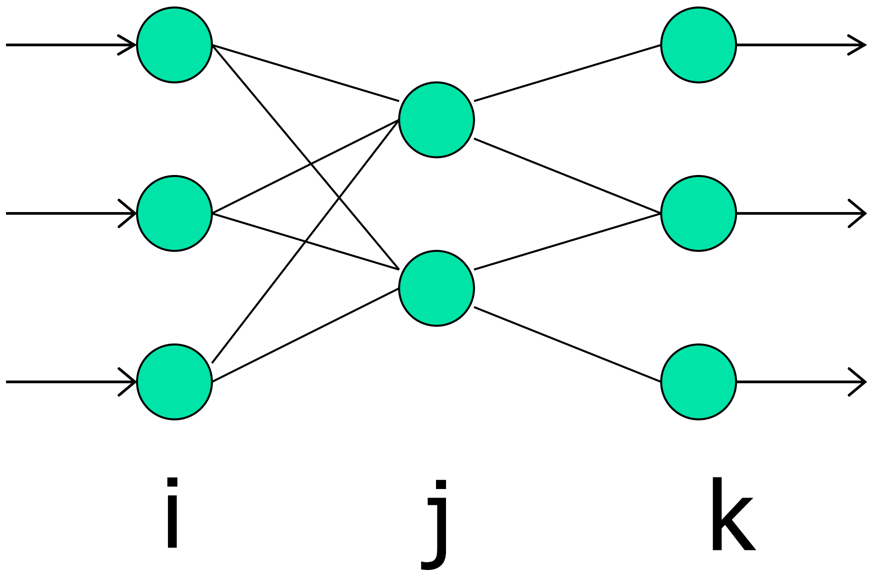
$$-\frac{\partial \text{ Error}}{\partial \text{ Wj,k}} =$$ (use equation 2)

$$-\frac{\partial \text{ Error}}{\partial \text{ net}_j} =$$ (use equation 3)

* See Table 4.2 in Mitchell for results

Recall: $\boxed{\Delta w_{i,j} = - \eta \, (\partial E / \partial w_{i,j}) \, x_{i,j}}$

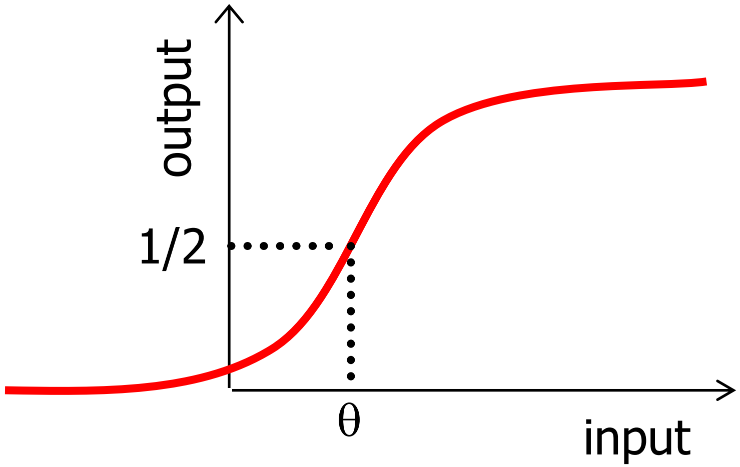
Terminology



net_j sum of weight inputs to j

Differentiating the Sigmoid

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



$$\frac{\partial\, \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

Update: Output Units

$$\frac{\partial\, Error}{\partial\, W_{j,k}} = \frac{\partial\, Error}{\partial\, o_k} \frac{\partial\, o_k}{\partial\, W_{j,k}}$$

$$\frac{\partial\, Error}{\partial\, o_k} = ½\, \Sigma(t_a - o_a)^2$$

$$= ½\, (t_k - o_k)^2$$

$$= ½\, 2(t_k - o_k) \frac{\partial\, (t_k - o_k)}{\partial\, o_k}$$

$$= -(t_k - o_k)$$

Update: Output Units

$$\frac{\partial \text{ Error}}{\partial W_{j,k}} = \frac{\partial \text{ Error}}{\partial o_k} \frac{\partial o_k}{\partial W_{j,k}}$$

$$\frac{\partial \text{ Error}}{\partial o_k} = -(t_k - o_k)$$

$$\frac{\partial o_j}{\partial W_{j,k}} = o_k(1 - o_k)$$

$$\frac{\partial \text{ Error}}{\partial W_{j,k}} = -o_k(1 - o_k)(t_k - o_k)$$

$$\boxed{\Delta w_{j,k} = - \eta \; (\partial E / \partial w_{j,k}) \; x_{j,k}}$$

Update: Hidden Units

$$\frac{\partial\, \text{Error}}{\partial\, \text{net}_j} = \sum \frac{\partial\, \text{Error}}{\partial\, \text{net}_k} \frac{\partial\, \text{net}_k}{\partial\, \text{net}_j}$$

$$= \sum -\delta_k \frac{\partial\, \text{net}_k}{\partial\, \text{net}_j}$$

$$= \sum -\delta_k \frac{\partial\, \text{net}_k}{\partial\, o_j} \frac{\partial\, o_j}{\partial\, \text{net}_j}$$

$$= \sum -\delta_k\, w_{j,k} \frac{\partial\, o_j}{\partial\, \text{net}_j}$$

$$= \sum -\delta_k\, w_{j,k} o_j (1 - o_j)$$

$$= -o_j (1 - o_j) \sum \delta_k\, w_{j,k}$$

Backpropagation Algorithm

Set initial weights to random value

Repeat

for each example X_j

1. compute current output o_j
2. For each output unit k: $\delta_k = o_k(1- o_k)\ (t_k - o_k)$
3. For each hidden unit h: $\delta_h = o_h(1- o_h)\ \Sigma_k\ w_{i,j}\delta_k$
4. Update each network weight $w_{i,j}$

$$w_{i,j} = w_{i,j} + \Delta w_{i,j}\ \eta$$

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

Until train set error rate is small enough

Notes

- Initiate weights & bias to small random values for example in [-0.3, 0.3]

- Randomize order of training examples

- Propagate activity <u>forward</u> to output units

$$\boxed{out_j = F(\Sigma\, w_{i,j} \text{ x } out_i)}$$

- Measure accuracy on test set to estimate *generalization* (future accuracy)

Learning Rate

- This is a subtle art
  - Too small: Days instead of minutes to converge
  - Too large: Diverges (MSE gets larger and larger while the weights increase and usually oscillate)
  - The learning rate influences the ability to escape local optima
- Very often, different learning rates are used for units in different layers
- Each unit has its own optimal learning rate
- The -just right value is hard to find

Adjusting η on-the-Fly

0. Let η = 0.25
1. Measure ave. error over *k* examples

   - call this $\mathbf{E}_{\mathbf{before}}$
2. Adjust wgts according to neural-net learning algorithm being used
3. Measure ave error on <u>same</u> *k* examples

   - call this $\mathbf{E}_{\mathbf{after}}$

Adjusting η (cont.)

4. If $E_{after} > E_{before,}$
   then $\eta \leftarrow \eta * 0.99$
   else $\eta \leftarrow \eta * 1.01$

5. Go to 1

Note: *k* can be all training examples
but could be a subset

Including a "Momentum" Term in Backprop

To speed up convergence, often another term is added to the weight-update rule

$$\Delta W_{i,j}(t) = \frac{-\eta \partial E}{\partial W_{i,j}} + \beta \underbrace{\Delta W_{i,j}(t-1)}_{\text{The previous change in weight}}$$

Typically, $0 < \beta < 1$

Online, Batch and Momentum

The "momentum term" variant of backprop can be written as

The weights at time *t*

$$\Delta \overrightarrow{w_t} = -\eta \sum_{i=0}^{t} \beta^i \nabla \overrightarrow{w_{t-i}}$$

So we're doing an "exponentially decaying" weighted sum of the individual gradients

Sort of a cross between <u>pure batch</u> & <u>pure on-line</u>

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

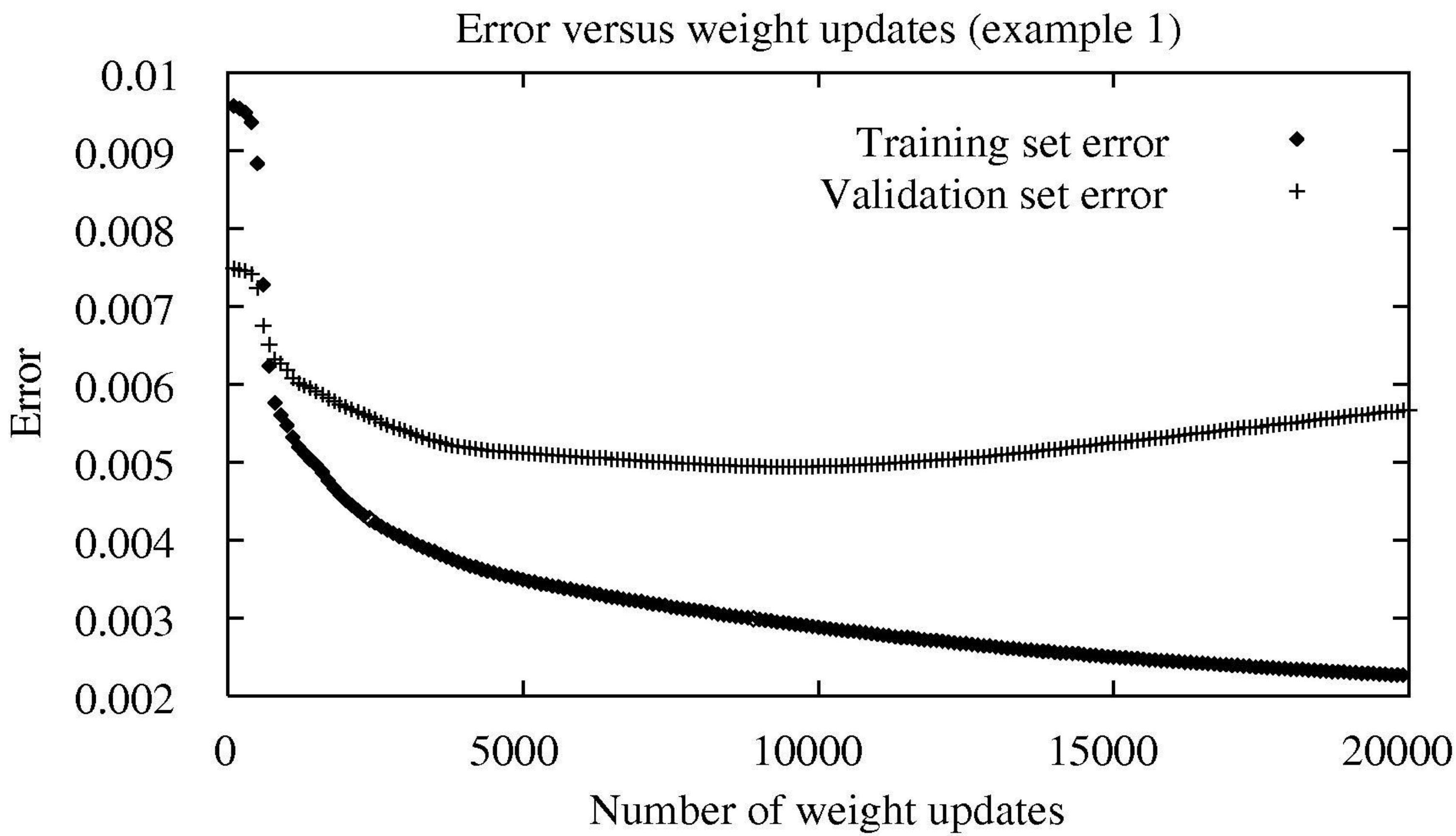
Continuous functions:

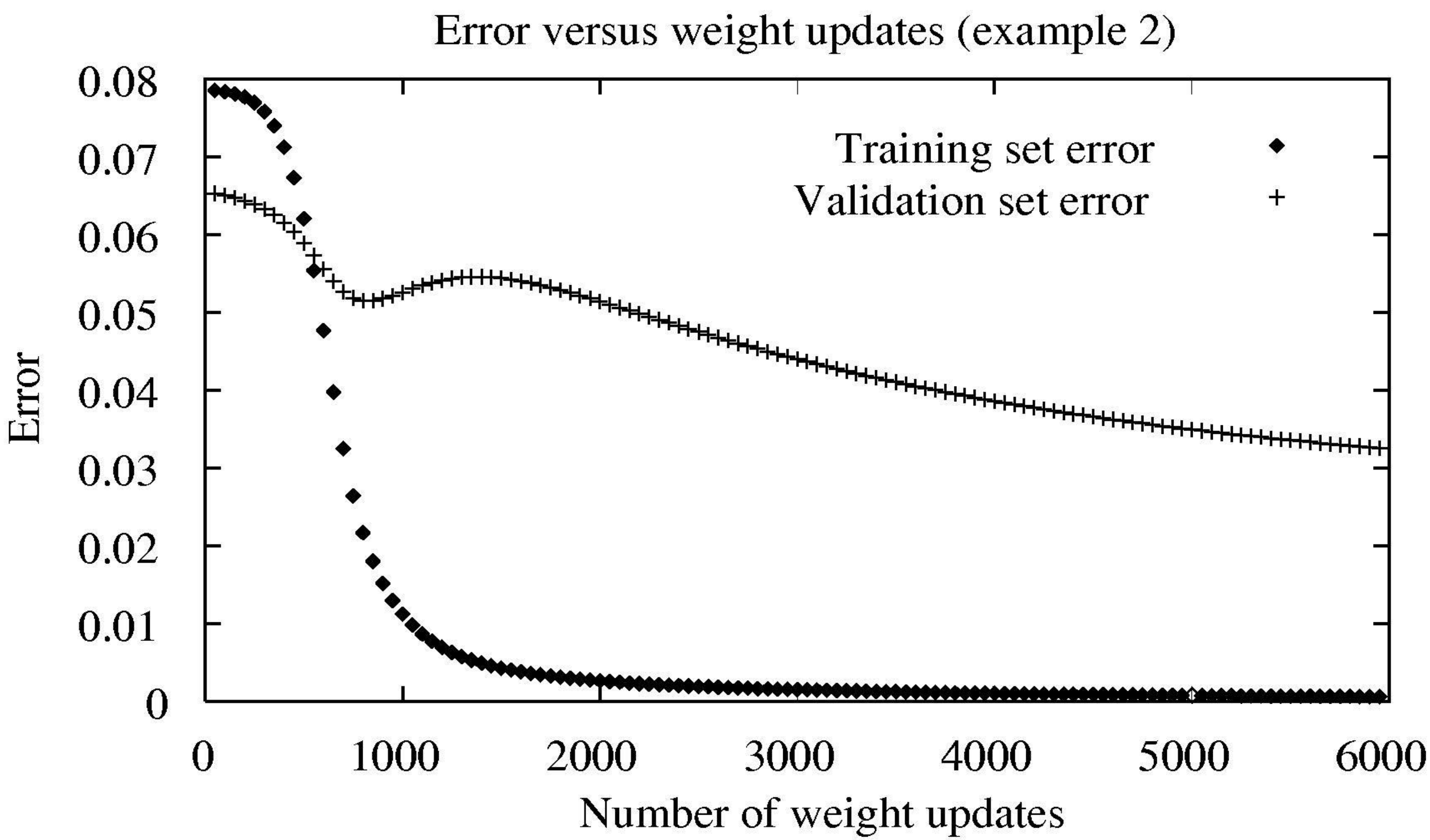
- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers

Design Choices

- Overfitting: too many parameters compared to the amount of data available
- Choosing the number of hidden units:
  - Too few do not allow the concept to be learned
  - Too many lead to slow learning and overfitting
  - n binary inputs: log n is a good heuristic choice
- Choosing the number of layers
  - Always start with one hidden layer
  - Never go beyond 2 hidden layers, unless the task structure suggests something different

Overfitting in Neural Nets




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

Overfitting Avoidance

Penalize large weights:

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

Train on target slopes as well as values:

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

Weight sharing

Early stopping

Interpretability

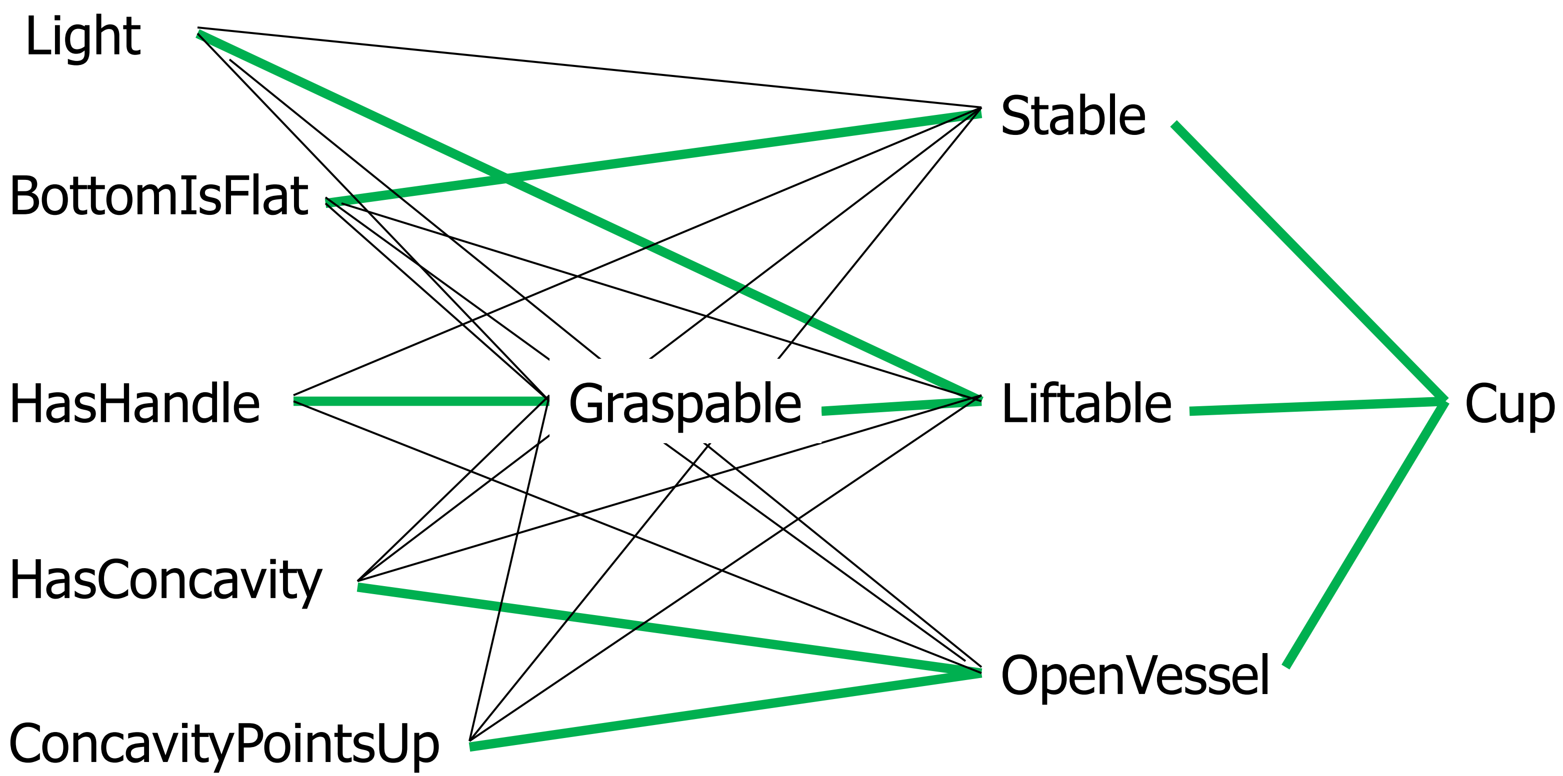
- Multilayer neural networks are difficult for humans to understand

- One idea:
  - Attempt to extract a decision or rule set from a learned neural network
  - Train the decision tree to mimic the decisions made by the learned neural network
  - Present decision tree to user

KBANN: Incorporating Background Knowledge

- Cup $\leftarrow$ Stable, Liftable, OpenVessel
- Stable $\leftarrow$ BottomIsFlat
- Liftable $\leftarrow$ Graspable, Light
- Graspable $\leftarrow$ HasHandle
- OpenVessel $\leftarrow$ HasConcavity,ConcavityPointsUp

Knowledge: Network Topology



Application: Face Recognition

- **Given:** Sets of photos
- **Task:** Recognize DIRECTION of face
- **Framework:** Different people, poses, "glasses", different

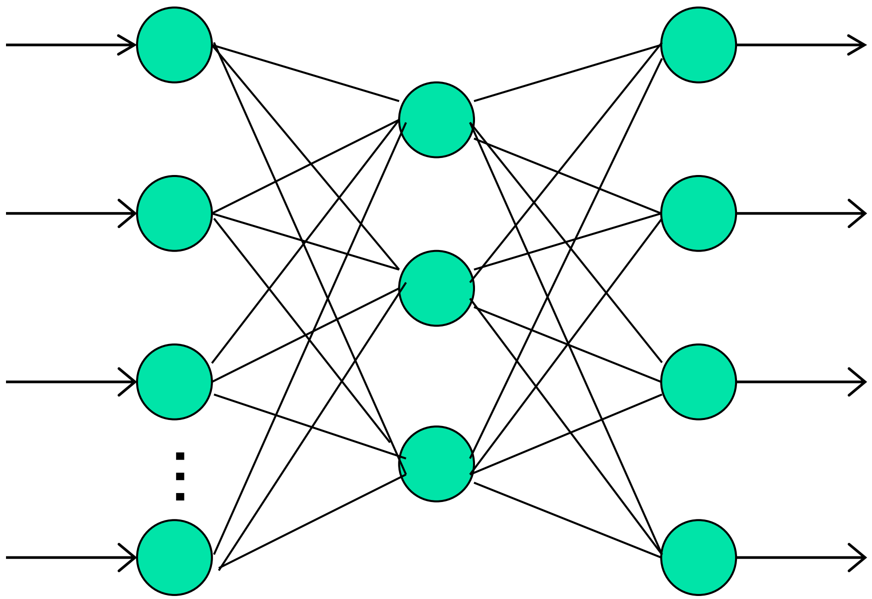
Design Decision

- **Input Encoding:**
  - Just pixels? (subsampled? averaged?)
  - or perhaps lines/edges?
- **Output Encoding:**
  - Single output ([0, 1/n] = #1, . . .)
  - Set of n-output (take highest value)
- **Network structure: # of layers**
  - Connections (training time vs accuracy)
- **Learning Parameters: Stochastic?**
  - Initial values of weights?
  - Learning rate h, Momentum a, . . .
  - Size of Validation Set, . . .

Images

Subsample: 30 x 32 pixels: 4x4 blocks get mean activation and normalize [0,1]

Network Structure



0.1: Inactive
0.9: Active

When Use A Neural Network

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete, real valued, or a vector of values
- Possibly noisy data
- Training time is unimportant
- Form of target function is unknown
- Human readability of result is unimportant
- Output computation has to be fast

Next Class

- Model Ensembles [Dietterich, AI Magazine article, section 2 only]

- Genetic algorithms [read Mitchell, Chapter 9]

Summary

- Peceptrons: Linear decision boundary
- Multilayer networks: Very expressive
- Learning: Find weights
  - Gradient descent
  - Backpropagation

Questions?