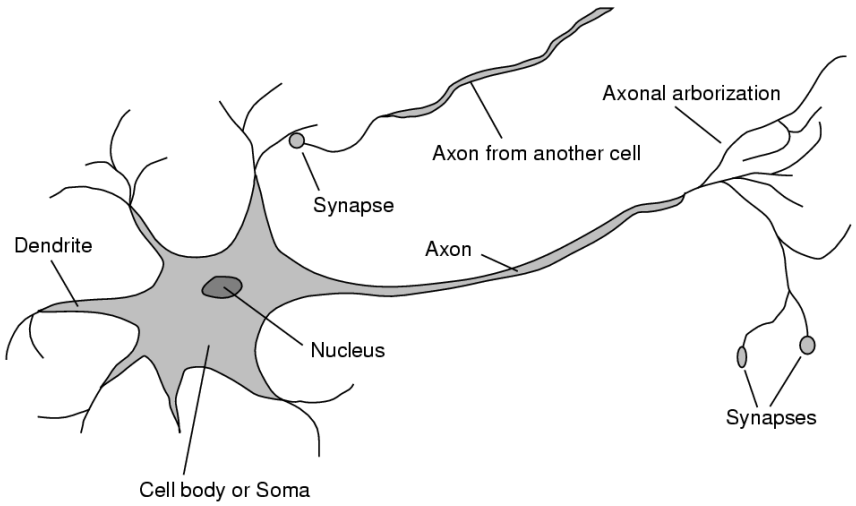
Neural Networks

CPSC 470 – Artificial Intelligence

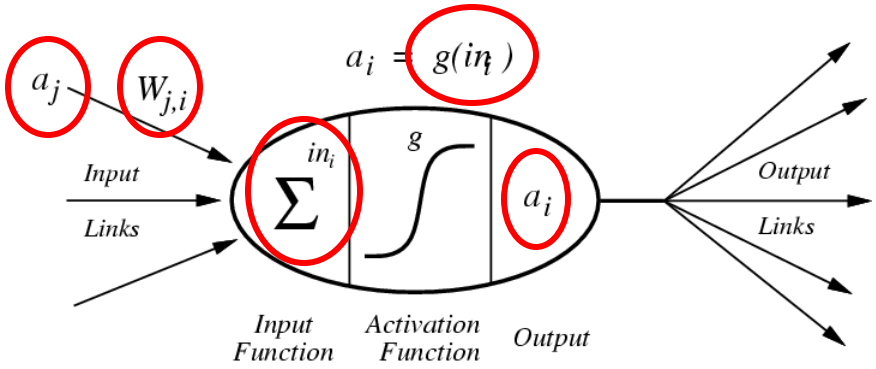
Brian Scassellati

Neural Networks



- Inspired by early models of biological neurons
- Ignore all the comparisons with biological neurons in your textbook
  - (They are misleading, and based on an outdated model of neural function)
- We will treat neural nets as a cool computational technique, but never as a model of biology

Simple Computing Elements



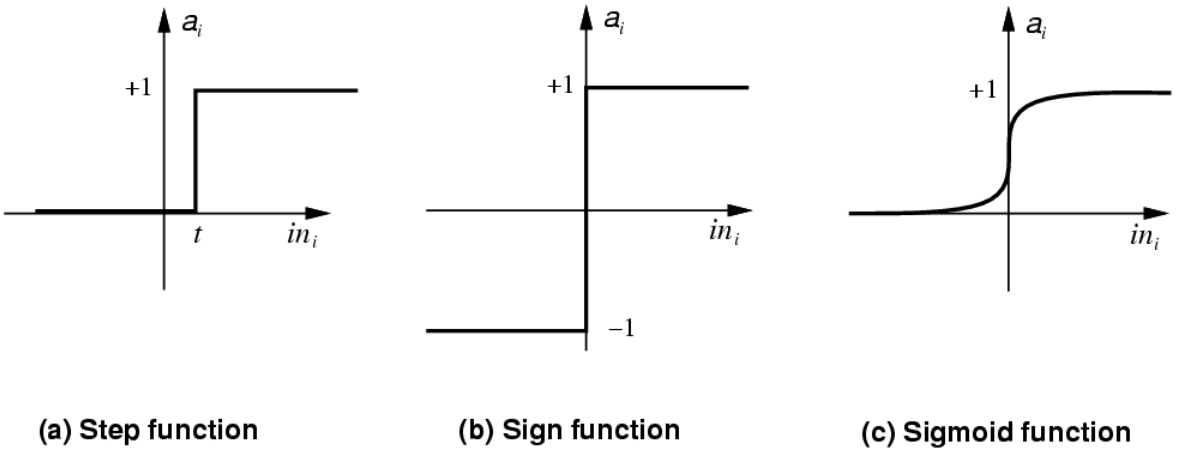
For neuron i :

$$in_i = \sum_j W_{j,i} a_j \quad \text{for all inputs j}$$

$$a_i = g(in_i)$$

- Each neuron has
  - A set of inputs
  - A weight associated with each input
  - A weighted sum of inputs
  - An activation function
  - An output that links to some number of other elements
- Output is the activation function applied to the weighted inputs

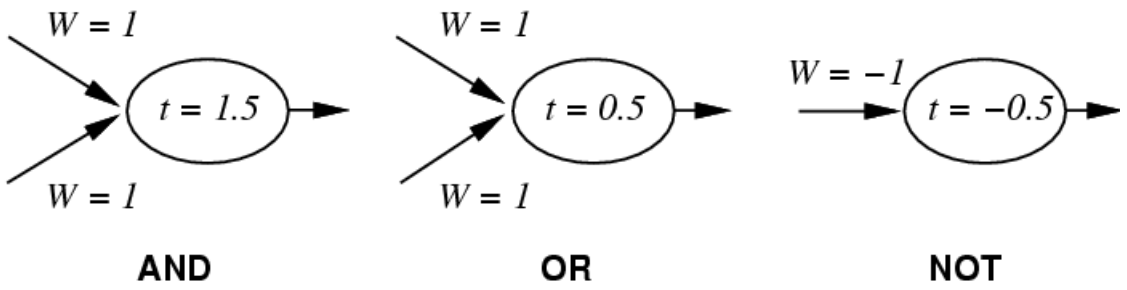
Activation Functions



(a) Step function (b) Sign function (c) Sigmoid function

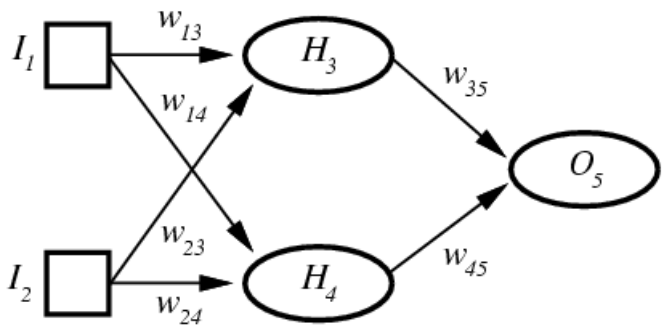
- Thresholds can be added by using a constant input (positive or negative)
  - Thus converting any step function to a sign function
- Sigmoid is most common activation function, as it avoids discontinuities (and has a useful derivative for gradient descent)

Representing Boolean Functions with Neurons



- Using step functions with the given thresholds
- If we can simulate any logic gate with these elements, we can build arbitrary computations from networks of these elements.

A Typical Network

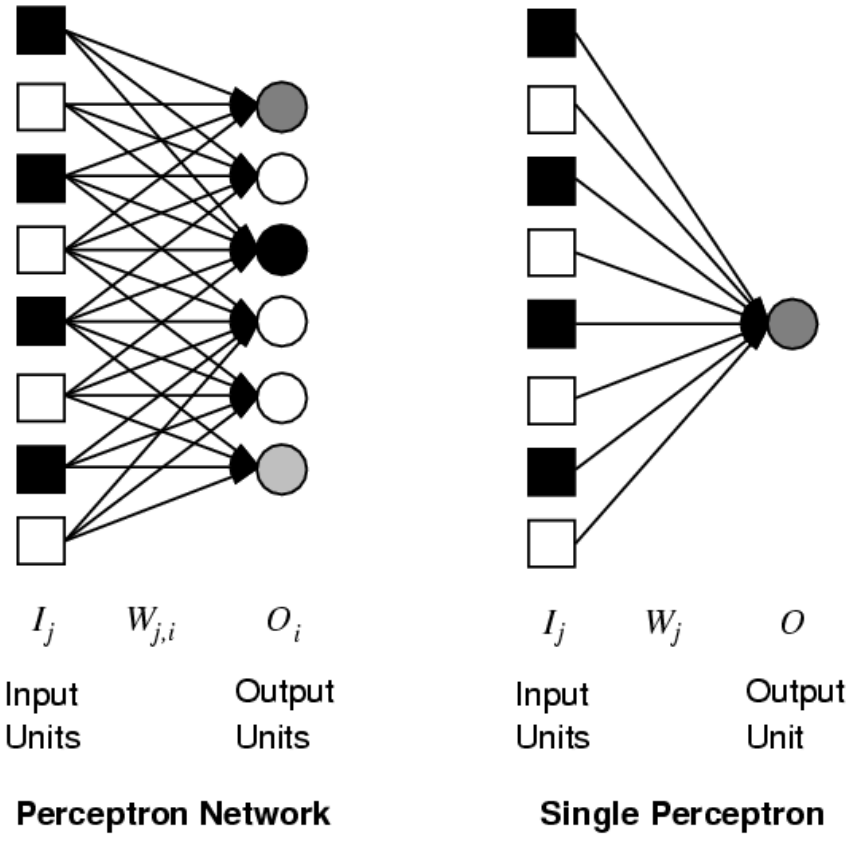


$$a_5 = g(W_{3,5}a_3 + W_{4,5}a_4)$$

$$a_5 = g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2))$$

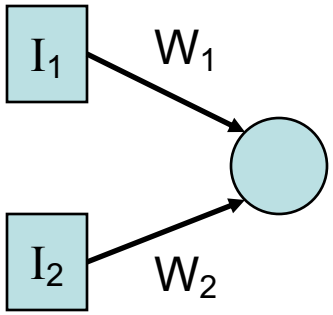
- Node types
  - Input nodes
  - Output nodes
  - Hidden nodes
- Network connections
  - Feed forward networks have no loops (they are directed acyclic graphs)
  - Recurrent networks allow for feedback loops

Perceptrons



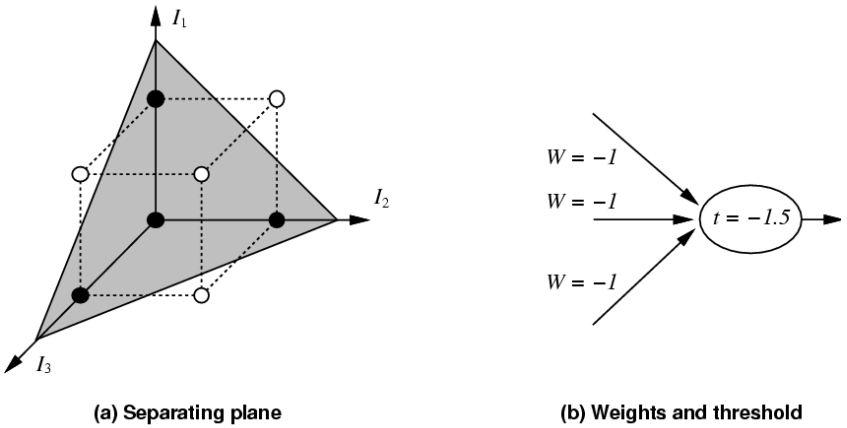
- Layered feed-forward networks
  - Output=Step(W*I)
  - Assume threshold=0 without loss of generality
- Majority function (output 1 if at least half the inputs are 1)
  - All weights are 1
  - Threshold is –n/2 for n inputs
  - (would have required a decision tree with 2^n nodes)
- Can we represent any Boolean function?

Linear Separability of Perceptrons



- Output is 1 if and only if
  $(W_1I_1+W_2I_2) > 0$
  $I_1 = -(W_2 / W_1)\ I_2$
- Separation between output states and input states is a line (it is linearly separable)
  - Adding a threshold will only allow for an offset of the line
- Some functions can be computed in this way (AND, OR)
- Some functions cannot (XOR)

Linear Separability in 3 Dimensions



(a) Separating plane

(b) Weights and threshold

- In three dimensions, linear separability is defined by a plane that separates positive from negative responses
- Example: Perceptron for computing the Minority function

Learning with Perceptrons

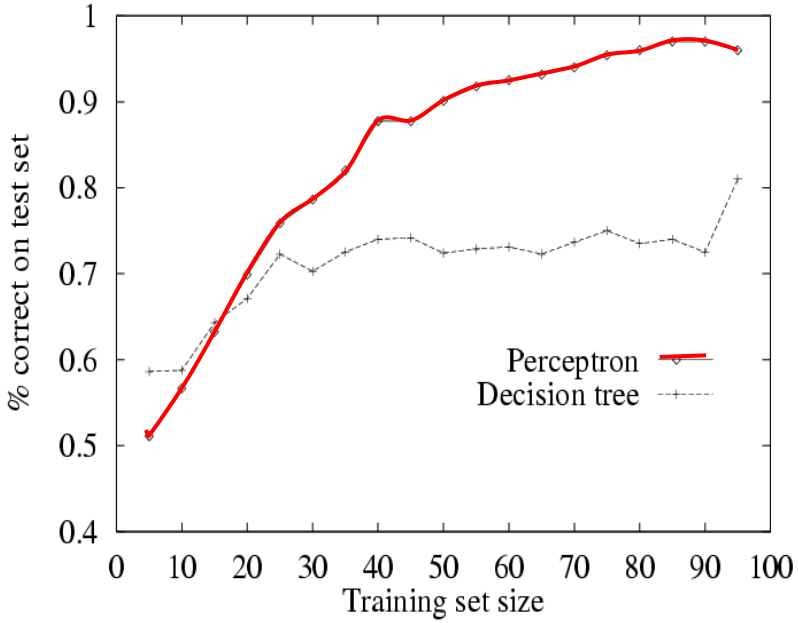
- At each time step, compute the error

  Err = CorrectOutput – ActualOutput
- Update each weight according to the learning rule:

  $W_j \leftarrow W_j + \alpha * I_j * Err$

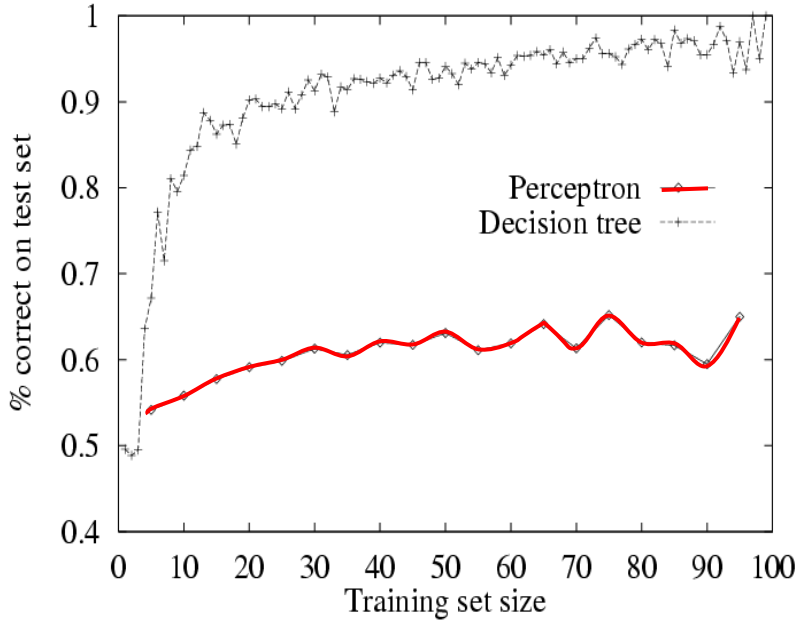
  where α is the learning rate
- Guaranteed to learn any linearly separable function given sufficient training examples

Which are better, Perceptrons or Decision Trees?



11-input majority function

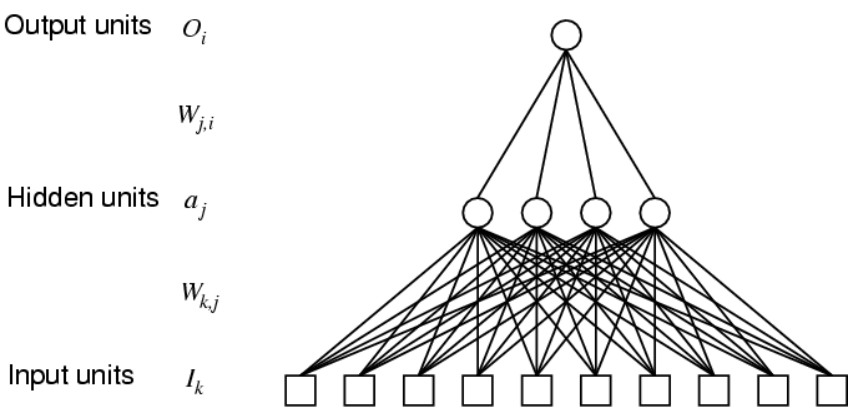
Perceptrons are better



WillWait restaurant example

Decision Trees are better

Multi-Layer Feed-Forward Networks



- Adding more layers (hidden units) allows us to compute more complex problems
- Can solve problems that are not linearly separable
- But how do we train these networks to do the right thing?

Back-propagation

function BACK-PROP-UPDATE(*network, examples*, α) **returns** a network with modified weights

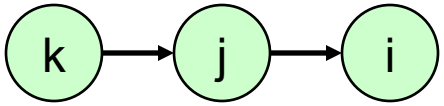
inputs: *network*, a multilayer network
examples, a set of input/output pairs
α, the learning rate

repeat
for each *e* **in** *examples* **do**
/* *Compute the output for this example* */
$\mathbf{O} \leftarrow$ RUN-NETWORK(*network*, $\mathbf{I}^e$)
/* *Compute the error and* Δ *for units in the output layer* */
$\mathbf{Err}^e \leftarrow \mathbf{T}^e - \mathbf{O}$
/* *Update the weights leading to the output layer* */
$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i^e \times g'(in_i)$
for each subsequent layer **in** *network* **do**
/* *Compute the error at each node* */
$\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i} \Delta_i$
/* *Update the weights leading into the layer* */
$W_{k,j} \leftarrow W_{k,j} + \alpha \times I_k \times \Delta_j$
end
end
until *network* has converged
return *network*

Run the network on an example

For output nodes, add to the weight the quantity (learningRate * priorNode * error * GradientOfActivationFunction)

Compute the error between the expected and the actual output

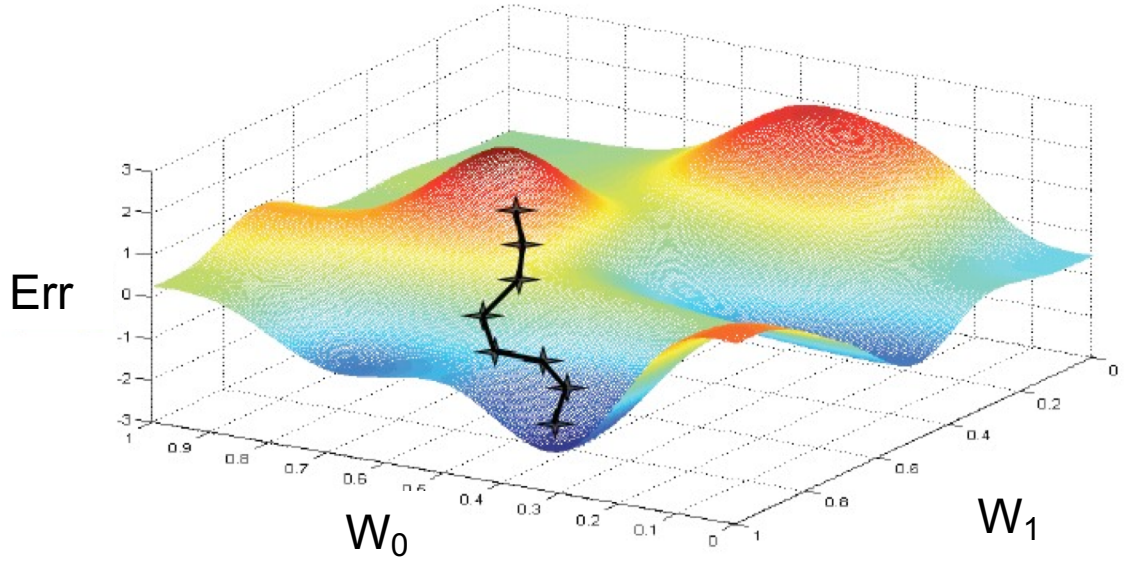


For internal node j, it is responsible for some fraction of the error at node i

Update the weights going into node j according to that fraction of the following error

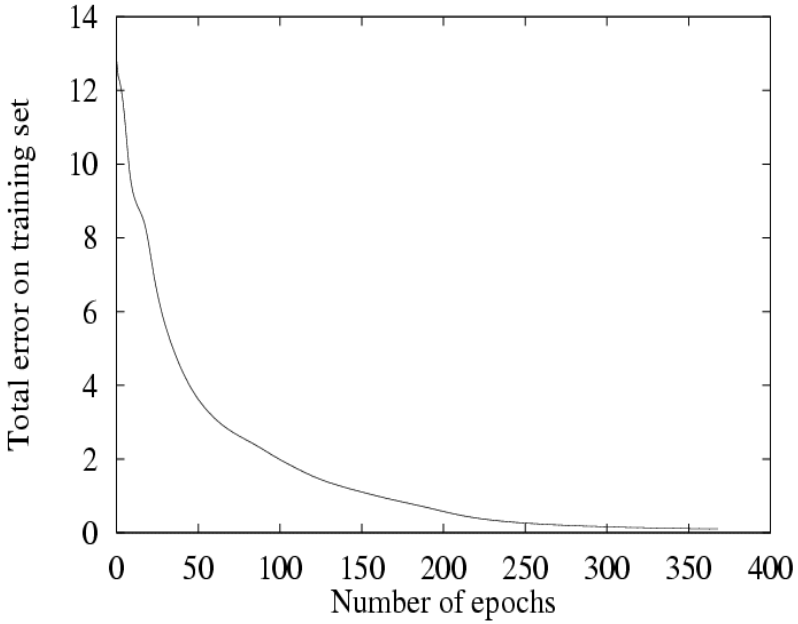
Errors propagate back from the output

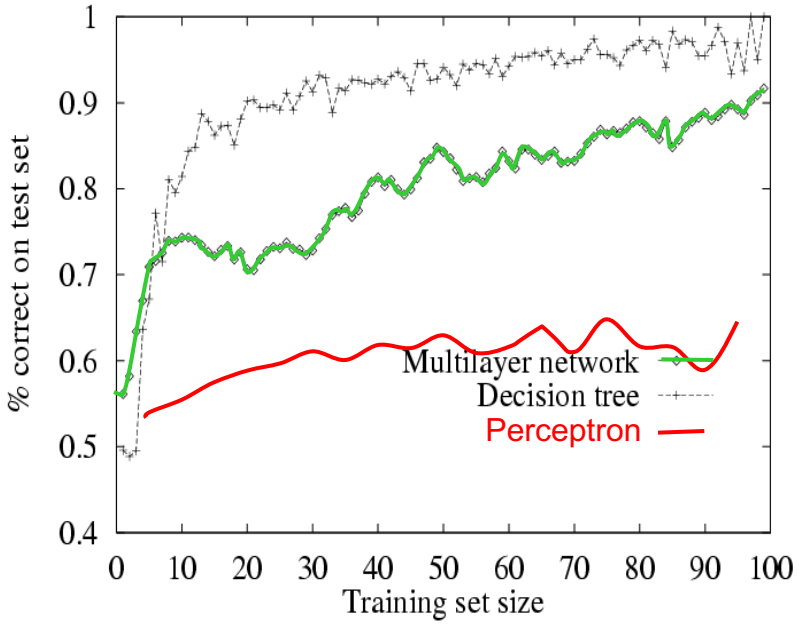
Back-Propagation is Gradient Descent



- Error surface for gradient descent in the weight space
- Back-prop provides a way of dividing the calculation of the gradient among the units, so the change in each weight can be calculated by the unit to which the weight is attached using only local information

Performance of Back-Propagation





- Performance on the WillWait Restaurant problem
- At left, a training curve showing the error over the number of epochs (iterations of back-prop)
- At right, the performance relative to decision trees
- How does this compare to the perceptron?

Example Applets

- https://playground.tensorflow.org/

Noise and Overfitting

- In the presence of noise, some feature vectors will have multiple examples with conflicting results
- If there are many possible hypotheses, you must be careful to avoid finding meaningless "regularity" in the data (overfitting)
  - Every time I roll the dice with my left hand it comes up heads
  - I always encounter less traffic on Mondays (but I'm always late on Mondays)
- There are techniques for dealing with overfitting, but they rely on domain information

Handling Noise

- PAC learning : probably approximately correct learning
- For a given hypothesis *h* of some function *f*
- Approximately correct if

  error(h) ≤ ε

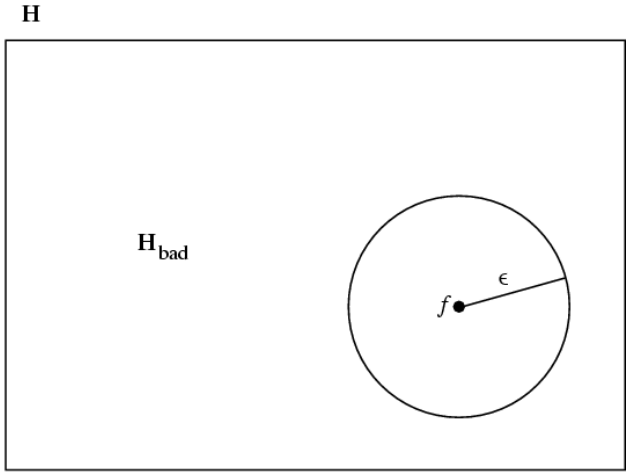
  Where

  error(h) = P(h(x) ≠ f(x) | example x)

  (that is, the hypothesis is within some epsilon-ball of f)
- Probably Approximately correct if

  P(error(h) ≤ ε) ≥ ε_2

  (that is, the hypothesis is probably within some epsilon-ball of f)



Administrivia

- PS #5 out now, due next Monday