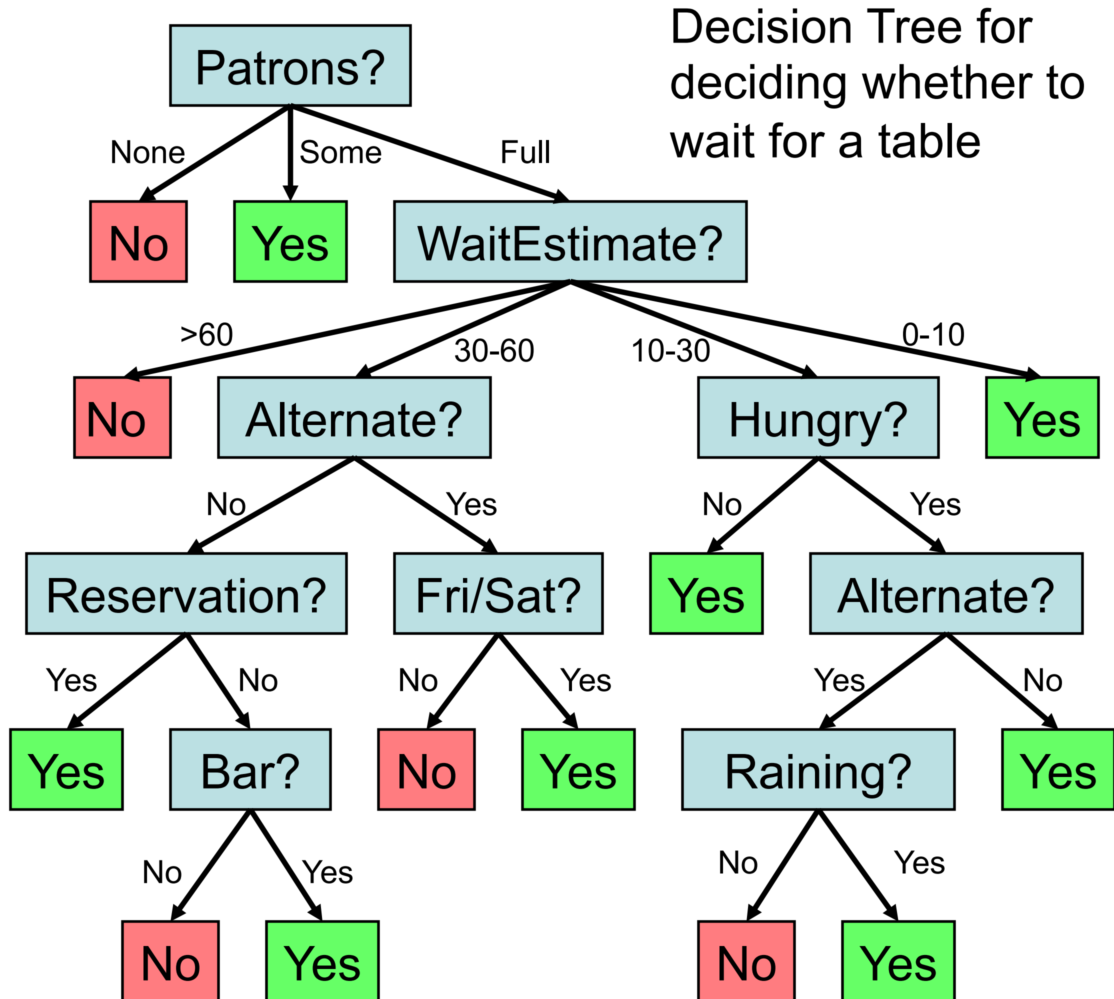
# Supervised Learning

CPSC 470 – Artificial Intelligence

Brian Scassellati

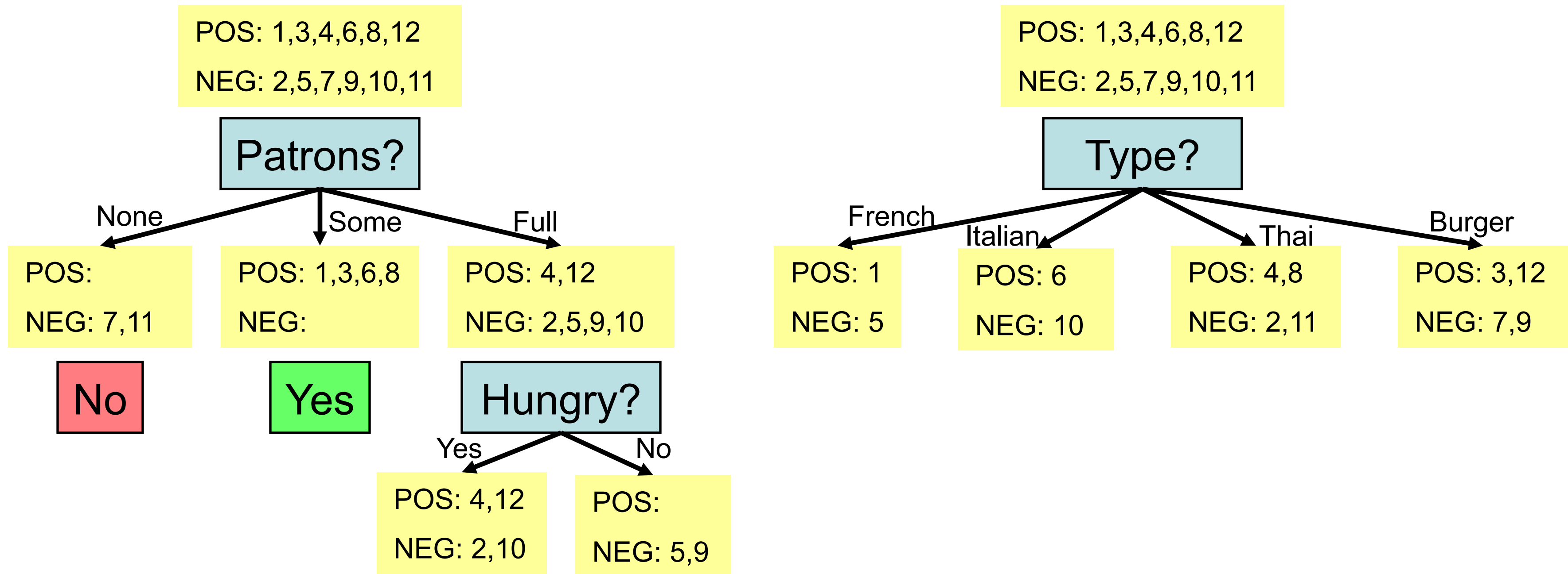
# Decision Trees




Decision Tree for deciding whether to wait for a table

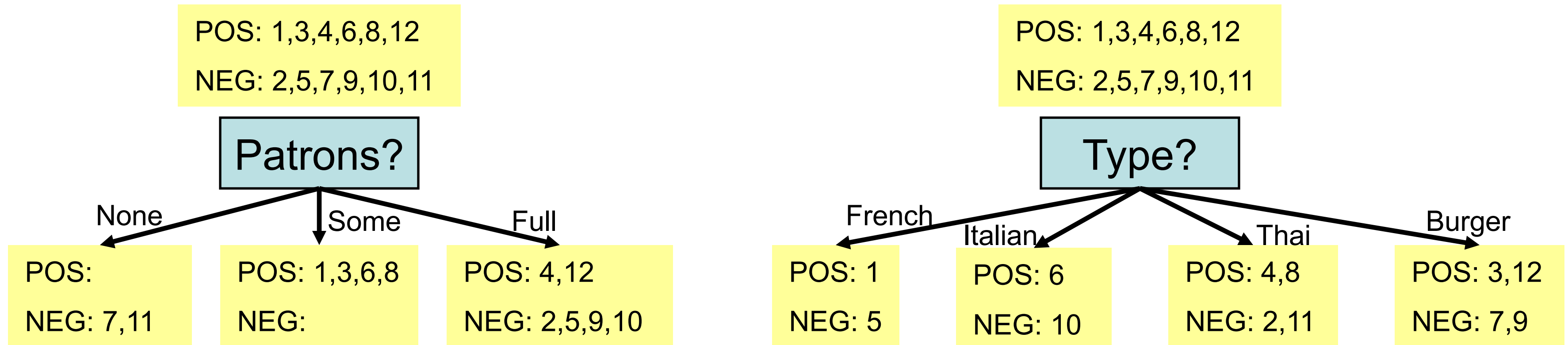
- Represents a Boolean function (the goal predicate WillWaitForTable())
- Leaves are Boolean values
- Internal nodes are tests of a feature/property
- Represent a propositional logic statement
  - Each path could be a line in a truth table

# Learning a (Reasonable) Decision Tree




- Test the most important feature first
- If you have only one type of example, return a leaf
- Else, choose the next most important feature
- If you run out of examples, return a default value (no data)
- If you run out of features, you are in trouble (two examples have same description: noisy data)

# Information Content




$$Remainder(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$

$$Remainder(Patrons) = \tfrac{2}{12} I(0,1) + \tfrac{4}{12} I(1,0) + \tfrac{6}{12} I(\tfrac{2}{6}, \tfrac{4}{6})$$

$$Remainder(Patrons) \approx 0 + 0 + \tfrac{6}{12}(-\tfrac{2}{6}\log\tfrac{2}{6} - \tfrac{4}{6}\log\tfrac{4}{6})$$

$$Remainder(Patrons) \approx 0.459 \text{ bits}$$

Needs less info to make a perfect choice

$$Remainder(Type) = \tfrac{2}{12} I(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{2}{12} I(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{4}{12} I(\tfrac{2}{4}, \tfrac{2}{4}) + \tfrac{4}{12} I(\tfrac{2}{4}, \tfrac{2}{4})$$

$$Remainder(Type) = \tfrac{2}{12}(1) + \tfrac{2}{12}(1) + \tfrac{4}{12}(1) + \tfrac{4}{12}(1)$$

$$Remainder(Type) = 1$$

# Outline

- Decision Trees learn Boolean functions (basically, a predicate in propositional logic)
- Today, we look at two additional learning techniques
  - Version spaces allow us to learn arbitrary propositional logical statements
  - Neural Networks allow us to learn arbitrary functions mapping input features to output features

# Version Spaces

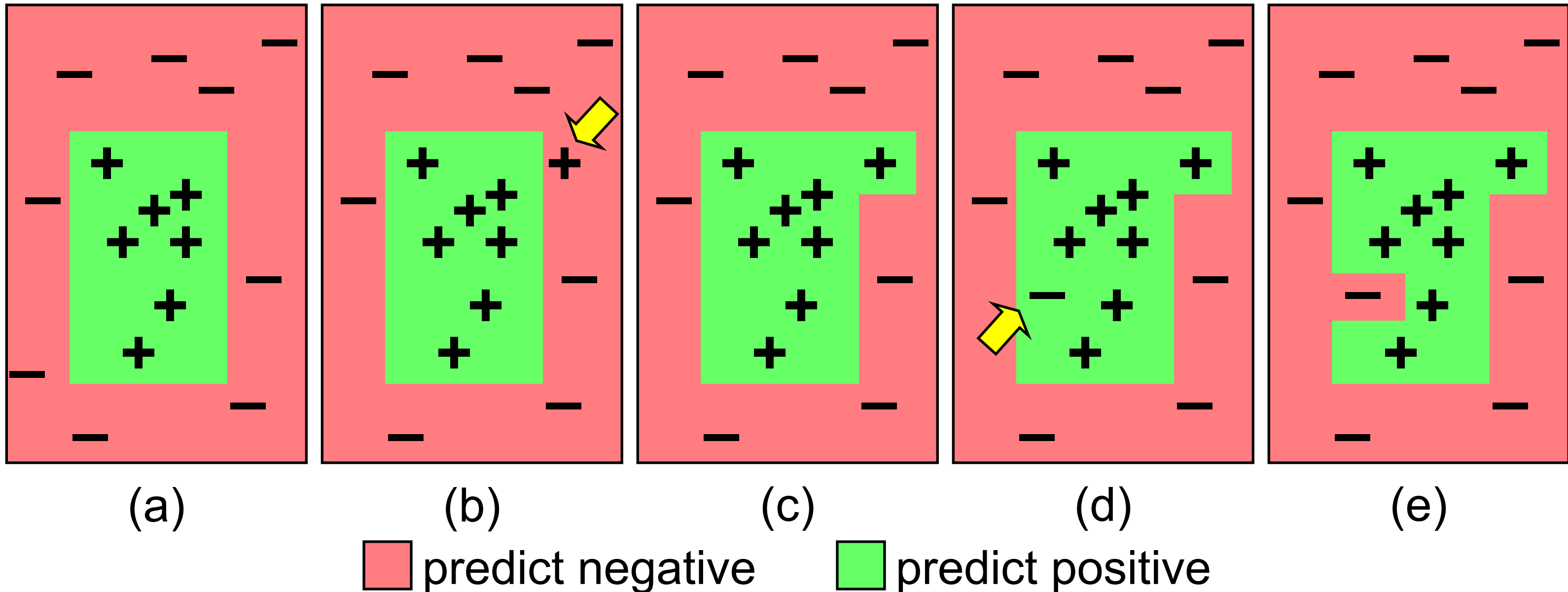
(Learning arbitrary
propositional logical statements)

# First, a Few Definitions

- In Decision Trees, hypothesis is a predicate
  - Such as WillWait for the restaurant example
- Example is a situation in which the predicate may or may not be true
  - Express an example as an arbitrary sentence

    $Alternate(X_1) \wedge Hungry(X_1) \wedge \neg Fri/Sat(X_1) \wedge \ldots$
- A hypothesis agrees with all the examples if and only if it is logically consistent with each example

# Consistency of a Hypothesis




- (a) consistent
- (b) false negative
- (c) generalization includes the false negative example
- (d) false positive
- (e) specialization removes the false positive example

# Finding Consistent Hypotheses: Current-Best-Hypothesis Search

```
function CURRENT-BEST-LEARNING(examples) returns a hypothesis

  H ← any hypothesis consistent with the first example in examples
  for each remaining example in examples do
      if e is false positive for H then
          H ← choose a specialization of H consistent with examples
      else if e is false negative for H then
          H ← choose a generalization of H consistent with examples
      if no consistent specialization/generalization can be found then fail
  end
  return H
```

Example truth table

| Hot | Windy | Arthritis | Rainy | Prediction |
|---|---|---|---|---|
| True | True | True | True | |
| True | False | False | False | H1=True |
| False | True | True | True | H2=False |

- Start with a true example

  H1 : $\forall$d Rainy(d) $\Leftrightarrow$ Hot(d)

- False Positives (negative examples) trigger specialization and add a clause

  H2 : $\forall$d Rainy(d) $\Leftrightarrow$ Hot(d) $\wedge$ Windy(d)

- False Negatives (positive examples) trigger generalization and remove a clause

  H3 : $\forall$d Rainy(d) $\Leftrightarrow$ Windy(d)

# Problems with Current-Best-Hypothesis Search

- Need to check all previous examples against any modification
  - Can be very expensive
  - Must maintain all examples in memory
- Hard to find good heuristics of what to add or remove
  - May need to backtrack if you make the wrong pick
  - Unless you keep track of the various modifications that you have tried, your function may not terminate

# Finding Consistent Hypotheses: Least-Commitment Search

```
function VERSION-SPACE-LEARNING(examples) returns a version space
   local variables: V, the version space: the set of all hypotheses

   V ← the set of all hypotheses
   for each example e in examples do
      if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
   end
   return V
```

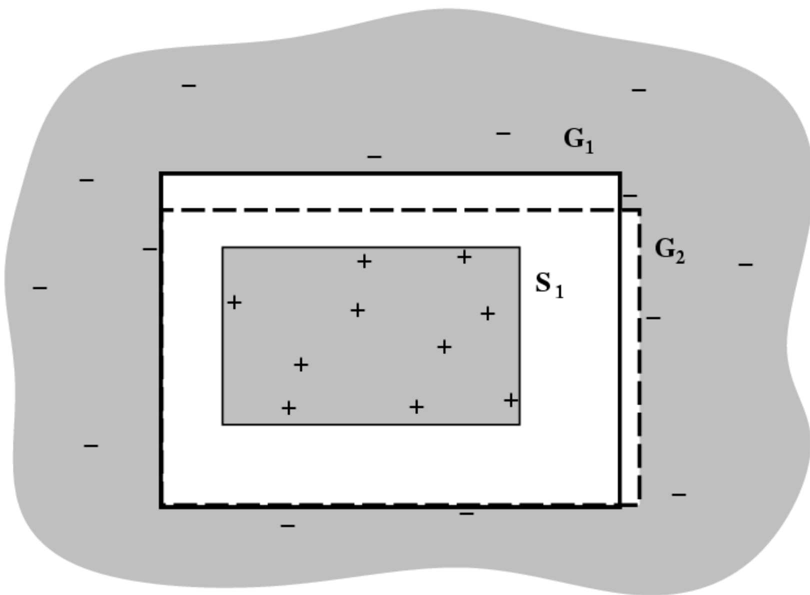
---

```
function VERSION-SPACE-UPDATE(V, e) returns an updated version space

   V ← {h ∈ V : h is consistent with e}
```

- Rather than pick a solution, consider the entire space of hypotheses H1 ∨ H2 ∨ H3 ∨ H4 ∨ H5 ∨ …
- The right answer will be in there (somewhere)
- Remove hypotheses that are inconsistent with particular examples
- Makes no arbitrary choices… least commitment principle
- But how do we represent the space of all possible hypotheses?

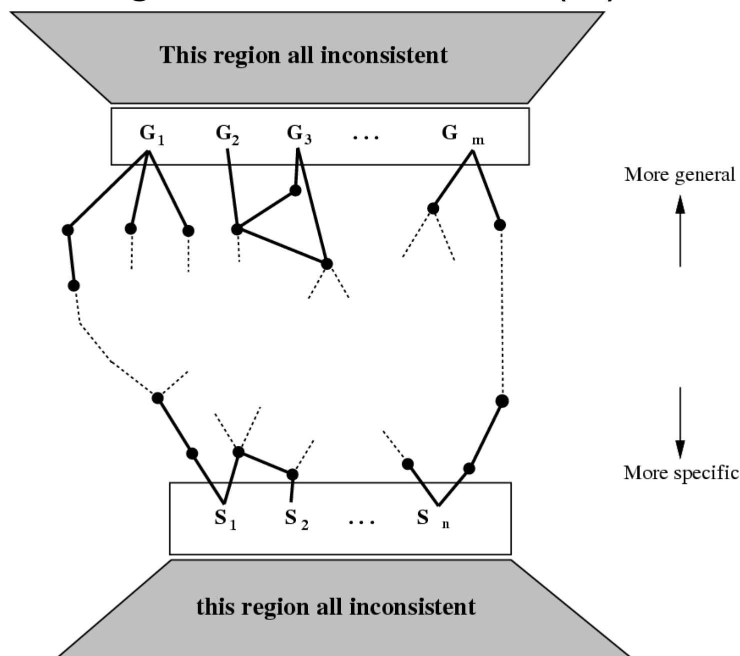
# Version Space Learning




- How does this work?
  - Maintain boundaries on the limits of the valid positives (S set) and the valid negatives (G set)
  - Adjust boundaries as new examples are generated

# Version Space Learning

Most general boundaries (G)




Most specific boundaries (S)

- Initialize G=True and S=False
- For each positive instance i+
  - *Remove members of G that don't match i+*
  - *Generalize members of S until they match i+, retaining only most general members that remain more specific than G.*
- For each negative instance i-
  - *Remove members of S that match i-*
  - *Specialize members of G until they don't match i-, retaining only most specific members that remain more general than S.*
- Continue until
  - Obtain one concept
  - Space collapses (S or G becomes empty) … failure
  - Run out of examples

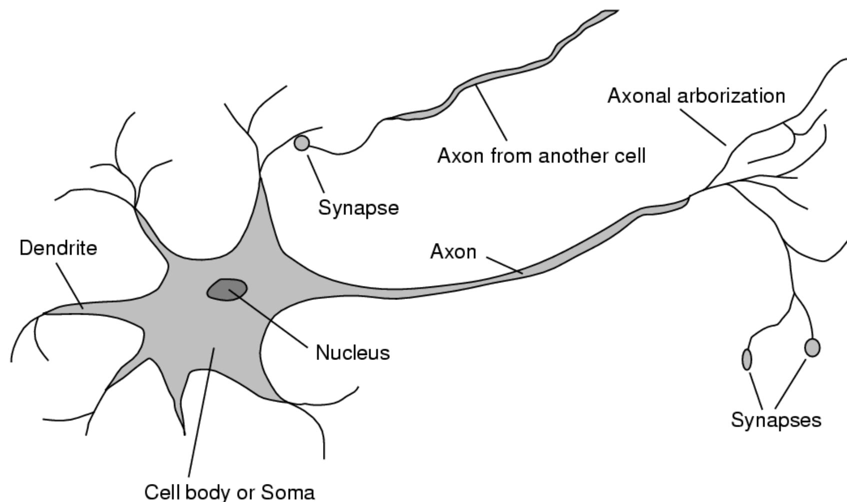
# Problems with Version Spaces

- Noise
  - If the domain contains noise or insufficient attributes for exact classification, the space will collapse
- May not tell us anything practical
  - If we allow unlimited disjunctions in the hypothesis space
  - Then the S-set will always contain the most-specific hypothesis (the disjunction of all the positive examples)
  - And the G-set will contain the negation of the disjunction of the descriptions of the negative examples…
  - We have a rule that is just the listing of examples

# Neural Nets

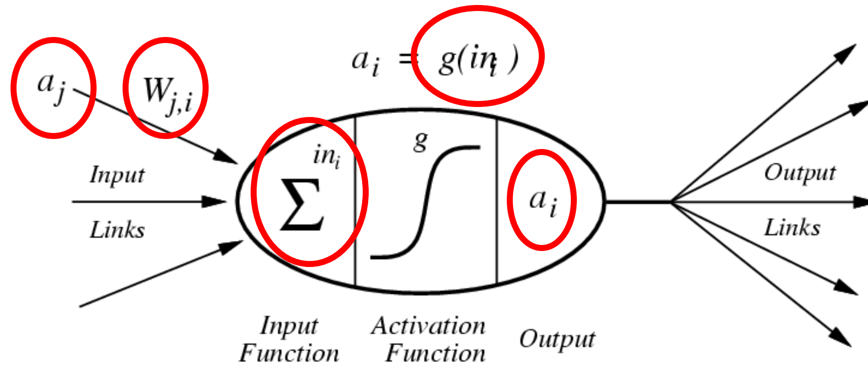
(Learning arbitrary functions mapping input features to output features)

# 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



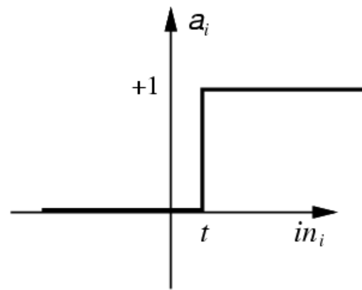

For neuron i :

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

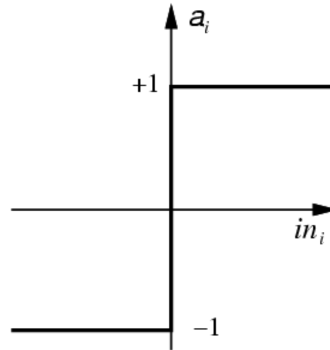
$$a_i = g(in_i)$$

- 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

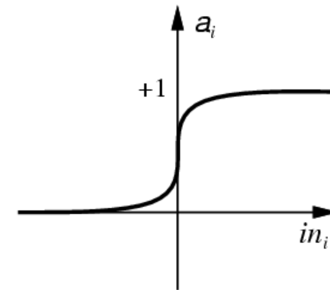
# Activation Functions




(a) Step function



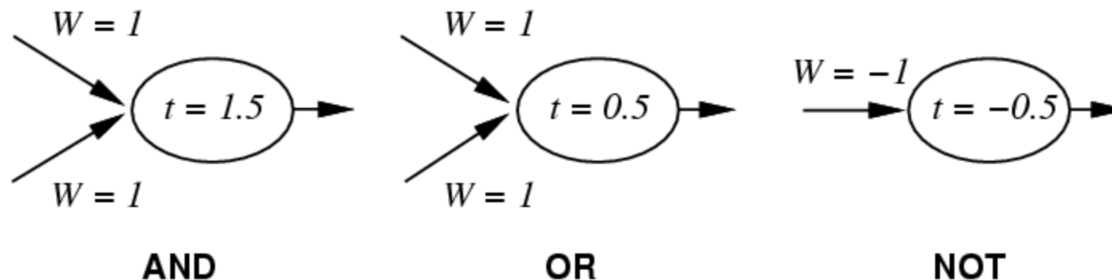

(b) Sign function




(c) Sigmoid function

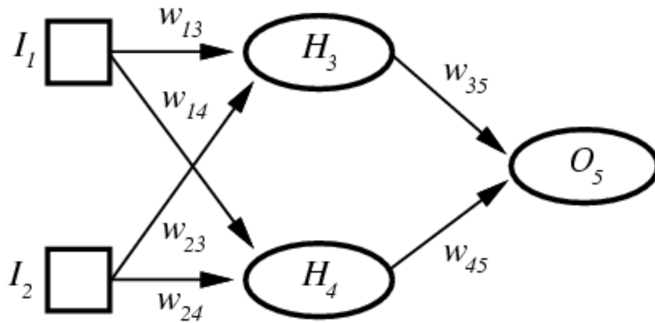
- Thresholds can be added by using a constant input (positive or negative)
  - Thus converting any step function to a sign function
- Sigmoid is most common activation function, as it avoids discontinuities (and has a useful derivative for gradient descent)

# 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

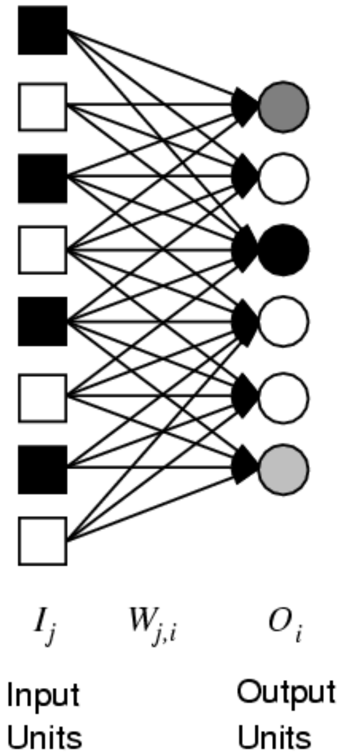


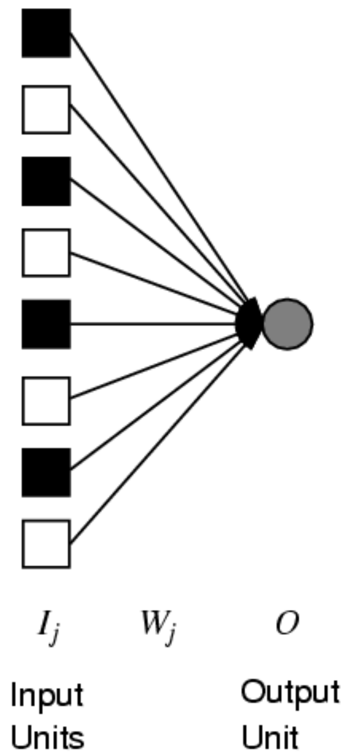

$$a_5 = g(W_{3,5}a_3 + W_{4,5}a_4)$$

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

- 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

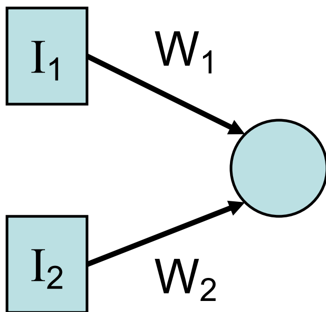
# Perceptrons



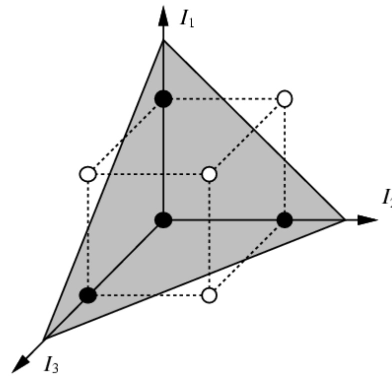

- Layered feed-forward networks
  - Output=Step(W*I)
  - Assume threshold=0 without loss of generality
- Majority function (output 1 if at least half the inputs are 1)
  - All weights are 1
  - Threshold is –n/2 for n inputs
  - (would have required a decision tree with $2^n$ nodes)
- Can we represent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

# Linear Separability of Perceptrons

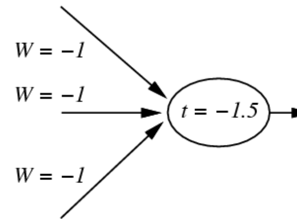



- Output is 1 if and only if
  $(W_1 I_1 + W_2 I_2) > 0$
  $I_1 = -(W_2 / W_1)\, I_2$
- Separation between output states and input states is a line (it is linearly separable)
  - Adding a threshold will only allow for an offset of the line
- Some functions can be computed in this way (AND, OR)
- Some 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

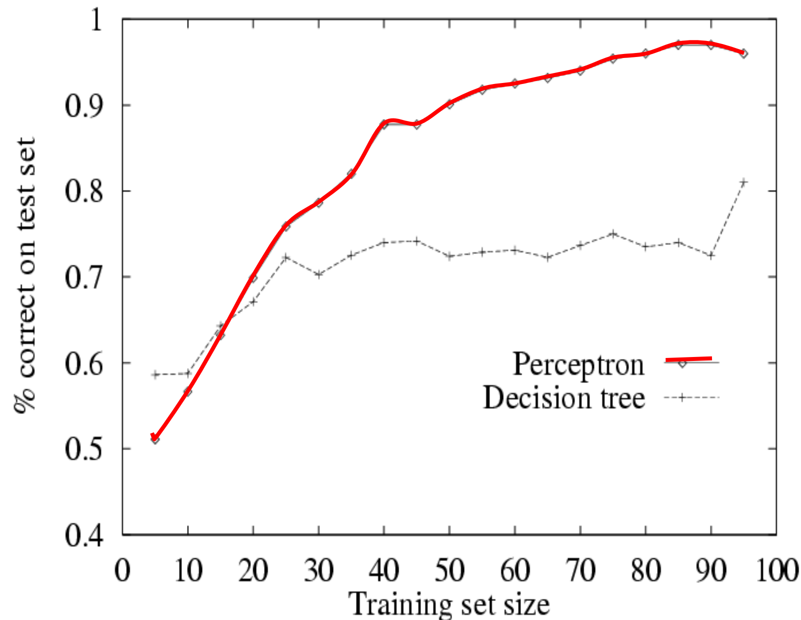
- At each time step, compute the error

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

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

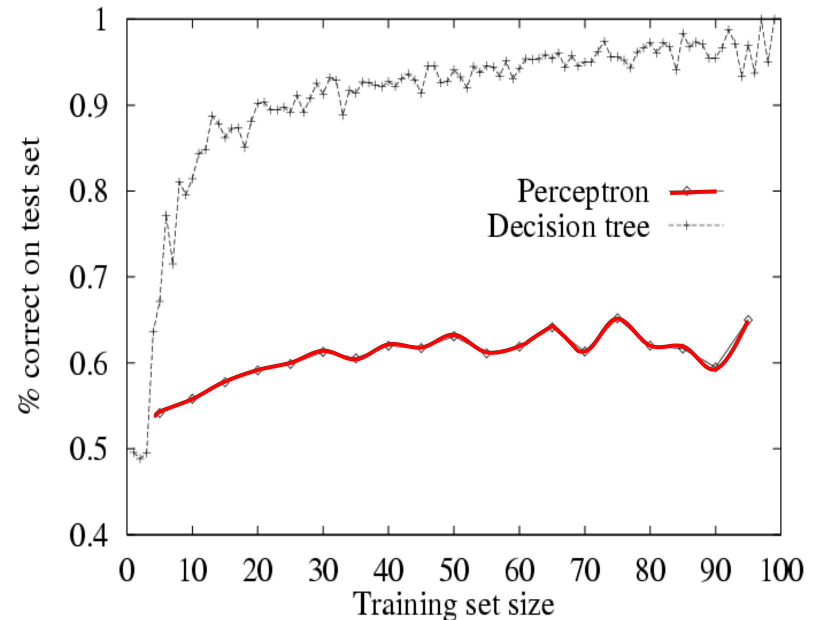
  where $\alpha$ is the learning rate
- Guaranteed to learn any linearly separable function given sufficient training examples

# Which are better, Perceptrons or Decision Trees?




11-input majority function

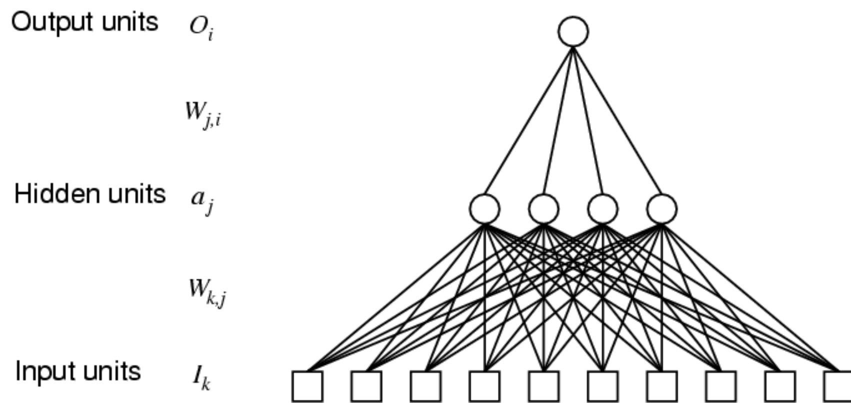
Perceptrons are better




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

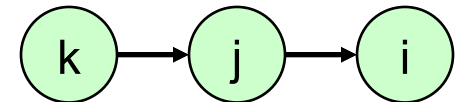
**function** BACK-PROP-UPDATE(*network*, *examples*, $\alpha$) **returns** a network with modified weights
  **inputs**: *network*, a multilayer network
    *examples*, a set of input/output pairs
    $\alpha$, the learning rate

  **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 $\Delta$ 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

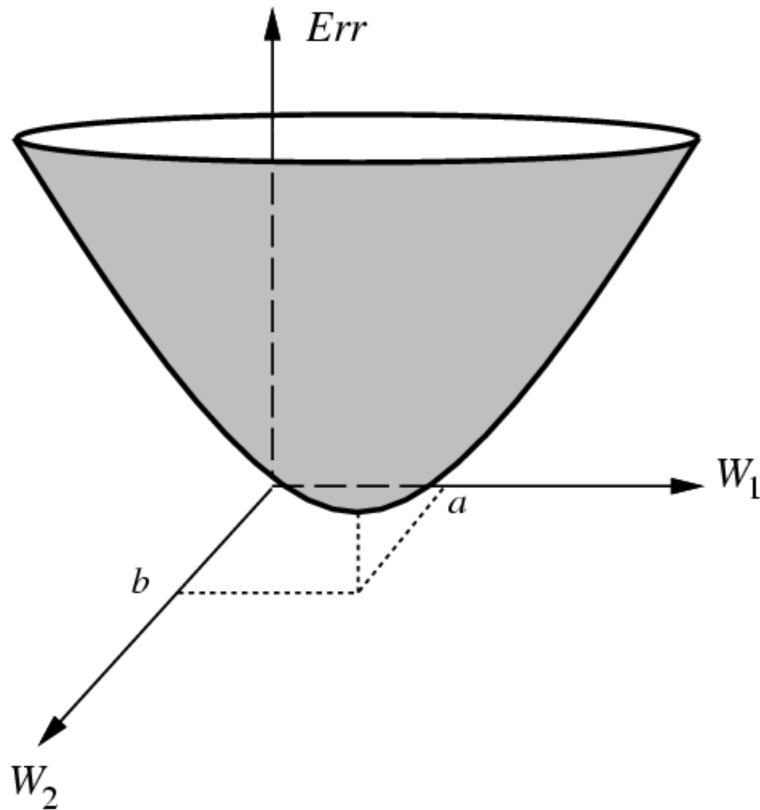



For internal node j, it is responsible for some fraction of the error at node i

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

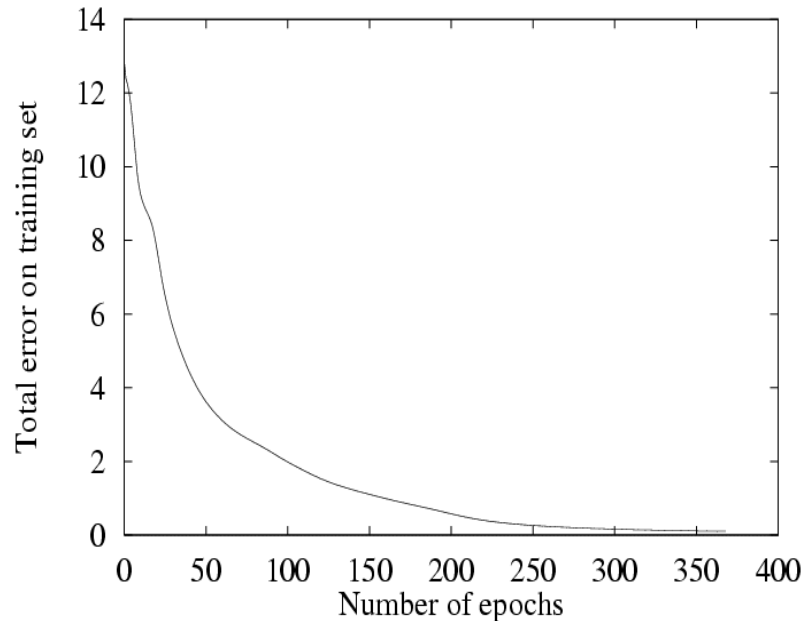
Errors propagate back from the output

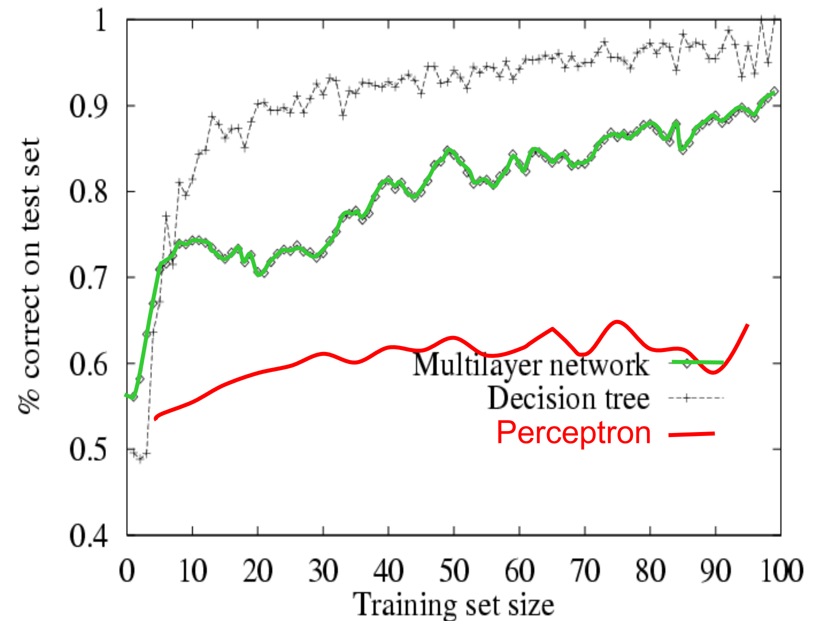
# Back-Propagation is Gradient Descent




- 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

- The examples shown in class are available from the Computational Intelligence group at the University of British Columbia
  - http://www.aispace.org/

# Other Network Definitions

- Hopfield networks
  - All nodes are input and output
    - Symmetric bidirectional connections ($W_{ij}=W_{ji}$)
    - Activation levels are +1 or -1
    - Function is the sign function
  - Associative memory
    - Once trained on a set of inputs, a new presentation will settle to the trained input that most closely resembles the novel input
- Boltzmann machines
    - Symmetric weights
    - Some hidden nodes (neither input nor output)
    - Stochastic activation function

# Administrivia

- Midterm on Monday!
- Q&A session today, 3:30-4:30 here