(Concluding) Decision Trees
and (Starting) Perceptrons

CSCI 3202, Fall 2010
Assignments

• Problem Set 3 is due Tuesday 11/2
• Read Section 18.7 in the textbook
Splitting Examples into “Training” and “Test” Sets

• Given our initial set of examples, split it into a (randomly-chosen) training set and a test set.

• Once the algorithm has generated a tree for the training set, use the test set to gauge the accuracy of the tree (measure the percent of the test set that is correctly classified).

• For “sufficiently large” and “sufficiently representative” training sets we converge on a high accuracy for the test set.
Things We’ve Swept Under the Rug

• Doesn’t this strategy lead to attributes with many possibilities? (E.g., “day of the year”?)
• Are we sure that all new examples will be completely classified?
• Aren’t there some functions that are hard to express using decision trees?
A “Bad” Concept for Decision Trees to Learn

• Majority function (classified as true whenever the majority of the attributes are positive, false otherwise)

• Each attribute is equally important, and none are very effective at dividing the set
Neural Networks: Some First Concepts

• Each neural element is loosely based on the structure of neurons
• A neural net is a collection of neural elements connected by weighted links
• We think of some set of neurons as “input elements”; these are linked to “output elements” which can be interpreted as a classification of the input pattern
• Standard formats: perceptrons and multilayer feedforward networks
Standard diagrams of feedforward networks
Structure of an (artificial) neuron

- Think of each neuron as a very simple computational element: it receives numeric input values, sums those values, and compares them to a threshold.
- If the sum of the inputs is greater than the threshold, the neuron outputs a 1; otherwise it outputs a zero.
- The output of this neuron is connected (as usual, via weighted links) to subsequent neurons in the net.
Perceptrons: the Simplest Neural Network

- Perceptrons are two-layer networks: one layer of inputs directly connected to a layer of outputs.
- For simplicity, we can look at a perceptron with a single output node.
Input 1

Weight \( W_1 \)

Input 2

Weight \( W_2 \)

\[ \cdots \]

Weight \( W_N \)

Input \( N \)

\[ \sum \]

Threshold \( \theta \)

\[ \text{Sigmoid} \]

Output
An AND perceptron

An OR perceptron
Training a Perceptron by Adjusting its Weights

- Overall error is the value of the difference between what we wanted and what we got from our perceptron. (In some situations we use the “squared” error to avoid issues of sign.)

- Once we see that our perceptron is in error, we can adjust each of the weights leading to the output node. We’ll adjust each weight in such a way as to make the error value smaller.
The Update Rule for a Weighted Edge of a Perceptron

To update the weight from a node j leading to an output node, we adjust the weight according to the following formula:

\[ W_j \leftarrow W_j + (\alpha \cdot \text{Err} \cdot g'(\text{in}) \cdot x_j) \]

Here,

- \text{Err} is the difference between what we wanted and what we got from the output. (i.e., the “un-squared” error)
- \( g'(\text{in}) \) is the derivative of our output function
- \( x_j \) is the output from node j leading to us
- \( \alpha \) is a “rate parameter”
Side Issue 1: The Derivative of the Output Function of a Neuron

• A handy choice for output function of a neuron is a sigmoidal function. Let’s call “x” in this case the sum of all the inputs. Then for a sigmoidal function:
  \[ \text{out}(x) = \frac{1}{1 + e^{-x}} \]

• If we use the sigmoidal function \( \text{out}(x) \), then
  \[ \text{out}'(x) = \text{out}(x)(1 - \text{out}(x)) \]
The Sigmoidal Output Function

\[
g(in) = \frac{1}{1+e^{-in}}
\]

\[
g'(in) = g(1 - g)
\]
Side Issue 2: What Can’t a Perceptron Learn?

• Consider the XOR function. Suppose we want a two-input perceptron, with inputs A and B, that can output 1 when A xor B, and outputs 0 otherwise.

• Since our perceptron outputs a 0 when both A and B are 0, and since increasing A alone changes the output to 1, the edge weight on A must be positive…
Perceptrons Can’t Learn XOR

- By similar considerations, the edge weight on B must also be positive…
- But then suppose we increase A to get the perceptron to output 1, and then increase B. The total input to the perceptron’s output neuron must be increasing, but it has to change from 1 to 0!
Multi-Layer Neural Networks

Now, we expand our network architecture so that it has several (or more) layers of neural elements. We think of the layers as numbered in columns left to right, with the leftmost layer as “layer 1” (input) and the rightmost layer as “layer n” (output).
Adjusting Weights for Multilayer Networks

Here’s the perceptron rule, again:

\[ W_j \leftarrow W_j + (\alpha \cdot \text{Err} \cdot g'(\text{in}) \cdot x_j) \]

Note that this is the update rule to adjust a weight from neuron j to an output neuron. Let’s rewrite this for our new multilayer network as:

\[ W_{j\rightarrow k} \leftarrow W_{j\rightarrow k} + (\alpha \cdot \Delta_k \cdot a_j) \]

That is, we call the output neuron k and we combine the error term and the derivative-at-k term into one symbol, \( \Delta \). We also relabel the output coming from neuron j as \( a_j \). Think of the \( \Delta \) term as “how much node k wants its output to change”. For an output node, this is just the product of the error at k and the rate at which error changes for a small change in input at k.
How Much Does a Hidden Node Want to Change Its Output?

Okay, what about a hidden-layer node: what should its value of $\Delta$ be?

It is the product of its own derivative and the weighted sum of how much its own targets “want” to change:

$$\Delta_j = g'(\text{in}_j) \sum_m W_{j\rightarrow m} \Delta_m = a_j(1 - a_j) \sum_m W_{j\rightarrow m} \Delta_m$$

Again, think of these terms as “how fast I will change my output for a little change in input” and “the sum over all my targets of how much they want to change, weighted by my influence upon them”
Backpropagation: Step 1

Start at the output layer. For each neuron in that layer, we compute:

$$\Delta_{\text{output-}i} = g'(\text{in}_{\text{output-}i}) \text{Error}_{\text{output-}i}$$
$$= a_{\text{output-}i}(1 - a_{\text{output-}i}) \text{Error}_{\text{output-}i}$$
Backpropagation: Step 2

Now, for each neuron $j$ in the previous layer, compute $\Delta j$ as follows:

$$\Delta_j = a_j (1 - a_j) \sum_i W_{j\rightarrow i} \Delta_i$$

Now, use each of these $\Delta$ values to compute the $\Delta$ values for the layer before that, and so on until you reach the input layer.
Backpropagation: Final Step

We now have, for each node, a measure of how much that node wishes to change its own output value. We now adjust the weights on each edge as follows:

$$W_{j\rightarrow i} \leftarrow W_{j\rightarrow i} + (\alpha a_j \Delta_i)$$
A Sampler of Additional Issues

• How do we choose an appropriate network size to train? Too small a network, and we may not be able to represent our concept; too big, and we run the risk of *overfitting*.

• How do we choose a training rate parameter? (The notion of “simulated annealing”).

• Training partial nets (e.g., one output at a time).
Neural Nets: What Do People Like?

*The pros:*

Utility for a variety of pattern recognition tasks (handwriting, spoken word, face, etc.)

Simplicity of programming (reinforcement learning)

An elegant model of emergence in programming: the idea that large numbers of simple programming elements are collectively capable of interesting, higher-level behaviors.
Wait! Wait! ... Cancel that, I guess it says ‘helf.’"
What Do People Dislike?

The cons:

A vague feeling that we haven’t really learned anything about these pattern recognition skills

Neural nets seem to be a far less natural means for demonstrating “explicit rule learning”
What Is It Like to Be a Bat?
The Problems of Bathood

• How do you avoid hurting your own ears?
• How can you estimate which objects are closer to you and which are farther away?
• How can you tell if an object is moving toward or away from you?
David Marr’s Levels of Abstraction

• Computational Level: *What is the goal of the computation, why is it appropriate, and what is the logic of the strategy by which it can be carried out?*

• Algorithm Level: *How can this computational theory be implemented? In particular, what is the representation for the input and output, and what is the algorithm for the transformation?*

• Mechanism Level: *How can the representation and algorithm be realized physically?*
The Computational Approach to Vision: A Sampler of Issues

• Low-level vision:
  Edge segmentation: which areas of the scene projected onto the retina correspond to the potentially interesting objects?
  Depth perception: how far away are the objects?

• Interpreting Edges as “Line Drawings”

• Interpreting Shapes as (possibly familiar) objects
What Do We Mean by “Low-Level” Vision, Anyway?

- Based on a (not entirely accurate) portrait of vision as proceeding “bottom-up” from raw (pixel-like) data to object recognition
- Tasks like: object boundary detection, surface orientation, depth perception, motion detection
- Fast, “rough”, domain-independent processing
- Parallel processing
- Unconscious levels of (human) vision processing