Decision Tree Classification Example

- Consider a problem where instances have the following attributes: fever, cough and pain location. A decision tree example:

- To classify an object, you start at the root node and traverse the branches
- Each node specifies an attribute that an object must be tested on
- Each leaf node represents a classification
Which attribute is best?

For now, let's consider a problem with two classes: positive and negative (like in concept learning). And two attributes A1 and A2, each can take on values true or false.

- Want a test that creates subsets which are relatively "pure" in one class so they are closer to being leaf nodes.
- Information gain is the most popular of many such possible methods.
Entropy

- $D$ is a sample of training examples
- $p_\oplus$ is the proportion of positive examples in $D$
- $p_\ominus$ is the proportion of negative examples in $D$
- By definition: $0 \leq p_\oplus \leq 1$, $0 \leq p_\ominus \leq 1$ and $p_\oplus + p_\ominus = 1$.
- Entropy measures the impurity, or disorder, of $D$

\[
Entropy(D) \equiv -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus
\]
Evaluating Attributes with Information Gain

Information gain is the expected reduction in entropy due to distributing instances in a sample with respect to values of an attribute.

\[
Gain(D, A) \equiv Entropy(D) - \sum_{v \in Values(A)} \frac{|D_v|}{|D|} Entropy(D_v)
\]

\[
Entropy(D) = -\frac{29}{64}\log_2\left(\frac{29}{64}\right) - \frac{35}{64}\log_2\left(\frac{35}{64}\right) = 0.994
\]

\[
Entropy(A1_t) = -\frac{21}{26}\log_2\left(\frac{21}{26}\right) - \frac{5}{26}\log_2\left(\frac{5}{26}\right) = 0.706
\]

\[
Entropy(A1_f) = -\frac{8}{38}\log_2\left(\frac{8}{38}\right) - \frac{30}{38}\log_2\left(\frac{30}{38}\right) = 0.742
\]

\[
Gain(D, A1) = 0.994 - \left(\frac{26}{64}\right)0.706 - \left(\frac{38}{64}\right)0.742 = 0.2666
\]
Evaluating Attributes with Information Gain

\[ \text{Gain}(D, A) \equiv \text{Entropy}(D) - \sum_{v \in \text{Values}(A)} \frac{|D_v|}{|D|} \text{Entropy}(D_v) \]

Recall that \( \text{Gain}(D, A_1) = 0.2666 \). So we should choose \( A_1 \).
Basic algorithm for learning decision trees

• Top-down divisive algorithm. Starts from root node and all instances.
  – Asks: “Which attribute should be tested at this node?”
  – Funnels instances down the tree according to their values for that attribute.
  – Picks most common class if no attributes left to test. Otherwise, repeats the above two steps.

• Greedy (never backtracking) search for the “best” tree that fits the training examples.
Continuous-valued Attributes

• How can continuous-valued attributes be incorporated into decision trees?
  More precisely: What question should a node using continuous-valued attribute $A$ ask?

• We create a new boolean attribute $A_c$ that is true if $A < c$, false otherwise.
  – This procedure is called *thresholding*.
  – $c$ is called a *threshold*.

• While learning the tree:
  – How do we know if it's best to split based on $A$ or some other attribute?
  – How do we choose $c$?
How to choose a threshold?

- How do we choose the best value for $c$? There are infinitely many candidates.

Sort instances in $D$ according to their $A$ values.

Let's denote the sorted attributes $A_1, A_2, ..., A_N$, and the corresponding target classifications $f_1, f_2, ..., f_N$.

For all $i$ such that $f_i \neq f_{i+1}$

Let $c = (A_i + A_{i+1})/2$

Compute the information gain of $A < c$

Pick the $c$ that resulted in the highest information gain.

- Now this boolean attribute $A_c$ can be compared against other attributes using information gain as before.
Example

- Lets represent *temperature* and *new snow* with continuous values. We’ll call *temperature* $t$ and *new snow* $s$ for brevity.

- Training instances (t: degrees Fahrenheit, s: inches)

<table>
<thead>
<tr>
<th>$t$</th>
<th>$s$</th>
<th>$\text{ski?}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-18</td>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>-20</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>+</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$t$</th>
<th>$s$</th>
<th>$\text{ski?}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>5</td>
<td>+</td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>+</td>
</tr>
<tr>
<td>42</td>
<td>18</td>
<td>+</td>
</tr>
<tr>
<td>55</td>
<td>4</td>
<td>+</td>
</tr>
<tr>
<td>48</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>-25</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>
• Sort with respect to temperature:

<table>
<thead>
<tr>
<th>t</th>
<th>-25</th>
<th>-20</th>
<th>-18</th>
<th>3</th>
<th>7</th>
<th>15</th>
<th>22</th>
<th>33</th>
<th>36</th>
<th>42</th>
<th>48</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td>ski?</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

– Candidate thresholds: 5, 18.5, 27.5, 45 and 51.5
– Best information gain for $t$ obtained by $t < 5$ ($gain = 0.459$)

• Sort with respect to new snow:

<table>
<thead>
<tr>
<th>s</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>ski?</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Best information gain for $s$ also happens to be 0.459 (a rare coincidence). Lets just use $t$ at the root node of the tree.
What should be the next node of the decision tree?
Temperature (F) vs. New snow (inches) for skiing day classification.

- Good ski day: Temperatures above 0°F with snowfall above 5 inches.
- Not a good ski day: Temperatures below 0°F or snowfall below 5 inches.

Decision tree:
- Check if temperature is above 0°F.
  - Yes: Check if snowfall is above 5 inches.
    - Yes: Good ski day.
    - No: Not a good ski day.
  - No: Not a good ski day.
The case for linear discriminant functions

- Remember the decision tree for our example

- Are decision trees the best way to classify this type of data?
• Is this complex tree justified?

• No. The decision tree building block for real valued attributes \((\text{attribute} < \text{constant})\) is not the best fit for this problem.
• A better (much simpler) decision boundary:
  – Instances above the line are classified as positive.
  – Instances below the line are classified as negative.

• Training sets that can be separated without error by a linear function (line, plane, hyperplane, etc.) are called *linearly separable*. 
Linear discriminants

- Real valued attributes $x_1$ through $x_n$.
- Real valued weights $w_0$ through $w_n$.
- Linear discriminant function:

$$g(x_1, \ldots, x_n) = \left( w_0 + \sum_{i=1}^{N} w_i x_i \right)$$

- Vector notation:
  - Vector of augmented attributes $\vec{x} = (1 \ x_1 \ x_2 \ \ldots \ x_n)$
  - Vector of weights $\vec{w} = (w_0 \ w_1 \ w_2 \ \ldots \ w_n)$
  - Linear discriminant is the dot product $g(\vec{x}) = \vec{w} \cdot \vec{x}$
Perceptron

- How to use linear discriminants for classification

\[ \text{Perceptron: } o(\vec{x}) = \text{sgn}(\vec{w} \cdot \vec{x}) = \begin{cases} +1 & \text{if } \vec{w} \cdot \vec{x} \geq 0 \\ -1 & \text{otherwise} \end{cases} \]

\[ \vec{w} \cdot \vec{x} = 0 \] is called the decision surface. The decision surface for perceptrons is a hyperplane.
Logical operator perceptrons

- Perceptrons can represent all primitive logical operators. Examples:
  - $x_0 \land x_1$ can be represented with $w_0 = -0.6, w_1 = w_2 = 0.5$.
  - $x_0 \lor x_1$ can be represented with $w_0 = -0.4, w_1 = w_2 = 0.5$.
  - $\neg x_0$ can be represented with $w_0 = 0.5, w_1 = -1$
  - The choices for the weights are not unique.

- Not all logical expressions can be represented by perceptrons.
  - The exclusive or logic function
    \[
    XOR(x_1, x_2) = (x_1 \land \neg x_2) \lor (\neg x_1 \land x_2)
    \]
    can not be represented by a perceptron.
Multi-class classification

• Example decision boundaries for multi-class classification generated by linear machines:

• Linear machines can only generate convex and singly-connected classification regions.
Multi-class classification

- $c$ output classes, one discriminant function $g_j$ for each class

$$g_j(\vec{x}) = \vec{w}_j \cdot \vec{x}$$

Notice that each class has its own weight vector $\vec{w}_j$.

- Perceptrons are not adequate here. Why not?

- Assign $x$ to class $j$ if $g_j(\vec{x}) \geq g_k(\vec{x})$ for all $k \neq j$. This is sometimes called a linear machine.
Delta rule and gradient descent

- The *delta rule* can find a best-fit approximation to examples that are not linearly separable.

- *Gradient descent* is used to search the hypothesis space.

- *Gradient descent* works only on differentiable functions. Can’t use the perceptron, it is not differentiable. We will use the linear discriminant function (unthresholded perceptron):

\[
g (x_1, \ldots, x_n) = \left( w_0 + \sum_{i=1}^{N} w_i x_i \right)
\]

We want to learn \( w_i \)'s that minimize the squared error:

\[
E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - g(\vec{x}_d))^2
\]
The gradient of a function points in the direction of steepest ascent.

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

To find the minimum of \( E(\vec{w}) \), walk in the reverse direction of its gradient.

\[ \vec{w} \leftarrow \vec{w} + \Delta \vec{w} \text{ where } \Delta \vec{w} = -\eta \nabla E(\vec{w}) \]

\( \eta \) is a small positive constant (learning rate).
Derivation of gradient descent

Gradient descent update for weight vector $\Delta \mathbf{w} = -\eta \nabla E(\mathbf{w})$

Component-wise:

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

1. $\frac{\partial E}{\partial w_i} = \sum_{d \in D} \frac{1}{2} \frac{\partial}{\partial w_i} (t_d - \mathbf{w} \cdot \mathbf{x}_d)^2$

2. $= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} \left( t_d - \mathbf{w} \cdot \mathbf{x}_d \right)^2$

3. $= \frac{1}{2} \sum_{d \in D} \left( 2(t_d - \mathbf{w} \cdot \mathbf{x}_d) \frac{\partial}{\partial w_i} (t_d - \mathbf{w} \cdot \mathbf{x}_d) \right)$

4. $= \frac{1}{2} \sum_{d \in D} \left( 2(t_d - \mathbf{w} \cdot \mathbf{x}_d) \frac{\partial}{\partial w_i} (t_d - \mathbf{w} \cdot \mathbf{x}_d) \right)$

5. $= \eta \sum_{d \in D} \left( (t_d - \mathbf{w} \cdot \mathbf{x}_d) \mathbf{x}_{d,i} \right)$

6. $= \eta \sum_{d \in D} \left( (t_d - \mathbf{w} \cdot \mathbf{x}_d) \mathbf{x}_{d,i} \right)$
Gradient-Descent($D$, $\eta$)

Each training example in $D$ is a pair of the form $\langle \bar{x}, t \rangle$, where $\bar{x}$ is the vector of input values, and $t$ is the target output value. $\eta$ is the learning rate.

Initialize each $w_i$ to a small random value
Until the termination condition is met Do
  Initialize each $\Delta w_i$ to zero
  For each $\langle \bar{x}, t \rangle \in D$, Do
    Input the instance $\bar{x}$ to the unit & compute the output $g$
    For each linear unit weight $w_i$ Do
      $\Delta w_i \leftarrow \Delta w_i + \eta(t - g)x_i$
    For each linear unit weight $w_i$ Do
      $w_i \leftarrow w_i + \Delta w_i$
Properties of gradient descent

- The gradient descent procedure is guaranteed to find the weight vector that gives the minimum error regardless of whether the training instances are linearly separable.

- Only one condition to the previous statement: $\eta$ should be small enough. If $\eta$ is too large, gradient descent will overshoot the minimum and can actually diverge! Several ways to deal with this:
  - Reducing $\eta$ as a prescribed function of number of iterations.
  - Reducing $\eta$ every time gradient descent overshoots.
  - Stochastic gradient descent procedure is faster so can be used with a smaller $\eta$. It also helps with local minima if they are present.
  - Using matrix of second derivatives (Hessian) to help choose speed (Newton descent).
The sigmoid function: 

\[ \sigma(y) = \frac{1}{1 + e^{-y}} \]

- Behaves very much like the perceptron in terms of thresholding.
- It is differentiable: 
  \[ \frac{d\sigma(y)}{dy} = \sigma(y)(1 - \sigma(y)) \]
- We can derive gradient descent rules to train:
  (a) One sigmoid unit
  (b) *Multilayer network* of sigmoid units \(\rightarrow\) backpropagation
Training one sigmoid unit

Given a training instance $\langle \vec{x}_d, t_d \rangle$, the error of a single sigmoid unit is:

$$E_d(\vec{w}) = \frac{1}{2} (t_d - o)^2$$

where $o = \sigma(\text{net})$, $\sigma(\text{net}) = \frac{1}{1 + e^{-\text{net}}}$ and $\text{net}(\vec{x}_d) = \sum_{i=0}^{n} w_i x_{d,i}$

Stochastic gradient descent update rule:

$$w_i \leftarrow w_i + \Delta w_i \quad \text{where} \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

\[
\begin{align*}
\frac{\partial E}{\partial w_i} & = \frac{\partial E}{\partial \text{net}} \frac{\partial \text{net}}{\partial w_i} \\
& = \frac{\partial E}{\partial o} \frac{\partial o}{\partial \text{net}} \frac{\partial \text{net}}{\partial w_i} \\
& = -2\frac{1}{2} (t_d - o) o(1 - o) x_{d,i} \\
& = \underbrace{\frac{\partial E}{\partial o}}_{(1)} \underbrace{\frac{\partial o}{\partial \text{net}}}_{(2)} \underbrace{\frac{\partial \text{net}}{\partial w_i}}_{(3)}
\end{align*}
\]
Connectionism

- **Artificial neural networks (ANNs)** are inspired by the strong interconnectedness of the human brain. The brain is the best example we have of a robust learning system!

- **Intelligent behavior** is an **emergent property** from large numbers of simple units ($10^{11}$ neurons) and from a large number of connections between the simple units (each neuron makes synapses with approximately $10^4$ other neurons).

- Massive parallelism allows for computational efficiency.

- Graceful degradation due to distributed representations that spread the representation over many computational units.
Multi-layer Neural Networks

- Why multi-layer networks? To be able to represent arbitrary functions.

  - Multi-layer networks of linear discriminants still only represent linear functions! The building block unit must be nonlinear such as the perceptron.
  - But we will see that there is a problem with the perceptron, so we will use another unit called the sigmoid function.

- Generally, networks are composed of an input layer, hidden layer, and output layer, and activation feeds forward from input to output.

- Patterns of activation are presented at the inputs and the resulting activation of the outputs is computed.

- The value of the weights determine the function computed.
  - Learning is equivalent to tuning these weights for a particular task.
Multilayer Networks and Decision Boundaries

- Lets change our “good ski day” example
  - 3 classes: ski, hike and study.
  - 2 real valued inputs: temperature and new snow amount.

- To evaluate a test instance: (1) input the temperature and new snow amount, (2) compute the hidden node values, (3) compute the output node values, and (4) pick class with largest value at its output node.
The exclusive-or (XOR) problem

We already discussed that a perceptron (a single-layer neural network) cannot represent $XOR(x_1, x_2) = (x_1 \land \neg x_2) \lor (\neg x_1 \land x_2)$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

To see why this is true, we can try to express the problem as a linear equation again: $ax_1 + bx_2 = output$

\[
\begin{align*}
  a \cdot 0 + b \cdot 0 &= 0 \\
  a \cdot 0 + b \cdot 1 &= 1 \\
  a \cdot 1 + b \cdot 0 &= 1 \\
  a \cdot 1 + b \cdot 1 &= 0
\end{align*}
\]
But adding a third bit makes it doable!

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

We can try to express the problem as a linear equation:

$$ax_1 + bx_2 + cy = output$$

$$a \cdot 0 + b \cdot 0 + c \cdot 0 = 0$$
$$a \cdot 0 + b \cdot 1 + c \cdot 0 = 1$$
$$a \cdot 1 + b \cdot 0 + c \cdot 0 = 1$$
$$a \cdot 1 + b \cdot 1 + c \cdot 1 = 0$$

So the equation: $x_1 + x_2 - 2y$ will solve the problem.
\[
\begin{array}{cccc}
 x_1 & x_2 & y & output \\
 0 & 0 & 0 & 0 \\
 0 & 1 & 0 & 1 \\
 1 & 0 & 0 & 1 \\
 1 & 1 & 1 & 0 \\
\end{array}
\]

\[
XOR(x_1, x_2) = x_1 + x_2 - 2y
\]

- Using an extra variable \( y \) helped solve the problem
- What is \( y \) and how can we compute it?
  - Notice \( y = x_1 \land x_2 \). So it can be computed by a perceptron.
  - The computation of \( y \) will be called a hidden unit.
  - Once we have \( y \), \( XOR \) can be computed by a perceptron which has inputs \( x_1, x_2 \) and \( y \).
A Multilayer Net for XOR

Output Unit

Hidden Unit

Input Units

0.5

1.5

-2

+1

+1

+1

+1

+1
Hidden Units

- Hidden units are nodes that are situated between the input nodes and the output nodes.
- Hidden units allow a network to learn arbitrary functions.
- Given too few hidden units, the network may not be able to represent all of the necessary generalizations.
- Given too many hidden units, a neural net will simply memorize the input patterns.
- How do hidden units learn useful representations? For instance, how can a neural network learn to represent $y = x_1 \land x_2$ at a hidden node for the XOR problem.
Neural network design

**Architecture**: the pattern of nodes and connections between them. This is typically designed manually and fixed. Normally the network consists of a layered topology with units in any layer receiving input from all units in the previous layer. The most common layered topology is an input layer, 1 or 2 hidden layers, and an output layer.

**Activation function**: the function that produces an output based on the input values received by a node. This is also fixed. It can be the sigmoid function, hyperbolic tangent ($tanh$) among other possibilities.

**Learning algorithm**: (training method) the method for determining the weights of the connections. Backpropagation is the most commonly used.
Learning the connection weights

- How can we learn the connection weights in a multilayer network?
  - Gradient descent is a good general search technique over continuously parameterized hypotheses. We have already used it for training linear discriminants.
  - We have to define the error of the network and this error has to be differentiable with respect to the parameters of the hypothesis (weights for ANNs).

- What unit should we use at each node?
  - Multiple layers of linear discriminants still produce a linear discriminant. We need non-linearity at the level of the individual node.
  - The perceptron is nonlinear, but it is not differentiable at the threshold. Hence, we can’t learn its weights using gradient descent.
  - We need a differentiable threshold unit.
Training multilayer networks - Backpropagation

Named for the backward propagation of errors during learning, using the generalized delta rule.

1. **Feedforward training of input patterns**
   Each input node receives a signal, which is broadcast to the hidden units. Each hidden unit computes its activation and broadcasts it to the output units.

2. **Backpropagation of errors**
   Each output node compares its activation with the desired output. The error is propagated backwards to upstream nodes.

3. **Weight adjustment**
   The weights of all links are computed simultaneously based on the error propagated backwards.
Computing the output of the feedforward network

- Network topology: Each input node is linked to each hidden node. Each hidden node is linked to each output node.

- $x_{ij}$: The input from unit $i$ to unit $j$. Input nodes are also considered units in this notation.

- $w_{ij}$: The weight associated with the link between units $i$ and $j$. 
Computing the output of the feedforward network

- Each input unit broadcasts its value to all of the hidden units. Each hidden unit sums its input signals and applies its activation function to compute its output signal.

\[ \forall h \in \text{Hidden nodes} \quad o_h = \sigma \left( \sum_{i \in \text{Input nodes}} w_{hi} x_{hi} \right) \]

- Each hidden unit sends its signal to the output units. Each output unit sums its input signals and applies its activation function to compute its output signal.

\[ \forall k \in \text{Output nodes} \quad o_k = \sigma \left( \sum_{h \in \text{Hidden nodes}} w_{kh} x_{kh} \right) \]
Computing the update for output nodes

- Error for all training instances:

\[ E_d(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{Output nodes}} (t_{kd} - o_{kd})^2 \]

- Error for a single training instance:

\[ E_d(\vec{w}) = \frac{1}{2} \sum_{k \in \text{Output nodes}} (t_k - o_k)^2 \]

- We will take training instances one at a time (second error function) and use stochastic gradient descent.
Computing the update for output nodes

- Recall training a single sigmoid unit: $\Delta w_i = \eta(t - o) o (1 - o) (x_i)$

- Output nodes are very similar:
  - Error term $\delta_k$ for output unit $k$:
    $$\delta_k \leftarrow (t_k - o_k) o_k (1 - o_k)$$
  - Weight update for output unit $k$:
    $$\Delta w_{kh} = \eta \delta_k x_{kh}$$
Computing the update for hidden nodes

- Hidden units are bit more involved because the update must take into account the indirect ways it can influence the output nodes.

- The backpropagated error at hidden node $h$:

$$
\delta_h = o_h(1 - o_h) \sum_{k \in \text{Output nodes}} \delta_k w_{kh}
$$

- Weight update for hidden unit $h$:

$$
\Delta w_{hi} = \eta \delta_h x_{hi}
$$
Finally the algorithm

- Initialize all weights with small random values.

- While the stopping condition is not true.
  
  For each training example (one at a time), do

  1. Compute the values of all hidden and output nodes (feedforward stage).

  2. For each output unit $k$, calculate its error $\delta_k$.

  3. For each hidden unit $h$, calculate its error $\delta_h$.

  4. Update all weights

     \[ w_{ji} \leftarrow w_{ji} + \Delta w_{ji}, \text{ where } \Delta w_{ji} = \eta \delta_j x_{ji} \]

This is the stochastic version of backpropagation since updates are computed one training instance at a time.
Stochastic gradient descent

• Stochastic gradient descent may be preferred if local minima are known to exist in the error surface.
  – We know only one minima (the global minimum) exists when we are training a single linear unit.
  – However, we’ll soon be looking at training many non-linear units simultaneously (neural networks). In that setting having many local minima is almost guaranteed.

• Stochastic gradient descent uses training instances one at a time to update the weights (as opposed to regular gradient descent which accumulates the effect off all training instances and then updates the weights)
The learning rate

\[ w_{ji} \leftarrow w_{ji} + \eta \delta_j x_i \]

- The learning rate, \( \eta \), controls how big the weight changes are for each iteration.

- Ideally, the learning rate should be infinitesimally small, but then learning is very slow.

- If the learning rate is too high then the system can suffer from severe oscillations.

- For fast learning, you want the learning rate to be as large as possible without resulting in oscillations.
How long should you train the net?

- Typically, many iterations are needed (often thousands).
- The goal is to achieve a balance between correct responses for the training patterns and correct responses for new patterns. (That is, a balance between memorization and generalization.)
- If you train the net for too long, then you run the risk of overfitting.
- Possible stopping conditions:
  - Fixed number of iterations
  - Threshold on training set error (e.g., 5%)
  - Increased error on a validation set
Comments on Training

- No convergence guarantee, may oscillate or reach a local minima.
- However, in practice, many large networks have been adequately trained on large amounts of data for realistic problems.
- Adding momentum to the update helps avoid local minima.
- To avoid local minima, can run several trials from different random weights and:
  - Take the result with the best training or validation performance, OR
  - Build a committee of networks that vote during testing, possibly weighting vote by training or validation accuracy
- Backpropagation easily generalizes to acyclic networks with any number of hidden node layers and even to any directed acyclic network (no organized layers).
Expressive Capabilities of ANNs

Boolean functions:

- Every boolean function can be represented by network with a 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 a network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].