Decision tree:

A decision tree represents a function that takes a input as vector of attribute values and returns a decision a single output value. It can be discrete or continuous. Input will be classified as true or false.

As an example we will build a decision tree to decide whether to wait for a table at a restaurant. The aim here is to learn a definition for the goal predicate willwait.

Attributes:

1. Alternate: whether there is suitable alternate restaurant nearby
2. Bar: it has a comfortable bar area to wait in
3. Fri/Sat: true on Fridays and Saturdays
4. Hungry: whether we are hungry
5. Patrons: how many people are in restaurant
6. Price: the restaurant price range
7. Raining: whether it is raining outside
8. Reservation: whether we made a reservation
9. Type: the kind of restaurant (French, Italian, Thai, burger)
10. Wait Estimate: the wait estimated by the host
Expressiveness of decision trees:

A boolean decision tree is logically equivalent to the assertion that the goal attribute is true if and only if the input attribute satisfy one of the paths leading to a leaf with value true in propositional logic.

\[ \text{Goal} \iff (\text{path}_1 \lor \text{path}_2 \lor \ldots) \]

Each path is a conjunction of attribute value tests required to follow that path.

Decision tree:

A decision tree for deciding whether to wait for a table:

```
  Patrons?
   /     \
  /       \
No       Yes
  /     \
No       Yes
  /     \
   >60   30-60  10-30  0-10
  /     \
No     Alternate
  /     \
No     Yes
  /     \
   Reservation?
  /     \
No     Yes
  /     \
No     Yes
  /     \
   Fri/Sat?
  /     \
No     Yes
  /     \
No     Yes
  /     \
   Bar?
   /     \n  No     Yes
  /     \
No     Yes
```

The boolean decision tree consists of an \((x, y)\) pair where \(x\) is a vector of values for the input attributes and \(y\) is a single boolean output value.
### Inference:

<table>
<thead>
<tr>
<th>Example</th>
<th>Input Attribute</th>
<th>Goal</th>
<th>Est</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Alt</td>
<td>Box</td>
<td>Fri</td>
</tr>
<tr>
<td>$x_1$</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$x_2$</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$x_3$</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$x_4$</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>$x_5$</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>$x_6$</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$x_7$</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$x_8$</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$x_9$</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$x_{10}$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$x_{11}$</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$x_{12}$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

There are four cases to consider for these recursive branches:

1. If the remaining examples are all positive or negative, we can answer Yes or No.

2. If there are some positive and negative examples, then choose the best attribute to split them.

3. If there are no examples left, it means that no example has been observed for this combination of attribute value and return a default value.
4. If there are no attribute left but both positive and negative examples, it means that the examples have exactly the same description but different classification.

Splitting the example by testing an attributes:

Algorithm:

Function DECISION-TREE-LEARNING (examples, attributes, default) returns a tree.

if examples is empty then return default (parent examples)
else if all examples have the same classification then return the classification.
else if attribute is empty then return majority value (examples)
else
    Best <- choose-attributes (attributes, examples)
tree ← a new decision tree with root test Best

N ← majority-value (example)

for each value v_k of Best do

exs ← {ex: ex examples and e.Best = v_k}

subtree ← Decision Tree Learning (exs, attributes - Best, examples)

add a branch to tree with label (Best = v_k) and subtree = subtree

return tree

The decision tree induced from the 12-example training set:

```
Patrons?  
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
<tr>
<td>Some</td>
</tr>
<tr>
<td>Full</td>
</tr>
</tbody>
</table>
  
| No    |
| Yes   |

Hungry?  
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
</tr>
</tbody>
</table>

Type     
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>French</td>
</tr>
<tr>
<td>Italian</td>
</tr>
<tr>
<td>Thai</td>
</tr>
<tr>
<td>Burger</td>
</tr>
</tbody>
</table>

| Yes   |
| No    |
| Fri/Sat?   |

| Yes   |
| No    |

| Yes   |
| No    |
```
Backpropagation Learning

**Neural Network:**

A neural network is a collection of units connected together. The properties of a network are determined by its topology and the properties of its "neurons." It implements a linear classifier as the kind described in the preceding section.

**Neural network structures:**

Neural networks are composed of nodes (or) units connected by directed links.

A link from unit i to unit j serves to propagate the activation a_j from i to j. Each link also has a numeric weight w_{ij} associated with it.

which determines the strength and sign of the connection.

Each unit i first computes a weighted sum of its inputs.

There are two fundamentally distinct ways:

- **Feed-forward networks** have connections only one in one direction - it forms directed acyclic graphs. A feed-forward network represents a function of its current input. It has no internal state other than the weights.

- **Recurrent networks** have its outputs back into its own inputs.

It has one or more layers of hidden units that are learning problems with a single output variable. Hidden units have not connected to the outputs of the network.

**Single-layer feed-forward neural networks:**

In the input connected directly to the output & called a single-layer neural network on a perceptron network.

Depending on the type of the activation function used, it processes will be almost no perceptron learning rule on gradient descent rule for logistic regression.

**Daddy:** The solution problem is easily represented as a decision tree, but not be linearly separable.
Multilayer feed-forward neural networks:

It computing large numbers of units into networks of arbitrary depth, the problem was that nobody knew how to train such networks.

It composed of nested nonlinear soft threshold functions: neural networks as a tool for doing nonlinear regression.

For any particular network structure, it is harder to characterize exactly which functions can be represented and which ones cannot.

Learning in multilayer networks:

The interaction among the learning problems when the network has multiple outputs.

The major computation comes from the addition of hidden layers to the network. The error at the output layer & clear, the error at the hidden layers.

The training data don't say what value the hidden nodes should have. It turns out that we can back propagate.

The error from the output layer to the hidden layer: the back propagation procedure emerges directly from a derivation of the overall error gradient.

At the output layer, the weight update rule is identical. The weight update rule becomes:

\[ w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times A_k \]

To update the connections from the input unit and the hidden units, we need to define the quantities to the error term present for the output nodes.

The \( A_k \) values are divided according to the strength of the connection from the hidden node to the output node. The \( A_k \) values are propagated back to provide the \( A_j \) values for the hidden layer.
The propagation rule for the Δ values:

\[ Δ_j = g'(h_j) \sum_k w_{jk} Δ_k. \]

The weight-update rule for the weights between the input and hidden layer is essentially identical to the update rule for the output layer:

\[ w_{ij} \leftarrow w_{ij} + \alpha \times a_i x Δ_j \]

The back-propagation process can be summarized as follows:

- Compute the Δ values for the output units, using the observed error.
- Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
  - Propagate the Δ values back to the previous layer.
  - Update the weights between the two layers.

Algorithm:

The algorithm is a back-propagation algorithm for learning in multi-layer networks.

Refer page no: 745 ⇒ Figure 24.