MACHINE LEARNING

Learning

A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience.

Machine learning involves automatic procedures that learn a task from a series of examples. The most convenient source of examples is data

A learnable system is

- More flexible – being able to respond to new problems and situations.
- Easier to program.

Designing a machine learning approach involves a number of design choices, including choosing the type of training experience, the target function to be learned, a representation for this target function, and an algorithm for learning the target function from training examples.

Types of Learning

i). **Supervised learning: -**Supervised learning is the learning situation in which both the inputs and outputs can be perceived. Sometimes a friendly teacher can supply the outputs.

ii). Reinforcement learning: - Reinforcement learning is a type of learning situation in which the agent does not know the outcomes but is given some form of feedback on evaluating its action. It is however not told the correctness of its action.

iii). Unsupervised learning: - Unsupervised learning is a type of learning in which the no hint is given at all about the correct input.

Symbolic methods:

Learned concepts are represented using the KR languages discussed earlier.

- Two main approaches:
  i). Search the space of possible concepts to find one that matches the examples.
  ii). Building up the best *decision tree*.
- Sub-symbolic methods - Genetic Algorithms & Neural Networks.

APPLICATIONS OF MACHINE LEARNING

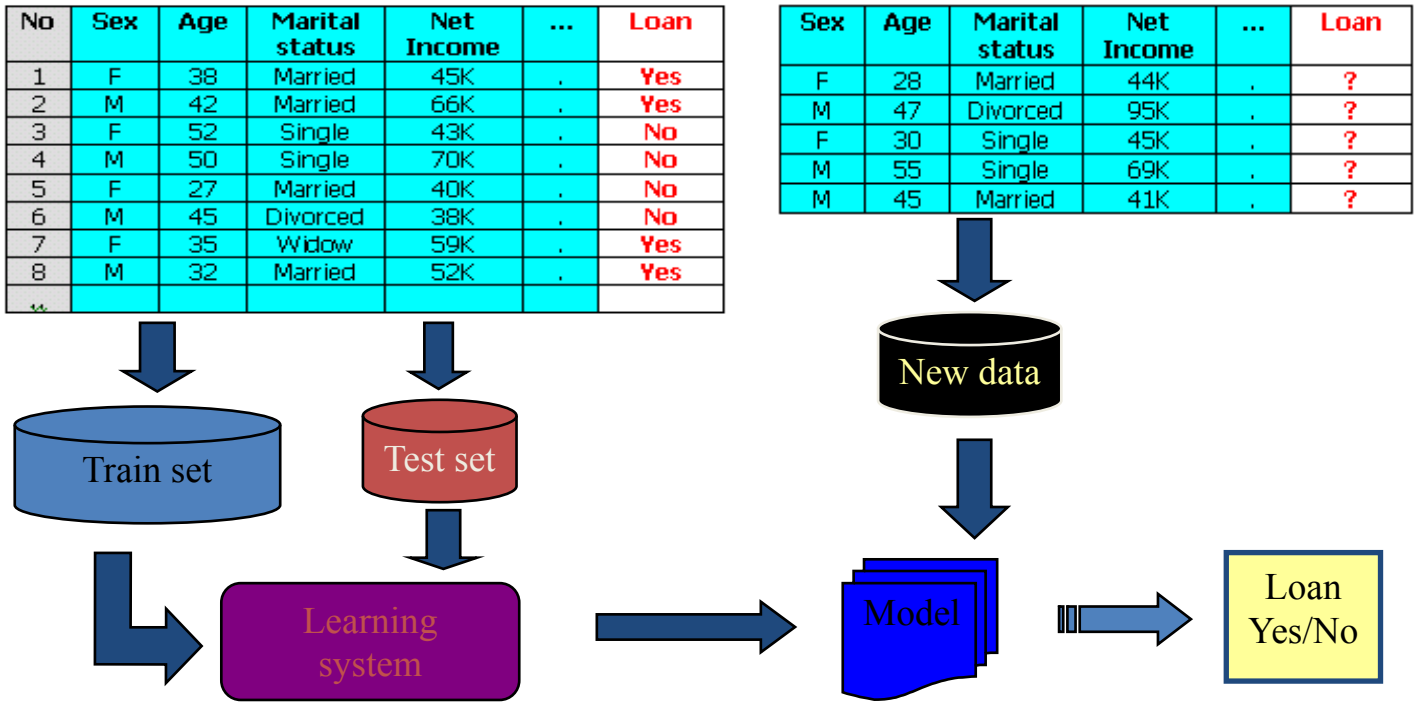
The main aim of machine learning is to make computer systems that can learn. If machines learn then their ability to solve problems will be enhanced considerably. In research learning has found applications that are related to knowledge acquisition; planning and problem solving. There some areas, that are side effects of research in Machine learning, that have seen intensive research in recent times that include data mining. Specifically some of these applications include:

- Where there are very many examples and we have no function to generate the outputs, machine learning techniques can be used to allow the system to search for suitable functions (hypotheses).
- Where we have massive amount of data and hidden relationships, we can use machine learning techniques to discover the relationships (data mining).
- Sometimes machines cannot be built to do what is required due to some limitations, if machines can learn then they can improve their performance.
- Where too much knowledge is available such that it is impossible for man to cope with it, then machines can be used to learn as much as possible.
- Environments change over time, so machines can adapt instead of re-design new ones. New knowledge is being discovered by humans, new vocabulary arise, new world events stream in and therefore new AI systems should be re-designed. Instead of doing this, learning systems may be built.

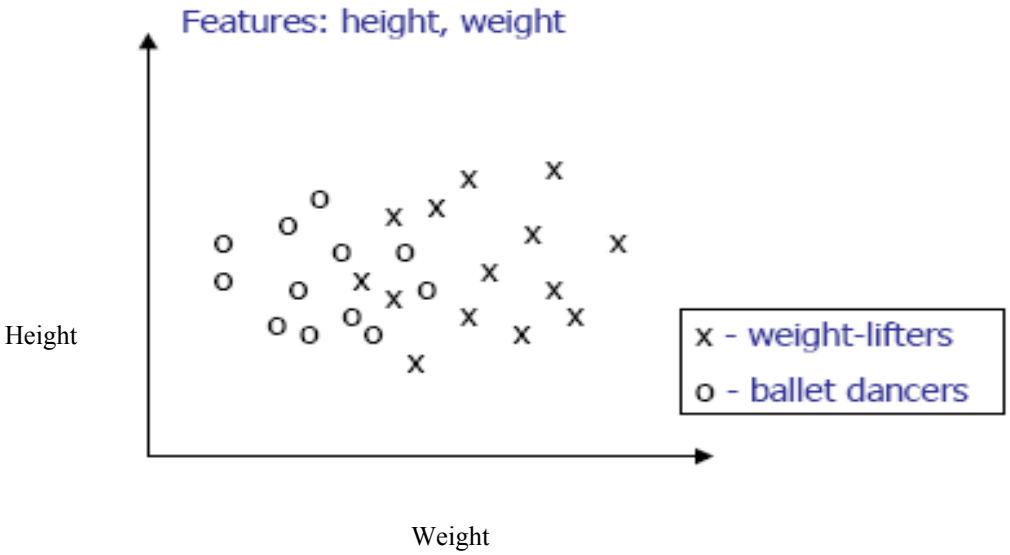
Machine Learning Models

- Classification
- Regression
- Clustering
- Time series analysis
- Association Analysis
- Sequence Discovery

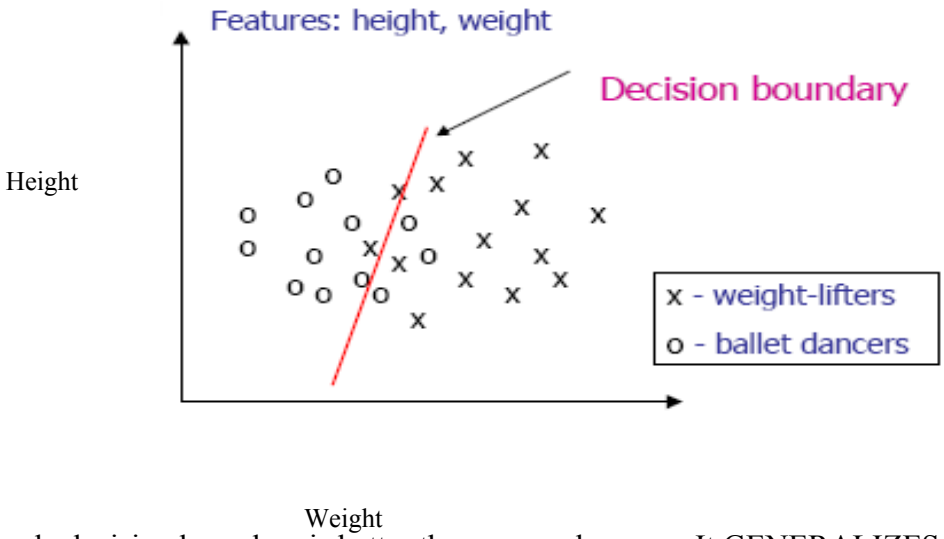
Classification example1



Classification example 2

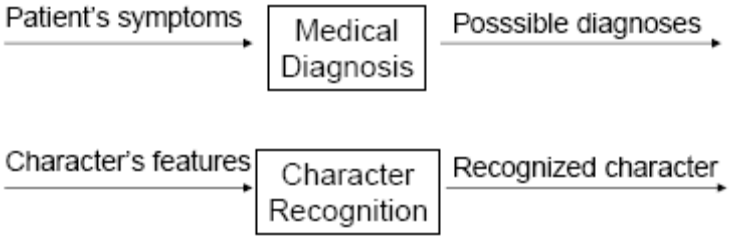


Classification example - Simple Model



Note: A simple decision boundary is better than a complex one - It GENERALIZES better.

Classification tasks:
The following are examples of classification tasks:



TECHNIQUES USED IN MACHINE LEARNING
Machine learning depends on several methods and which includes the following.

- Decision tree learning
- Artificial Neural Networks
- Instance Based Methods (CBR, k-NN)
- Bayesian Networks
- Evolutionary Strategies
- Support Vector Machines
- Inductive logic programming
- Explanation Based Learning
- Reinforcement learning

Learning Problem
Learning = Improving with experience at some task

- Improve over Task T
- with respect to performance measure P
- based on experience E

Example
A computer program that learns to play checkers might improve its performance as measured by its ability to win at the class of tasks invoving playing checkers games, through exprience obtained by playing games against

itself. In general, to a well-defined learning problem, we must identity these three features: the class of tasks, the measure of performance to be improved, and the source of exprience.

Checkers learning problem

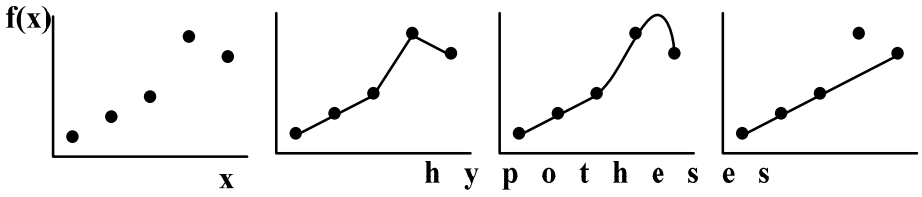
- T: Play checkers
- P: Percent of games won in world tournament..
- E: games played against self..

In order to design such a system, the following must be considered:

- What exactly should be learned
- How shall it be represented
- What specific algorithm to learn it

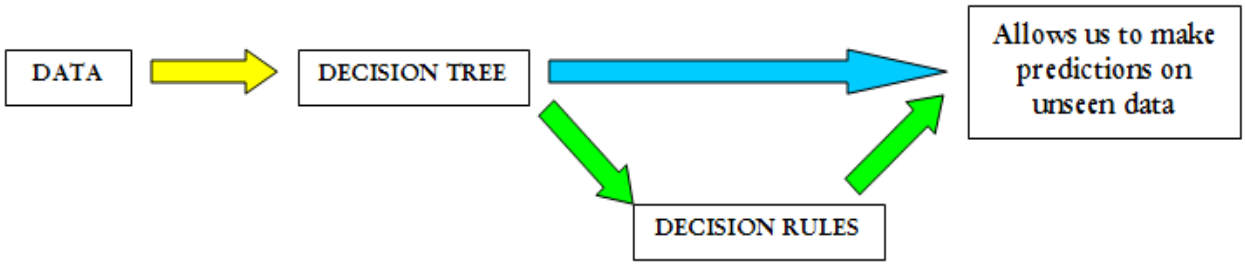
i). Inductive Learning (learning from examples)

Pure inductive inference problem seeks to find a hypothesis, **h**, that approximates the function, **f,** given the example **(x, f(x))**. Consider a plot of points. The possible curves that can be joined suggest various functions (hypotheses, h) that can approximate the original function. The aim is to find the hypothese that fits well on the training examples. *h* is used to predict the values of the unseen examples.
Where there is preference to hypothesis to a given example beyond consistency, we say there is a **bias**.
Having fixed the bias, learning can be considered as search in the hypothesis space which is guided by the used preference bias.

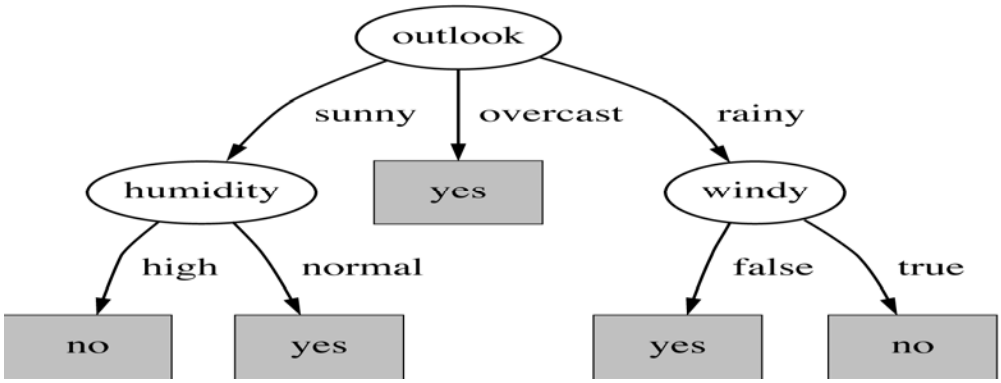


Consider an agent that has a reflex learning element that updates global variable, examples, and that it holds a list of pairs of (percept, action). When it is confronted with a percept and it is looking for an action it first checks the list. If the percept is there then it applies the action, otherwise it must formulate a hypothesis, h, that is used for selecting the action. If the agent instead of applying a new hypothesis adjusts the old hypothesis, then we say incremental learning occurs.
Note: the examples may be positive or negative.
One of the most known methods of inductive learning: learning decision trees

a) **Decision trees: -** This is a method that uses inductive inference to approximate a target function, which will produce discrete values. It is widely used, robust to noisy data, and considered a practical method for learning disjunctive expressions. Generally, a decision tree is constructed by looking for regularities in data as shown below**.**



Example
Decision Tree for PlayTennis



In these trees, the inputs are objects or situations described by a set of properties (attributes) while outputs are either yes or no decisions. Each node consists of a test to the value of one of the properties and the branches from the nodes are labelled with possible values of test result. Each leaf specifies the Boolean value if that leaf is reached. There are generally used as simple representation for classifying examples

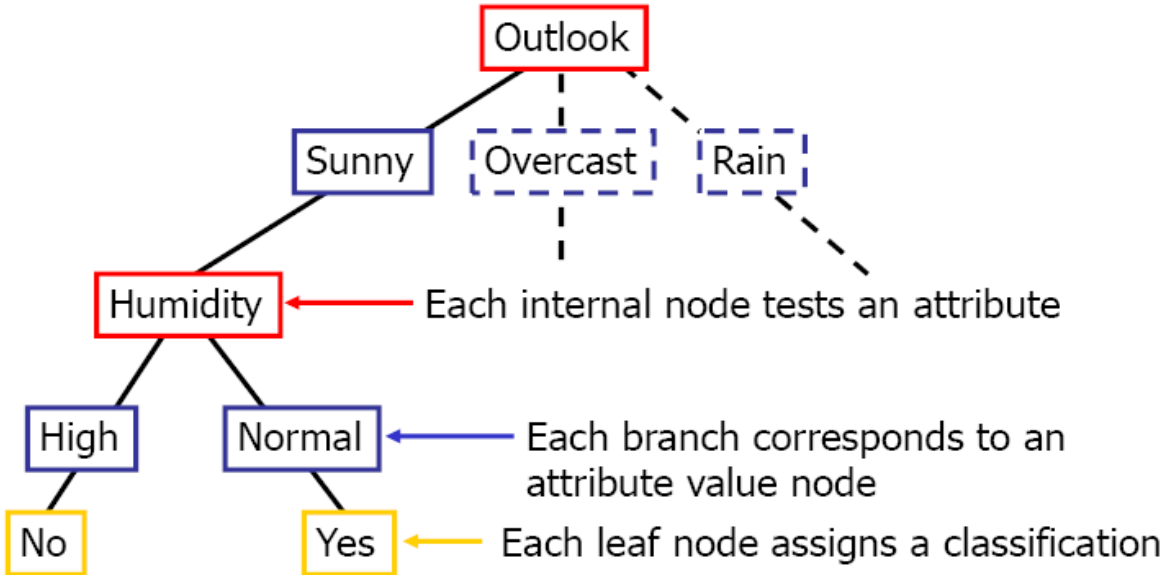
Elements of the decision tree:

- nonleaf (internal) nodes are labelled with attributes (*A*)
- arcs out of a node are labelled with possible attribute values of *A*
- leaf nodes are labelled with classifications (Boolean values –yes/no - in the simplest case)

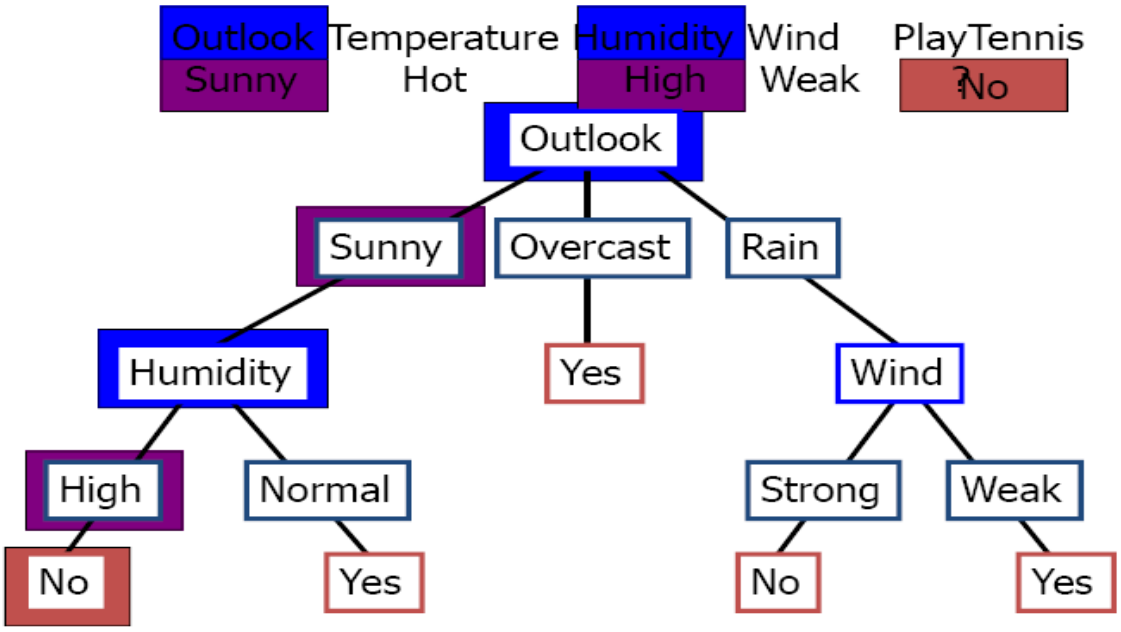
Decision Trees: definition

A decision tree over the attributes A_1, A_2,.., A_n, and B is a tree in which

- each non-leaf node is labelled with one of the attributes A_1, A_2, ..., and A_n
- each leaf node is labelled with one of the possible values for the goal attribute B
- a non-leaf node with the label A_i has as many outgoing arcs as there are possible values for the attribute A_i; each arc is labelled with one of the possible values for A_i

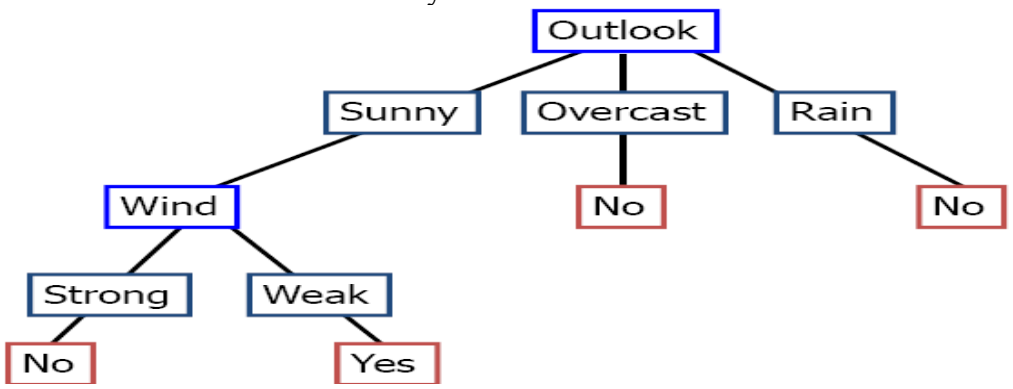


Decision Tree for PlayTennis



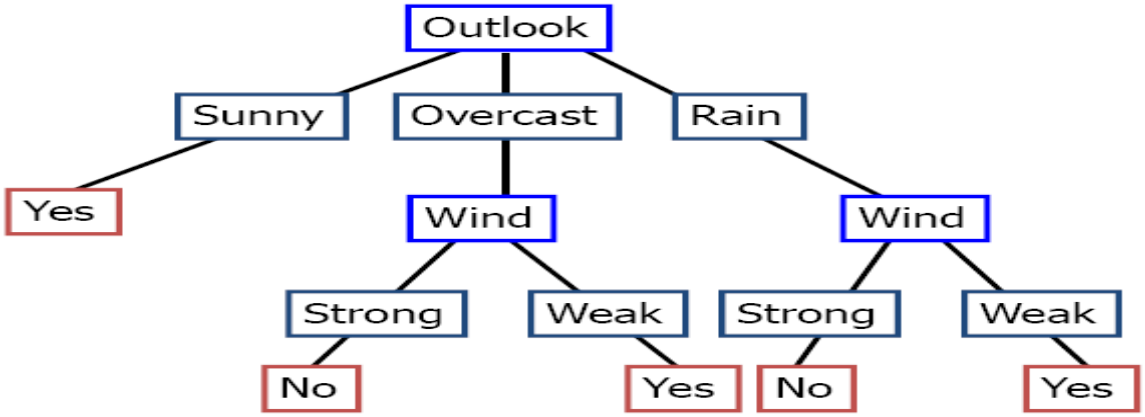
Decision Tree for Conjunction

Outlook=Sunny ∧ Wind=Weak

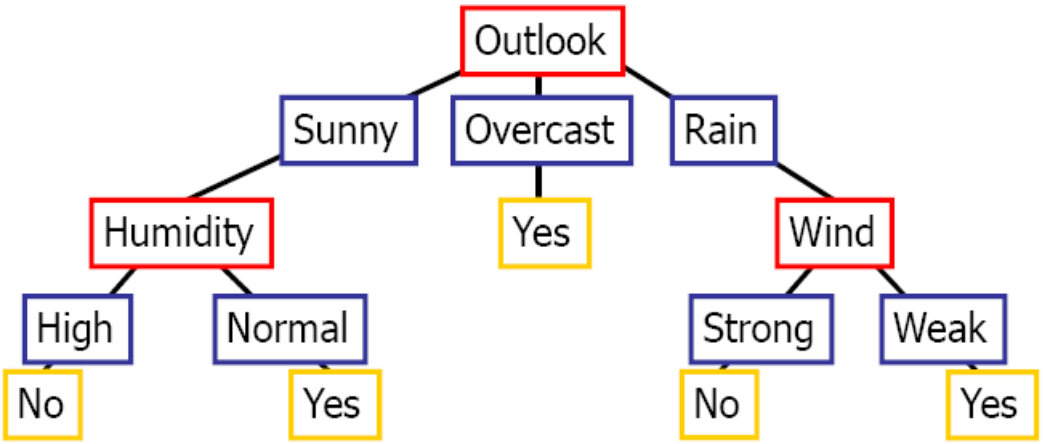


Decision Tree for Disjunction

Outlook=Sunny ∨ Wind=Weak



Generally decision trees represent disjunctions of conjunctions as shown below:



(Outlook=Sunny ∧ Humidity=Normal)
∨ (Outlook=Overcast)
∨ (Outlook=Rain ∧ Wind=Weak)

Appropriate Problems for Decision Tree Learning
Decision tree learning is generally best suited to problems with the following characteristics:

- Instances are represented by **attribute-value pairs**.
  - There is a finite list of attributes (e.g. hair colour) and each instance stores a value for that attribute (e.g. blonde).
  - When each attribute has a small number of distinct values (e.g. blonde, brown, red) it is easier for the decision tree to reach a useful solution.
  - The algorithm can be extended to handle real-valued attributes (e.g. a floating point temperature)
- The target function has **discrete output values**.
  - A decision tree classifies each example as one of the output values.
    - Simplest case exists when there are only two possible classes (**Boolean classification**).
    - However, it is easy to extend the decision tree to produce a target function with more than two possible output values.
  - Although it is less common, the algorithm can also be extended to produce a target function with real-valued outputs.
- Disjunctive descriptions may be required.
  - Decision trees naturally represent disjunctive expressions.
- The training data may contain errors.
  - Errors in the classification of examples, or in the attribute values describing those examples are handled well by decision trees, making them a robust learning method.
- The training data may contain missing attribute values.
  - Decision tree methods can be used even when some training examples have unknown values (e.g., humidity is known for only a fraction of the examples).

After a decision tree learns classification rules, it can also be re-represented as a set of if-then rules in order to improve readability.

Decision Tree Representation
A **decision tree** is an arrangement of tests that provides an appropriate classification at every step in an analysis. In general, decision trees represent a disjunction of conjunctions of constraints on the attribute-values of instances. Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions

More specifically, decision trees classify **instances** by sorting them down the tree from the **root node** to some **leaf node**, which provides the classification of the instance. Each node in the tree specifies a **test** of some **attribute** of the instance, and each **branch** descending from that node corresponds to one of the possible **values** for this attribute.
An instance is classified by starting at the root node of the decision tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute. This process is then repeated at the node on this branch and so on until a leaf node is reached.

Example:

Problem: decide whether to wait for a table at a restaurant, based on the following attributes:

1. Alternate: is there an alternative restaurant nearby?
2. Bar: is there a comfortable bar area to wait in?
3. Fri/Sat: is today Friday or Saturday?
4. Hungry: are we hungry?
5. Patrons: number of people in the restaurant (None, Some, Full)
6. Price: price range ($, $$, $$$)
7. Raining: is it raining outside?
8. Reservation: have we made a reservation?
9. Type: kind of restaurant (French, Italian, Thai, Burger)
10. WaitEstimate: estimated waiting time (0-10, 10-30, 30-60, >60)

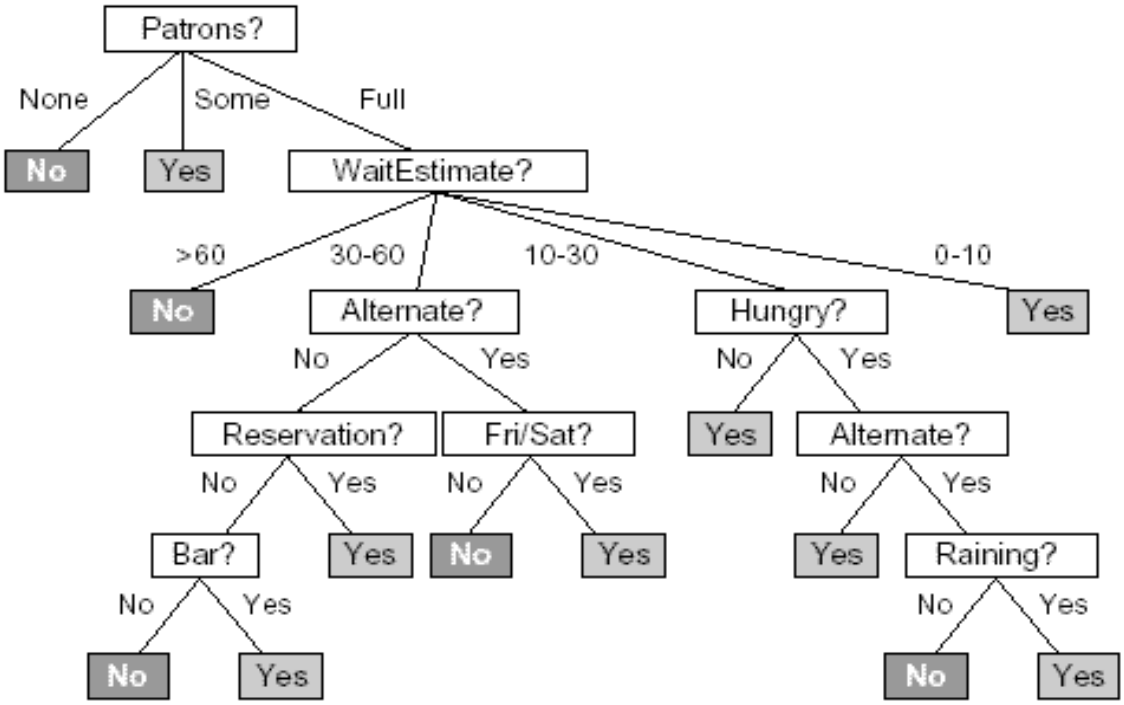
Attribute-based representations

- Examples described by attribute values (Boolean, discrete, continuous)

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*Wait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

Classification of examples is positive (T) or negative (F)

One possible representation for hypotheses
E.g., here is the "true" tree for deciding whether to wait:



Consider:

Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*
Yes	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*

The logical version is given by:

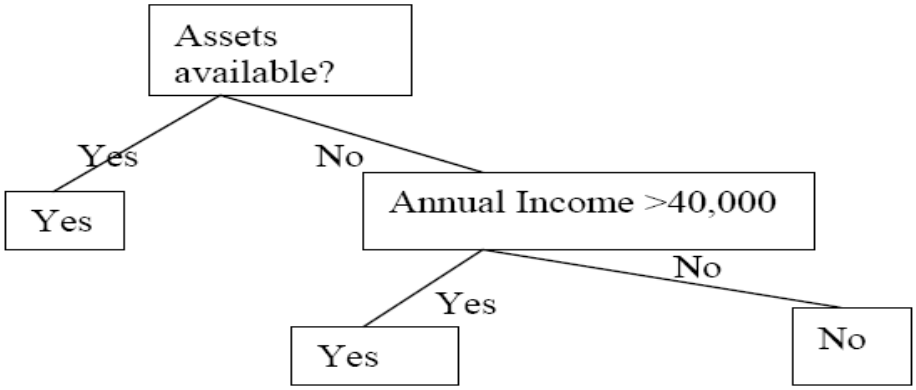
$\forall$ X Patrons(X,Full) $\wedge$ WaitEstimate(X, 0-10) $\wedge$ Hungry(X, No) $\Rightarrow$ WillWait(X).

Examples of problems solved be Decision trees:

Medical diagnosis, Credit risk analysis, Object classification for robot manipulator

Example 2

Applicant	Annual income	Assets	Age	Dependants	Decision
Okello	50,000	100,000	30	3	Yes
Kamoro	70,000	None	35	1	Yes
Mulei	40,000	None	33	2	No
Wanjiru	30,000	250,000	42	0	Yes



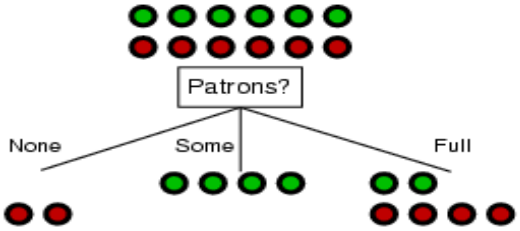
Logically: $\forall$ A has_assets(A) $\vee$ annual_income(A, >40,000) $\Rightarrow$ Approve_loan_for(A).

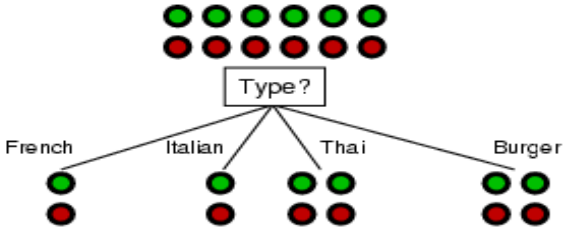
Inducing decision trees from examples
Decision trees may also be obtained from examples. In this case, a table is used and the goal conditions for yes are collected together. Trivially, just take the attributes and enumerate its possible options.
The table is processed attribute by attribute and selecting the attribute that minimizes noise or maximizes information. A typical example is ID3 algorithm.

Top-Down Induction of Decision Trees ID3

1. A ← the "best" decision attribute for next *node*
2. Assign A as decision attribute for *node*
3. For each value of A create new descendant
4. Sort training examples to leaf node according to the attribute value of the branch
5. If all training examples are perfectly classified (same value of target attribute) stop, else iterate over new leaf nodes.

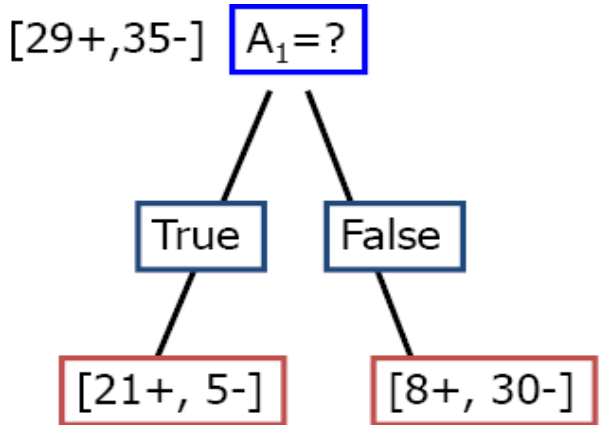
Generally, a good attribute splits the examples into subsets that are (ideally) "all positive" or "all negative"

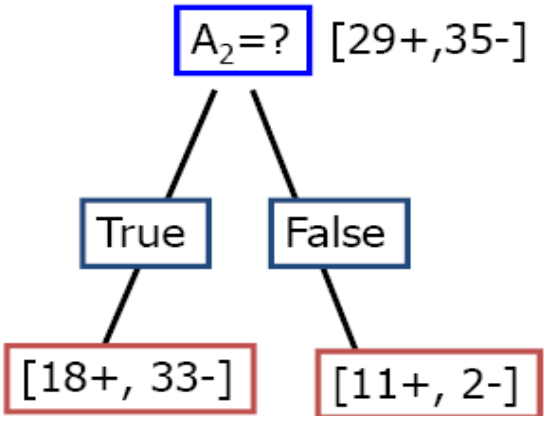




From the above, *Patrons* is a better choice

Which Attribute is "best"?





Entropy
In order to define information precisely, it is important to define a measure commonly used in information theory called entropy, that characterises the (im)purity of an arbitrary collection of examples. Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this Boolean classification is:

$$\textit{Entropy}(S) \equiv -p_+ \log_2 p_+ - p_- \log_2 p_-$$

Where p_+ is the proportion of positive examples in S and p_- is the proportion of negative examples in S. in all calculations involving entropy 0 log 0 is defined to be 0.

Generally, it is based on a notion of impurity in data. Imagine a set of boxes and balls in them

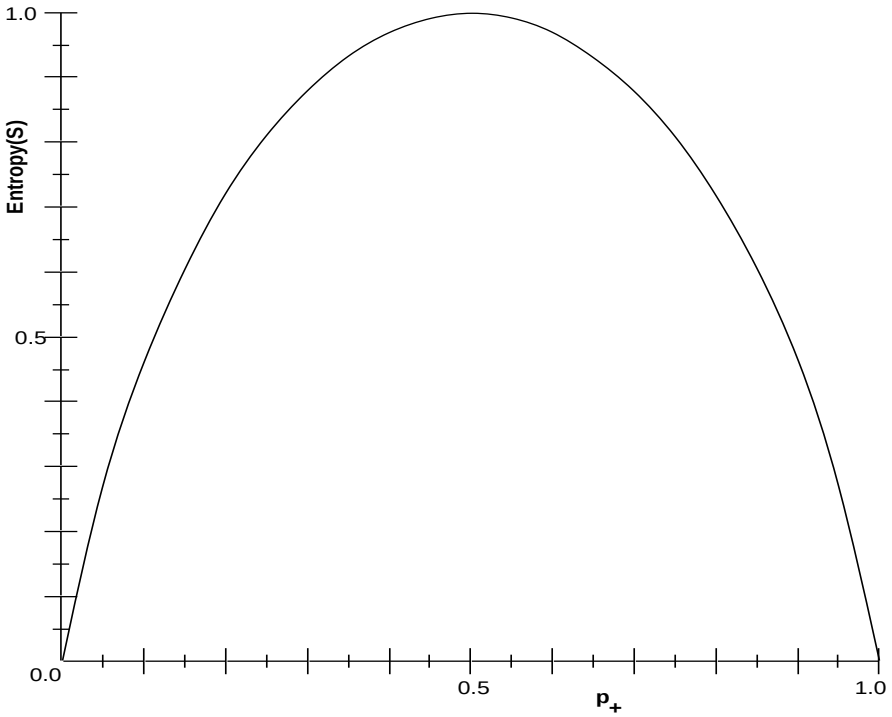
If all balls are in one box

- This is nicely ordered – so scores low for entropy

Calculate entropy by summing over all boxes

- Boxes with very few in scores low
- Boxes with almost all examples in scores low

The following is entropy function relative to a Boolean classification, as the proportion, $p_{+,}$ of positive examples varies between 0 and 1.



Each category adds to the whole measure

When p_i is near to 1

- (Nearly) all the examples are in this category
  So it should score low for its bit of the entropy
- $\log_2(p_i)$ gets closer and closer to 0
  And this part *dominates* the overall calculation
  So the overall calculation comes to nearly 0 (which is good)

When p_i is near to 0

- (Very) few examples are in this category
  So it should score low for its bit of the entropy
- $\log_2(p_i)$ gets larger (more negative), but does not dominate
- Hence overall calculation comes to nearly 0 (which is good)

Generally, Entropy(S)= expected number of bits needed to encode class (+ or -) of randomly drawn members of S (under the optimal, shortest length-code).
One interpretation of information theory is that it specifies the minumum number of bits of information needed to encode the classification of arbitrary member of S. e.g p_+ is 1, the receiver knows the drawn example will positive, thefore no message need be sent, and the entropy is zero, on the other hand, if p_+ is 0.5, one bit will be required to indicate whether the drawn example is positive or negative. if p_+ is 0.8, then collection of messages can be encoded using on average less than 1 bit per message by assigning shorter codes to collection of positive examples and longer codes to collection of negative examples.
Generally, if the target attribute can take on *c* different values, the entropy of S relative to this *c*-wise classification is defined:

$$Entropy(S) = \sum_{i=1}^{n} -p_i \log_2(p_i)$$

Where p_i is the proportion S belonging to class *i*

ID3 uses a measure called Information Gain, based on a notion of entropy i.e. "Impurity in the data", and which is used to choose which node to put in next
Node with the highest information gain is chosen, when there are no choices, a leaf node is put on

Information Gain
Given set of examples S and an attribute A

- A can take values $v_1 \ldots v_m$
- Let S_v = {examples which take value v for attribute A}

Calculate Gain(S,A)

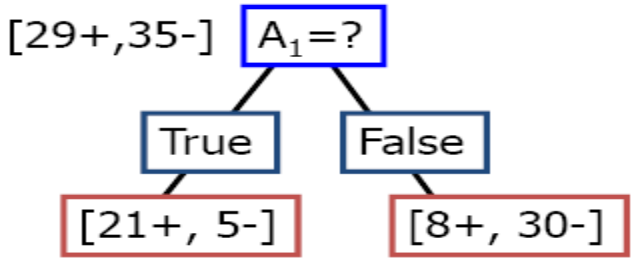
- Estimates the reduction in entropy we get if we know the value of attribute A for the examples in S

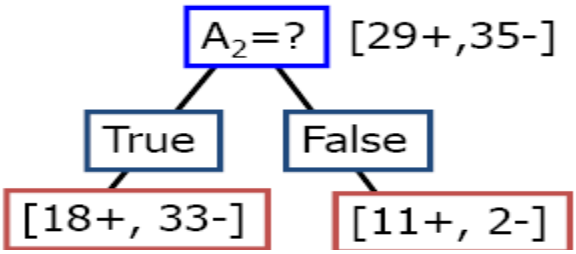
Generally, Gain(S,A) is the expected reduction in entropy due to sorting S on attribute A

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

Example

Entropy([29+,35-]) = -29/64 $\log_2$ 29/64 – 35/64 $\log_2$ 35/64
= 0.99

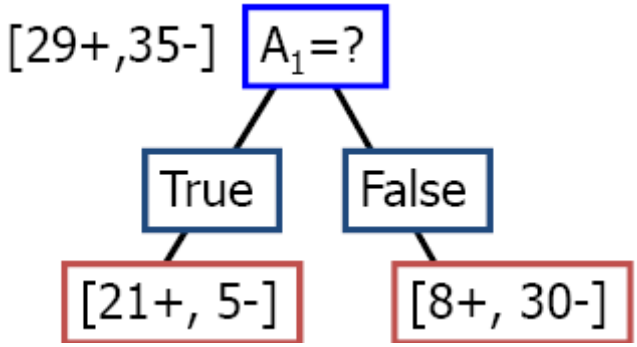


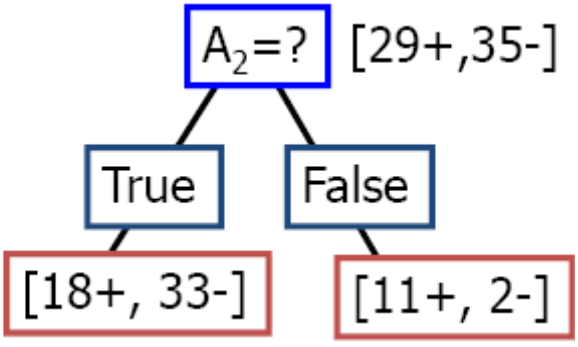


Information Gain becomes

Entropy([21+,5-]) = 0.71
Entropy([8+,30-]) = 0.74
Gain(S,A_1)=Entropy(S)
-26/64*Entropy([21+,5-])
-38/64*Entropy([8+,30-])
=0.27

Entropy([18+,33-]) = 0.94
Entropy([8+,30-]) = 0.62
Gain(S,A_2)=Entropy(S)
-51/64*Entropy([18+,33-])
-13/64*Entropy([11+,2-])
=0.12





The ID3 Algorithm

- *Given a set of examples, S*
  - *Described by a set of attributes A_i*
  - *Categorised into categories c_j*

1. Choose the root node to be attribute A

- *Such that A scores highest for information gain*
  Relative to S, i.e., gain(S,A) is the highest over all attributes

2. For each value v that A can take

- *Draw a branch and label each with corresponding v*

- *For each branch you've just drawn (for value v)*
  - *If S_v only contains examples in category c*
    Then put that category as a leaf node in the tree
  - *If S_v is empty*
    Then find the default category (which contains the most examples from S)
    - *Put this default category as a leaf node in the tree*
  - *Otherwise*
    Remove A from attributes which can be put into nodes
    Replace S with S_v
    Find new attribute A scoring best for Gain(S, A)
    Start again at part 2
- *Make sure you replace S with S_v*

Example

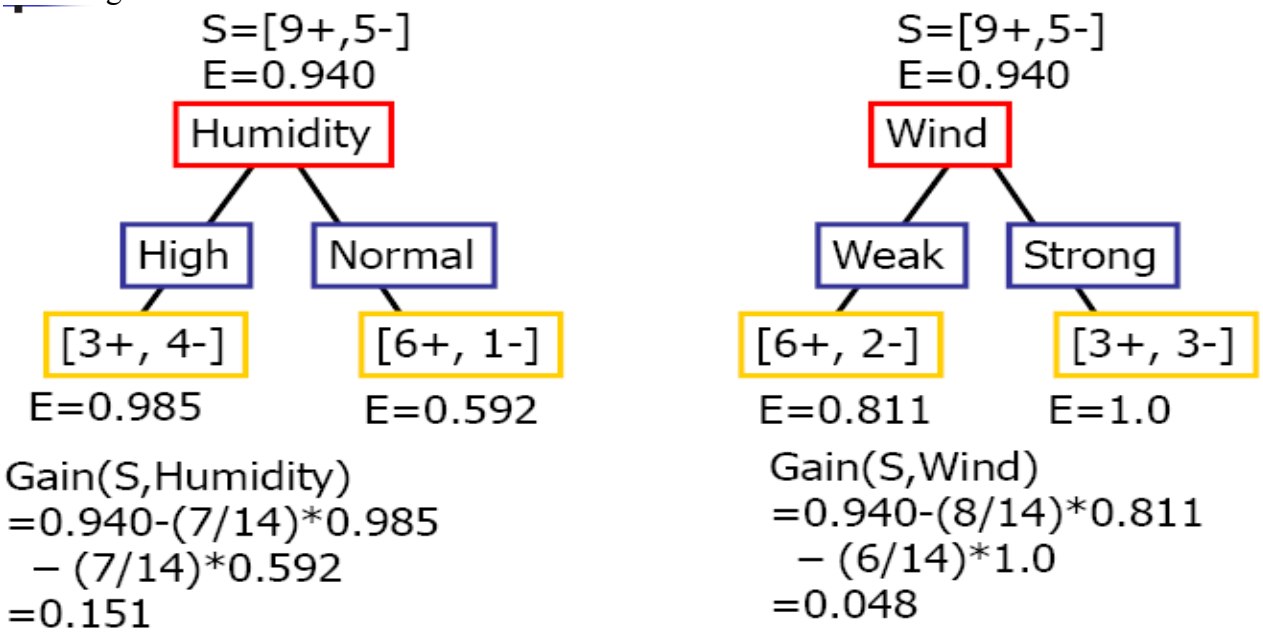
a) Consider the following training examples:

Day	Outlook	Temp.	Humidity	Wind	Play Tennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Weak	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cold	Normal	Weak	Yes
D10	Rain	Mild	Normal	Strong	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

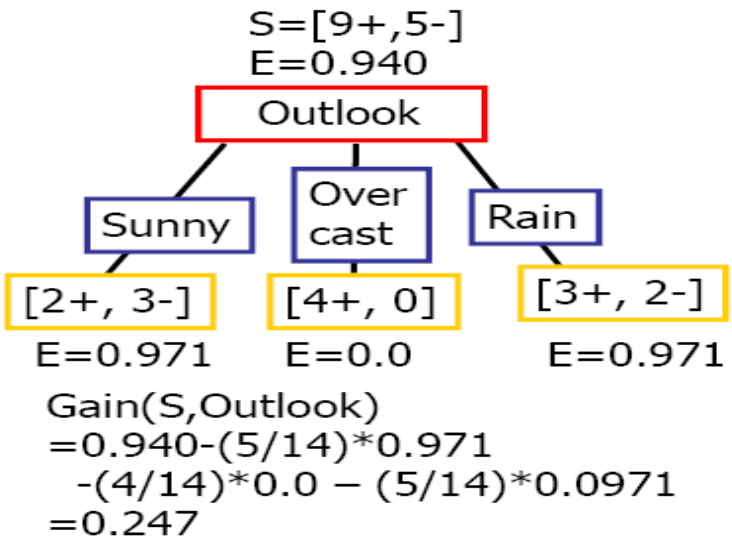
Use the ID3 algorithm to come up with a decision tree that models the above training examples

Solution:

Selecting the Next Attribute



Humidity provides greater information gain than Wind, w.r.t target classification.

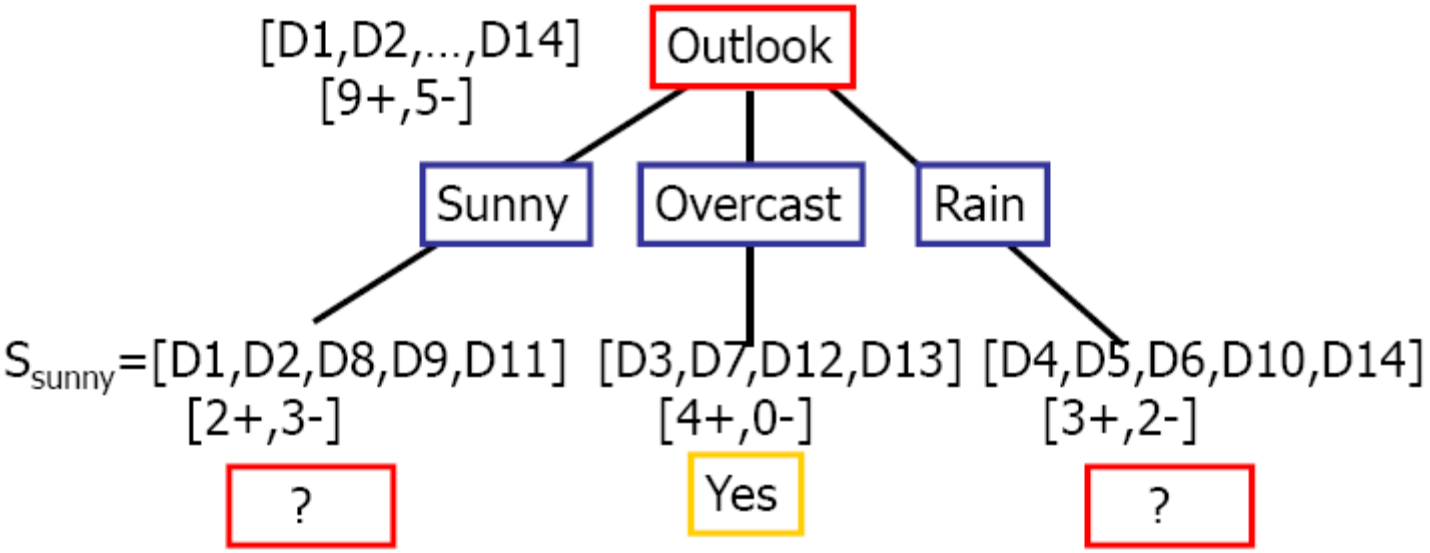


In summary, the following is information gain values for the 4 attributes :

- Gain(S,Outlook) =0.247
- Gain(S,Humidity) =0.151
- Gain(S,Wind) =0.048
- Gain(S,Temperature) =0.029

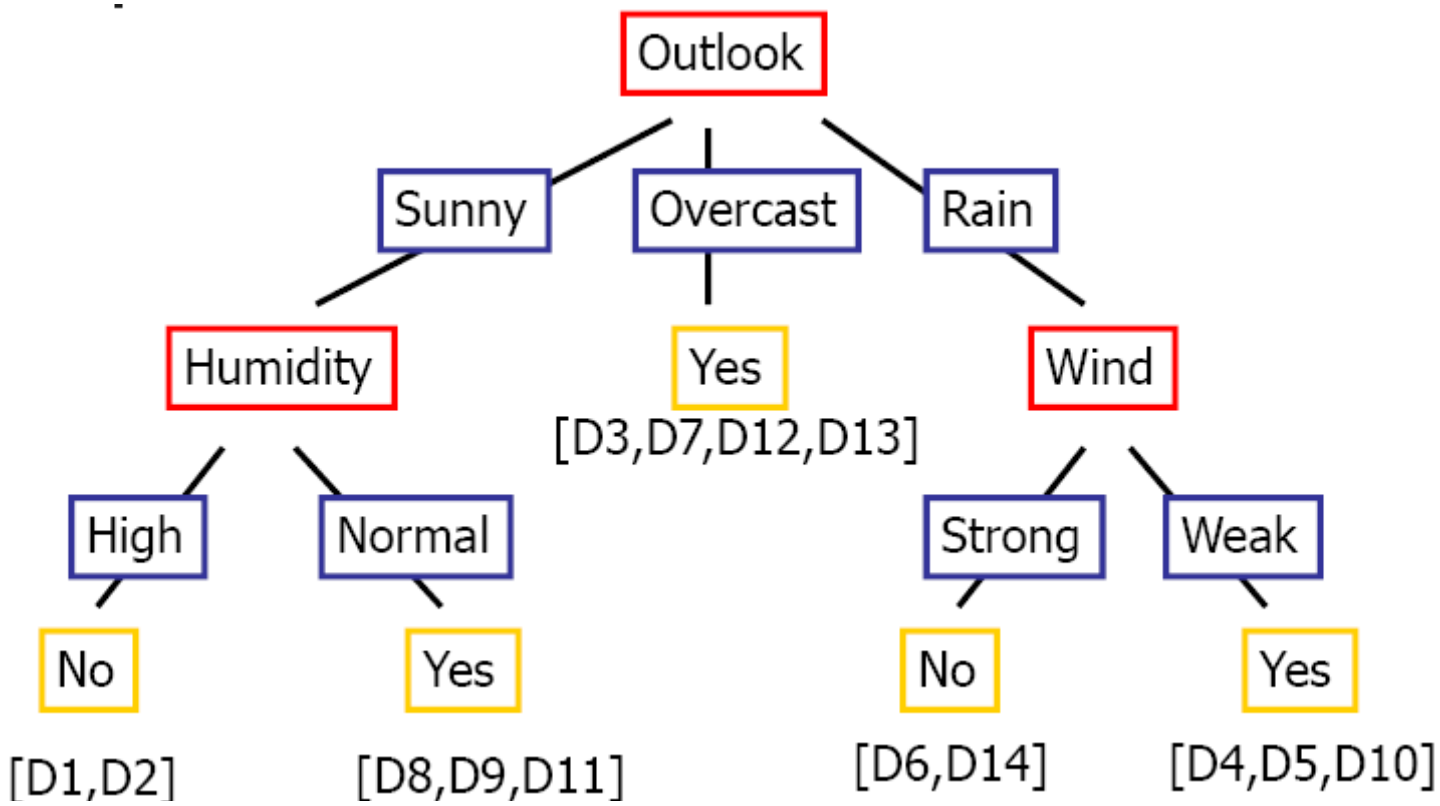
where S denotes the collection of training examples

From the above, you draw the following tree



Gain(S_{sunny} , Humidity)=0.970-(3/5)0.0 – 2/5(0.0) = 0.970
Gain(S_{sunny} , Temp.)=0.970-(2/5)0.0 –2/5(1.0)-(1/5)0.0 = 0.570
Gain(S_{sunny} , Wind)=0.970= -(2/5)1.0 – 3/5(0.918) = 0.019

-



ii). Artificial Neural Networks

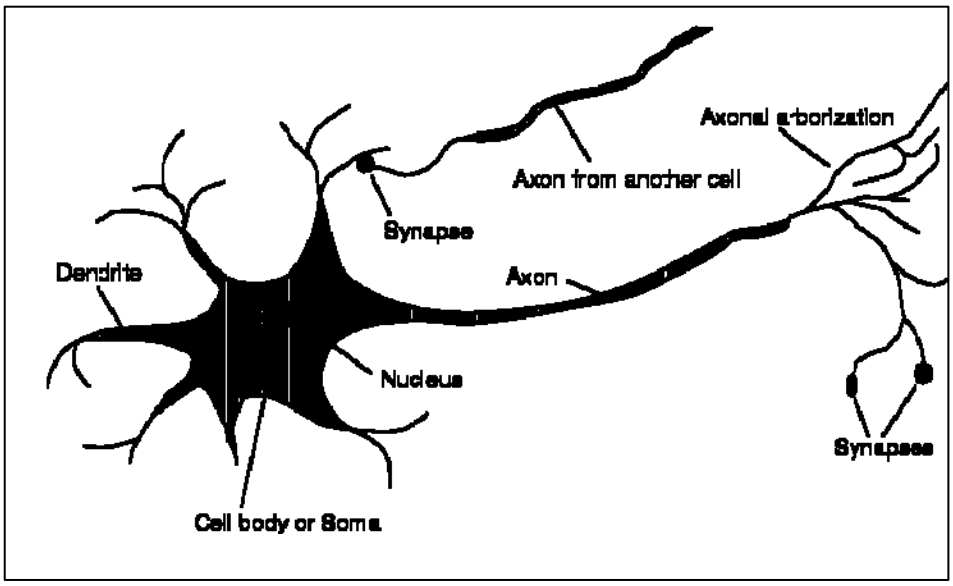
Neural networks model the human neuron. The neural network consists of a number of nodes that are connected using links. Each link has a numeric weight that is associated with it. The learning occurs by adjusting the weights so that the inputs correspond to the outputs. A neural network unit consists of a *linear input function* that computes the sum of weighted inputs. It also has a non linear component called activation function that transforms the final input values into a final activation value.

Neural networks have been used in pronunciation in which text streams are mapped to phonemes (basic sound elements; handwritten text recognition; driving (AlVIN(1993)-learnt how to steer a vehicle by observing the human driver.).

Neurobiology Constraints on Human Information Processing

- Number of neurons: 10^{11}
- Number of connections: 10^4 per neuron
- Neuron death rate: 10^5 per day
- Neuron birth rate: 0
- Connection birth rate: very slow
- Performance: about 10^2 msec, or about 100 sequential neuron firings for "many" tasks

The Structure of Neurons



A neuron has a cell body, a branching input structure (the dendrite) and a branching output structure (the axon)

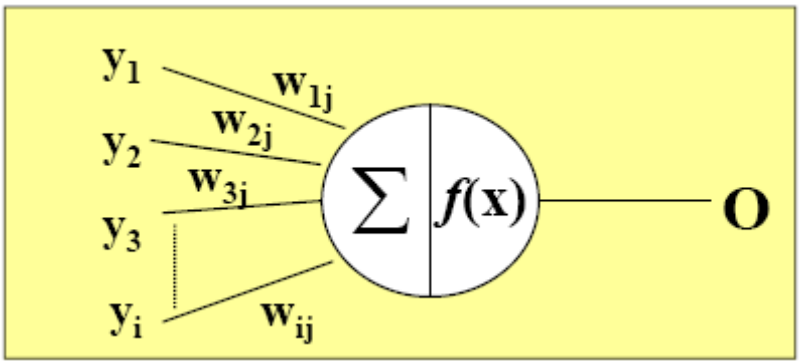
- Axons connect to dendrites via synapses.
- Electro-chemical signals are propagated from the dendritic input, through the cell body, and down the axon to other neurons
- A neuron only fires if its input signal exceeds a certain amount (the threshold) in a short time period.
- Synapses vary in strength
    - Good connections allowing a large signal
    - Slight connections allow only a weak signal.
    - Synapses can be either excitatory or inhibitory.

Why Neural Nets

- The autonomous local processing of each individual unit combines with similar simple behavior of many other units to produce "interesting," complex, global behavior
- Intelligent behavior is an "emergent" phenomenon

- Solving problems using a processing model that is similar to the brain may lead to solutions to complex information processing problems that would be difficult to achieve using traditional symbolic approaches in AI
- Associative memory access is directly represented. Hence pattern-directed retrieval and matching operations are promoted.
- Robust computation because knowledge is distributed and continuous rather than discrete or digital. Knowledge captured in a large number of fine-grained units and can match noisy and incomplete data
- Fault tolerant architecture because computations can be organized so as not to depend on a fixed set of units and connections

A Simple Model of a Neuron (Perceptron)



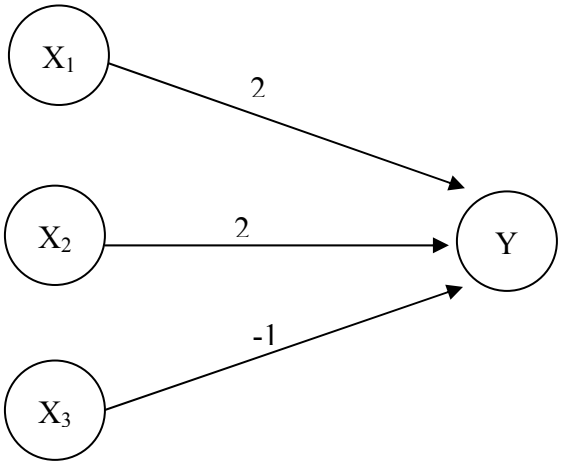
- Each neuron has a threshold value
- Each neuron has weighted inputs from other neurons
- The input signals form a weighted sum
- If the activation level exceeds the threshold, the neuron "fires"
- Each hidden or output neuron has weighted input connections from each of the units in the preceding layer.
- The unit performs a weighted sum of its inputs, and subtracts its threshold value, to give its activation level.
- Activation level is passed through a sigmoid activation function or any other function to determine output.

The First Artificial Neuron

McCulloch and Pitts (1943) produced the first neural network, which was based on their artificial neuron. Although this work was developed in the early forties, many of the principles can still be seen in the neural networks of today.

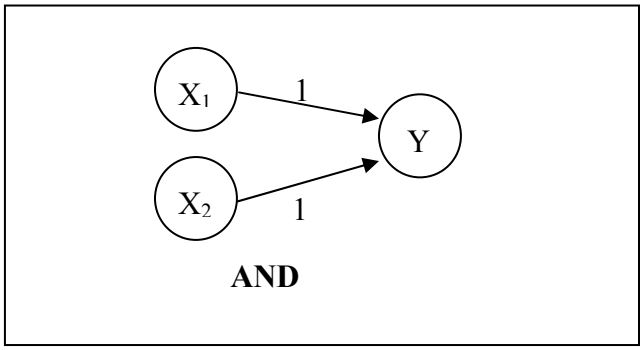
We can make the following statements about a McCulloch-Pitts network

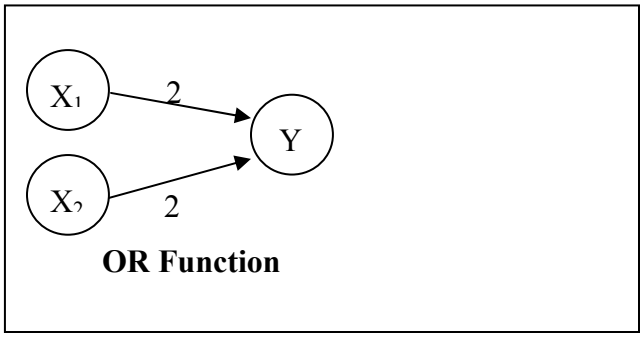
- The activation of a neuron is binary. That is, the neuron either fires (activation of one) or does not fire (activation of zero).
  For the network shown below the activation function for unit *Y* is
  $$f(y\_in) = 1, \text{ if } y\_in >= \theta \text{ else } 0$$
  where *y_in* is the total input signal received and θ is the threshold for *Y*.
- Neurons is a McCulloch-Pitts network are connected by directed, weighted paths.
- If the weight on a path is positive the path is excitatory, otherwise it is inhibitory.
- All excitatory connections into a particular neuron have the same weight, although different weighted connections can be input to different neurons.
- Each neuron has a fixed threshold. If the net input into the neuron is greater than the threshold, the neuron fires.
- The threshold is set such that any non-zero inhibitory input will prevent the neuron from firing.
- It takes one time step for a signal to pass over one connection.

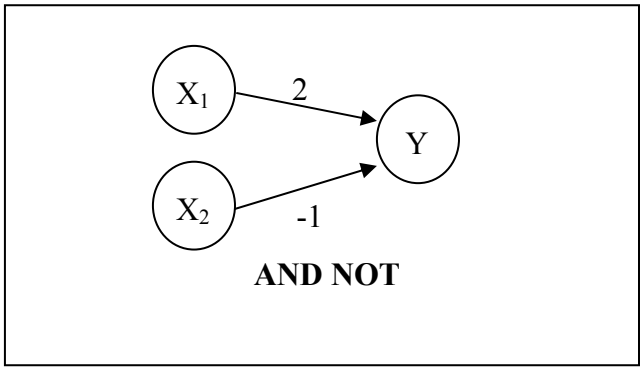


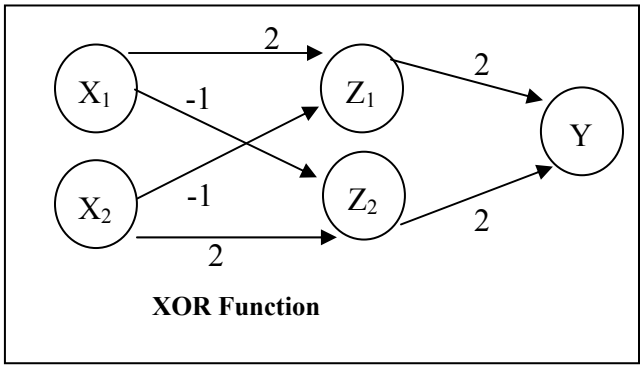
A sample McCulloch-Pitts network is shown above and some of the statements can be observed. In particular, note that the threshold for Y was equal 4 as this is the only value that allows it to fire, taking into account that a neuron cannot fire if it receives a nonzero inhibitory input.

Using the McCulloch-Pitts model we can model logic functions. Below we show and describe the architecture for four logic functions (the truth tables for each function is also shown)









AND		
X_1	X_2	Y
1	1	1
1	0	0
0	1	0
0	0	0

OR		
X_1	X_2	Y
1	1	1
1	0	1
0	1	1
0	0	0

AND NOT		
X_1	X_2	Y
1	1	0
1	0	1
0	1	0
0	0	0

XOR		
X_1	X_2	Y
1	1	0
1	0	1
0	1	1
0	0	0

AND Function

As both inputs (X_1 and X_2) are connected to the same neuron the connections must be the same, in this case 1. To model the AND function the threshold on Y is set to 2.

OR Function

This is almost identical to the AND function except the connections are set to 2 and the threshold on Y is also set to 2.

AND NOT Function

The AND NOT function is not symmetric in that an input of 1,0 is treated differently to an input of 0,1. As you can see from the truth table the only time true (value of one) is returned is when the first input is true and the second input is false. Again, the threshold on Y is set to 2 and if you apply each of the inputs to the AND NOT network you will find that we have modeled X_1 AND NOT X_2.

XOR Function

XOR can be modeled using AND NOT and OR;

$$X_1 \text{ XOR } X_2 = (X_1 \text{ AND NOT } X_2) \text{ OR } (X_2 \text{ AND NOT } X_1)$$

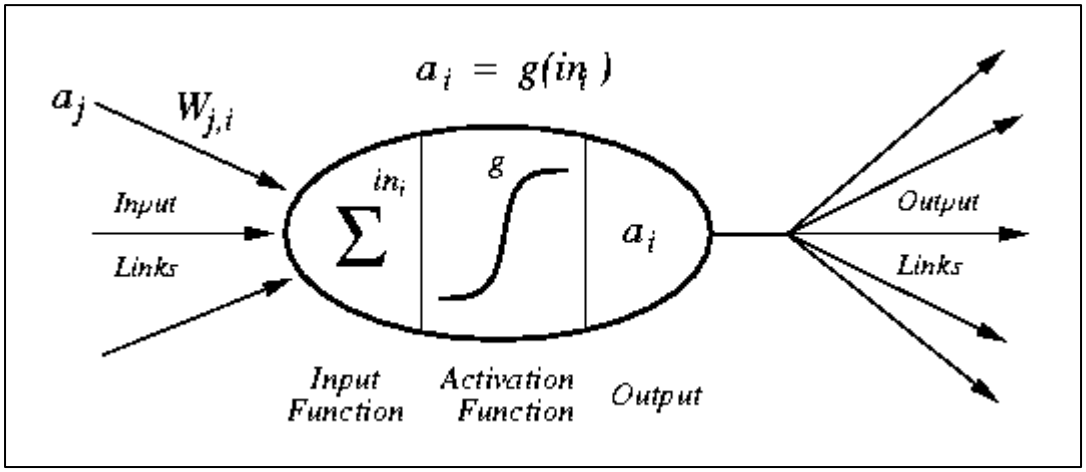
This explains the network shown above. The first layer performs the two AND NOT's and the second layer performs the OR. Both Z neurons and the Y neuron have a threshold of 2.

Modelling a Neuron

To model the brain we need to model a neuron. Each neuron performs a simple computation. It receives signals from its input links and it uses these values to compute the activation level (or output) for the neuron. This value is passed to other neurons via its output links.
The input value received of a neuron is calculated by summing the weighted input values from its input links. That is

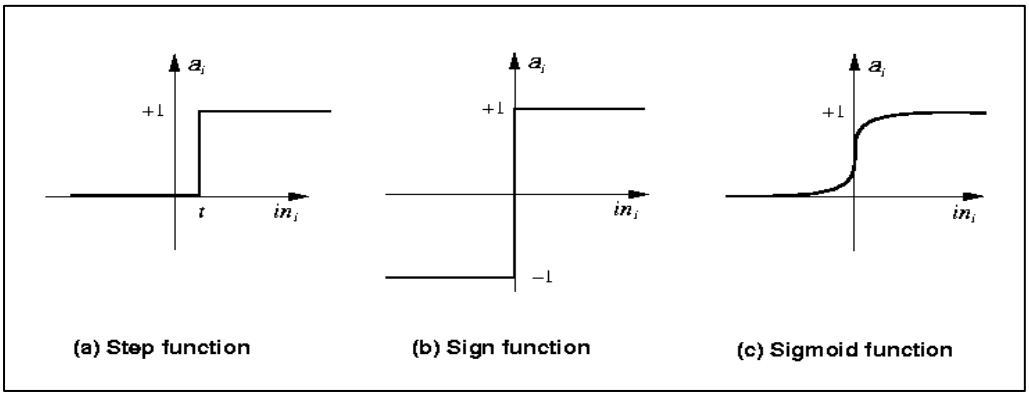
$$in_i = \sum_j W_{j,i} a_j$$

An activation function takes the neuron input value and produces a value which becomes the output value of the neuron. This value is passed to other neurons in the network.
This is summarised in this diagram and the notes below.



a_j : Activation value of unit j
$w_{j,i}$: Weight on the link from unit j to unit i
in_i : Weighted sum of inputs to unit i
a_i : Activation value of unit i (also known as the output value)
g : Activation function

Some common activation functions are shown below.



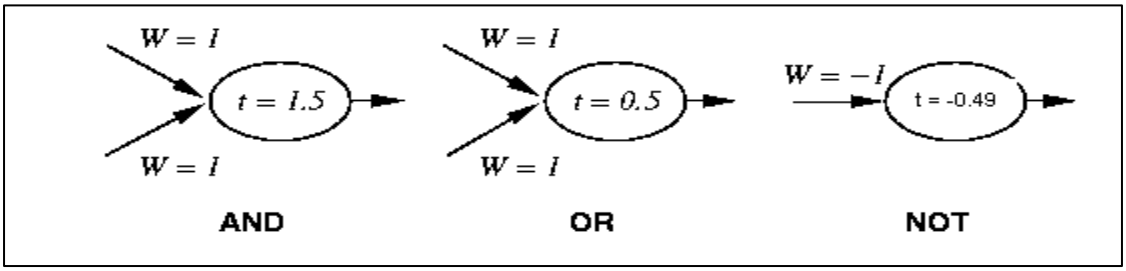
These functions can be defined as follows.
$Step_t(x)$ = 1 if x >= t, else 0
Sign(x) = +1 if x >= 0, else –1
Sigmoid(x) = $1/(1+e^{-x})$

Some Simple Networks
We can use what we have learnt above to demonstrate a simple neural network which acts as a logic gate. The diagram below is modelling the following truth tables

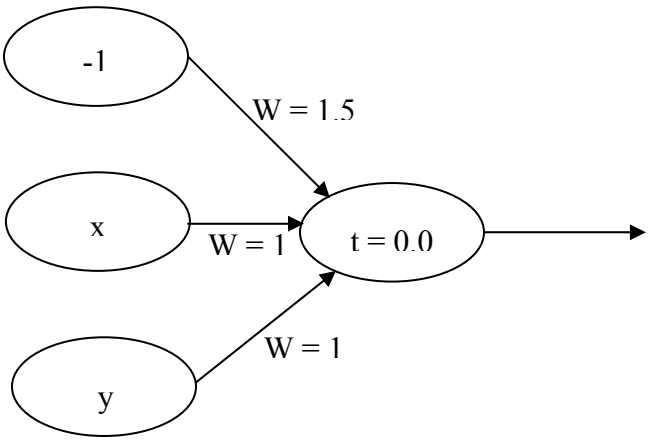
	AND			
Input 1	0	0	1	1
Input 2	0	1	0	1
Output	0	0	0	1

OR			
0	0	1	1
0	1	0	1
0	1	1	1

NOT	
0	1
1	0



In these networks we are using the step activation function. Notice that each neuron has a different threshold. From a computational viewpoint it would be easier if all the neurons had the same threshold value and the actual threshold was somehow modelled by the weights.
In fact, it is possible to do exactly this. Consider this network

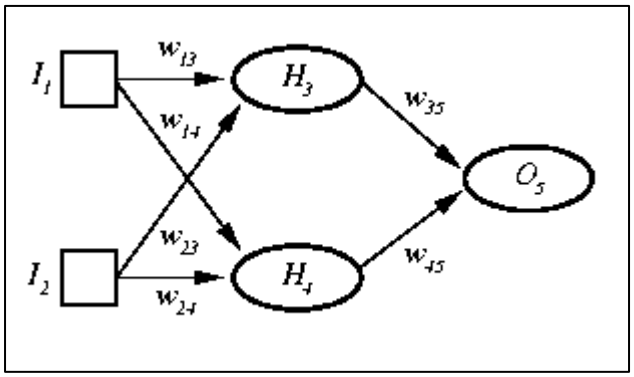


It is the same as the AND network in the diagram above except that there is an extra input neuron whose activation is always set to –1. The threshold of the neuron is represented by the weight that links this extra neuron to the output neuron. This means the threshold of the neuron can be set to zero.

This advantage of this method is that we can always set the threshold for every neuron to zero and from a computational point of view, when "training" the network, we only have to update weights and not both thresholds and weights.

Types of Network

The simple networks we have considered above only have input neurons and output neurons. It is considered a one layer network (the input neurons are not normally considered to form a layer as they are just a means of getting data into the network).
Also, in the networks we have considered, the data only travels in one direction (from the input neurons to the output neurons). In this respect it is known as a feed-forward network.
Therefore, we have been looking at one-layer, feed-forward networks.
There are many other types of network. An example of a two-layer, feed-forward network is shown below.



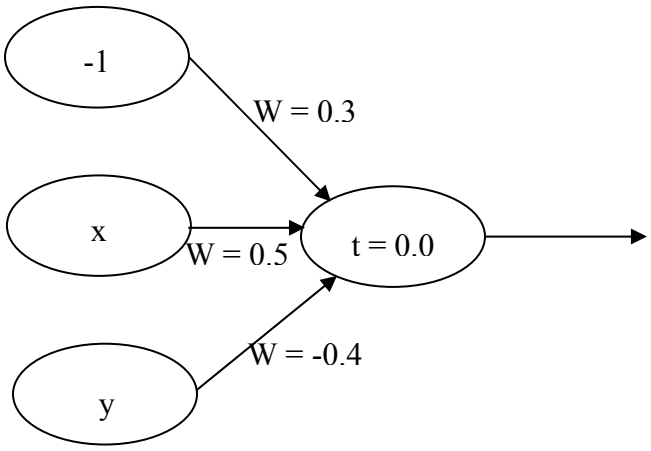
The Perceptron

The name perceptron is now used as a synonym for single-layer, feed-forward networks. They were first studied in the 1950's and although other network architectures were known about the perceptron was the only network that was known to be capable of learning and thus most of the research at that time concentrated on perceptrons.

Learning algorithm

With a function such as AND (with only two inputs) we can easily decide what weights to use to give us the required output from the neuron. But with more complex functions (i.e. those with more than two inputs it may not be so easy to decide on the correct weights).
Therefore, we would like our neural network to "learn" so that it can come up with its own set of weights.
We will consider this aspect of neural networks for a simple function (i.e. one with two inputs). Consider this truth table (AND) and the neuron that we ***hope*** will represent it.

	AND			
Input 1	0	0	1	1
Input 2	0	1	0	1
Output	0	0	0	1



In fact, all we have done is set the weights to random values between –0.5 and 0.5

By applying the activation function for each of the four possible inputs to this neuron it actually gives us the following truth table

	???			
Input 1	0	0	1	1
Input 2	0	1	0	1
Output	0	0	1	0

I_1	I_2	I_3	Summation	Output
-1	0	0	(-1*0.3) + (0*0.5) + (0*-0.4) = -0.3	0
-1	0	1	(-1*0.3) + (0*0.5) + (1*-0.4) = -0.7	0
-1	1	0	(-1*0.3) + (1*0.5) + (0*-0.4) = 0.2	1
-1	1	1	(-1*0.3) + (1*0.5) + (1*-0.4) = -0.2	0

The network, obviously, does not represent the AND function so we need to adjust the weights so that it "learns" the function correctly.

The algorithm to do this follows, but first some terminology.

Epoch : An epoch is the presentation of the entire training set to the neural network. In the case of the AND function an epoch consists of four sets of inputs being presented to the network (i.e. [0,0], [0,1], [1,0], [1,1]).

Training Value, T : When we are training a network we not only present it with the input but also with a value that we require the network to produce. For example, if we present the network with [1,1] for the AND function the training value will be 1.

Error, Err : The error value is the amount by which the value output by the network differs from the training value. For example, if we required the network to output 0 and it output a 1, then Err = -1.

Output from Neuron, O : The output value from the neuron

I_j : Inputs being presented to the neuron

W_j : Weight from input neuron (I_j) to the output neuron

LR : The learning rate. This dictates how quickly the network converges. It is set by a matter of experimentation. It is typically 0.1.

The Perceptron Training Algorithm

While epoch produces an error
    Present network with next inputs from epoch
    Err = T – O
    If Err <> 0 then
        Note : If the error is positive we need to increase O. If the error is negative we need to decrease O. Each input contributes WjIj to the total input so if Ij is positive, an increase in Wj will increase O. If Ij is negative an increase in Wj will decrease O).
        This can be achieved with the following
        $W_j = W_j + LR * I_j * Err$
        Note : This is often called the delta learning rule.
    End If
End While

Example

Let initial weight values are 0.3, 0.5, and -0.4 (taken from the above example) and we are trying to learn the AND function. If we present the network with the first training pair ([0,0]), from the first epoch, nothing will happen to the weights (due to multiplying by zero).
The next training pair ([0,1]) will result in the network producing zero (by virtue of the step function). As zero is the required output there is no error so training continues.
The next training pair ([1,0]) produces an output of one. The required output is 0. Therefore the error is –1. This means we have to adjust the weights.

This is done as follows (assuming LR = 0.1)
W_0 = 0.3 + 0.1 * -1 * -1 = 0.4
W_1 = 0.5 + 0.1 * 1 * -1 = 0.4
W_2 = -0.4 + 0.1 * 0 * -1 = -.04

Therefore, the new weights are 0.4, 0.4, -0.4.
Finally we apply the input [1,1] to the network. This also produces an error and the new weight values will be 0.3, 0.5 and –0.3. As this presentation of the epoch produced an error (two in fact) we need to continue the training and present the network with another epoch.
Training continues until an epoch is presented that does not produce an error.
Note: If the network is trained until all errors from an entire epoch are eliminated. such weights produce produce outputs which are linearly

Learning in Neural Networks
- Learn values of weights from I/O pairs
- Start with random weights
- Load training example's input
- Observe computed output
- Modify weights to reduce difference
- Iterate over all training examples
- Terminate when weights stop changing OR when error is very small

Linear Separability
A set of (2D) patterns (x_1, x_2) of two classes is linearly separable if there exists a line on the (x_1, x_2) plane
- $w_0 + w_1 x_1 + w_2 x_2 = 0$
- Separates all patterns of one class from the other class

A perceptron can be built with

- 3 input $x_0 = 1$, x_1, x_2 with weights w_0, w_1, w_2

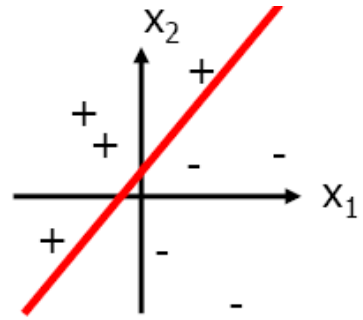
n dimensional patterns $(x_1, \ldots, x_n)$

- Hyperplane $w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = 0$ dividing the space into two regions

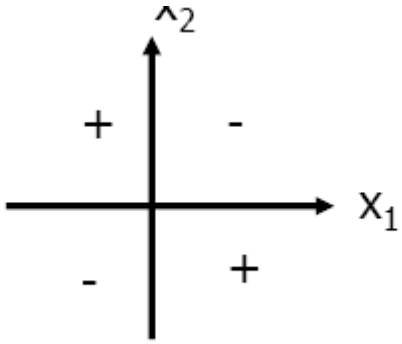
Can we get the weights from a set of sample patterns?

- If the problem is linearly separable, then YES (by perceptron learning)

Note: Perceptrons can only learn functions that are linearly separable



Linearly separable

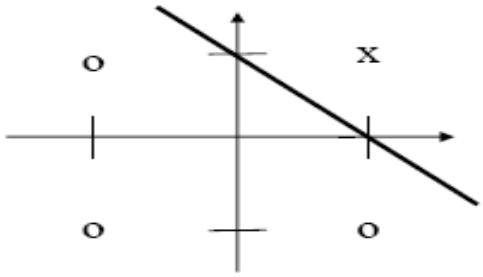


Non-Linearly separable

- Examples of linearly separable classes

Logical **AND** function

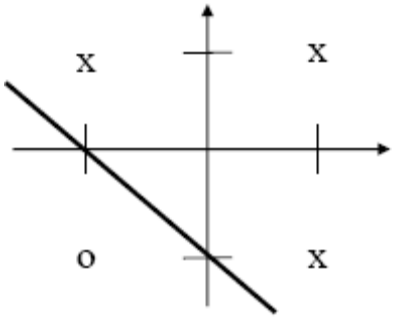
patterns (bipolar)			decision boundary
x1	x2	output	w1 = 1
-1	-1	-1	w2 = 1
-1	1	-1	w0 = -1
1	-1	-1	
1	1	1	-1 + x1 + x2 = 0



x: class I (output = 1)
o: class II (output = -1)

- Logical **OR** function

patterns (bipolar)			decision boundary
x1	x2	output	w1 = 1
-1	-1	-1	w2 = 1
-1	1	1	w0 = 1
1	-1	1	
1	1	1	1 + x1 + x2 = 0



x: class I (output = 1)
o: class II (output = -1)