# UNIT IV

**Planning and Learning:** Planning with state space search - conditional planning-continuous planning - Multi-Agent planning. Forms of learning - inductive learning - Reinforcement Learning - learning decision trees - Neural Net learning and Genetic learning

# What is Planning in AI?Explain the Planning done by an agent?

**Planning problem**

– Find a **sequence of actions** that achieves a given **goal** when executed from a given **initial world state**. That is, given

–a set of operator descriptions (defining the possible primitive actions by the agent),

–an initial state description, and

–a goal state description or predicate, compute a plan, which is

–a sequence of operator instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.

* Goals are usually specified as a conjunction of goals to be achieved

### An Agent Architecture



 Planning vs. problem solving

* Planning and problem solving methods can often solve the same sorts of problems
* Planning is more powerful because of the representations and methods used
* States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)
* Search often proceeds through *plan space* rather than *state space* (though there are also state- space planners)
* Subgoals can be planned independently, reducing the complexity of the planning problem Typical assumptions
* Atomic time: Each action is indivisible
* No concurrent actions are allowed (though actions do not need to be ordered with respect to each other in the plan)
* Deterministic actions: The result of actions are completely determined—there is no uncertainty in their effects
* Agent is the sole cause of change in the world
* Agent is omniscient: Has complete knowledge of the state of the world
* Closed World Assumption: everything known to be true in the world is included in the state description. Anything not listed is false.

Blocks world

The **blocks world** is a micro-world that consists of a table, a set of blocks and a robot hand. Some domain constraints:

–Only one block can be on another block

–Any number of blocks can be on the table

–The hand can only hold one block Typical representation:

ontable(a) ontable(c) on(b,a) handempty clear(b) clear(c)

**General Problem Solver**

* The General Problem Solver (GPS) system was an early planner (Newell, Shaw, and Simon)
* GPS generated actions that reduced the difference between some state and a goal state
* GPS used Means-Ends Analysis

–Compare what is given or known with what is desired and select a reasonable thing to do next

–Use a table of differences to identify procedures to reduce types of differences

* GPS was a state space planner: it operated in the domain of state space problems specified by an initial state, some goal states, and a set of operations

Situation calculus planning

* Intuition: Represent the planning problem using first-order logic
	+ Situation calculus lets us reason about changes in the world
	+ Use theorem proving to ―prove‖ that a particular sequence of actions, when applied to the situation characterizing the world state, will lead to a desired result

Situation calculus

* **Initial state**: a logical sentence about (situation) S0

At(Home, S0) ^ ~Have(Milk, S0) ^ ~ Have(Bananas, S0) ^ ~Have(Drill, S0)

## Goal state:

(s) At(Home,s) ^ Have(Milk,s) ^ Have(Bananas,s) ^ Have(Drill,s)

* **Operators** are descriptions of how the world changes as a result of the agent’s actions:

(a,s) Have(Milk,Result(a,s)) <=> ((a=Buy(Milk) ^ At(Grocery,s))  (Have(Milk, s) ^ a~=Drop(Milk)))

* Result(a,s) names the situation resulting from executing action a in situation s.
* Action sequences are also useful: Result'(l,s) is the result of executing the list of actions (l) starting in s:

(s) Result'([],s) = s

(a,p,s) Result'([a|p]s) = Result'(p,Result(a,s))

 Basic representations for planning

* Classic approach first used in the **STRIPS** planner circa 1970
* States represented as a conjunction of ground literals

–at(Home) ^ ~have(Milk) ^ ~have(bananas) ...

* Goals are conjunctions of literals, but may have variables which are assumed to be existentially quantified

–at(?x) ^ have(Milk) ^ have(bananas) ...

* Do not need to fully specify state

–Non-specified either don’t-care or assumed false

–Represent many cases in small storage

–Often only represent changes in state rather than entire situation

* Unlike theorem prover, not seeking whether the goal is true, but is there a sequence of actions to attain it

Operator/action representation

* Operators contain three components:

## –Action description

–**Precondition** - conjunction of positive literals

–**Effect** - conjunction of positive or negative literals which describe how situation changes when operator is applied

Example:

Op[Action: Go(there),

Precond: At(here) ^ Path(here,there), Effect: At(there) ^ ~At(here)]

* All variables are universally quantified
* Situation variables are implicit

–preconditions must be true in the state immediately before operator is applied; effects are true immediately after

Blocks world operators

Here are the classic basic operations for the blocks world:

–stack(X,Y): put block X on block Y

–unstack(X,Y): remove block X from block Y

–pickup(X): pickup block X

–putdown(X): put block X on the table Each will be represented by

–a list of preconditions

–a list of new facts to be added (add-effects)

–a list of facts to be removed (delete-effects)

–optionally, a set of (simple) variable constraints For example:

preconditions(stack(X,Y), [holding(X),clear(Y)]) deletes(stack(X,Y), [holding(X),clear(Y)]).

adds(stack(X,Y), [handempty,on(X,Y),clear(X)]) constraints(stack(X,Y), [X\==Y,Y\==table,X\==table])

STRIPS planning

* STRIPS maintain two additional data structures:

–**State List** - all currently true predicates.

–**Goal Stack** - a push down stack of goals to be solved, with current goal on top of stack.

* If current goal is not satisfied by present state, examine add lists of operators, and push operator and preconditions list on stack. (Sub goals)
* When a current goal is satisfied, POP it from stack.
* When an operator is on top stack, record the application of that operator on the plan sequence and use the operator’s add and delete lists to update the current state.

Typical BW planning problem

Initial state:

clear(a) clear(b) clear(c) ontable(a) ontable(b) ontable(c) handempty

Goal interaction

* Simple planning algorithms assume that the goals to be achieved are independent

–Each can be solved separately and then the solutions concatenated

* This planning problem, called the ―Sussman Anomaly,‖ is the classic example of the goal interaction problem:
	+ Solving on(A,B) first (by doing unstack(C,A), stack(A,B) will be undone when solving the second goal on(B,C) (by doing unstack(A,B), stack(B,C)).

–Solving on(B,C) first will be undone when solving on(A,B)

* Classic STRIPS could not handle this, although minor modifications can get it to do simple cases

**State-space planning**

* We initially have a space of situations (where you are, what you have, etc.)
* The plan is a solution found by ―searching‖ through the situations to get to the goal
* A **progression planner** searches forward from initial state to goal state
* A **regression planner** searches backward from the goal

–This works if operators have enough information to go both ways

–Ideally this leads to reduced branching –you are only considering things that are relevant to the goal

Plan-space planning

* An alternative is to **search through the space of *plans***, rather than situations.
* Start from a **partial plan** which is expanded and refined until a complete plan that solves the problem is generated.
* **Refinement operators** add constraints to the partial plan and modification operators for other changes.
* We can still use STRIPS-style operators:

Op(ACTION: RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn) Op(ACTION: RightSock, EFFECT: RightSockOn)

Op(ACTION: LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn) Op(ACTION: LeftSock, EFFECT: leftSockOn)

could result in a partial plan of [RightShoe, LeftShoe]

# What is learning and its representation in AI explain?

Learning is an important area in AI, perhaps more so than planning.

* + Problems are hard -- harder than planning.
	+ Recognised Solutions are not as common as planning.
	+ A goal of AI is to enable [computers](http://www.ntsearch.com/search.php?q=computers&v=56) that can be taught rather than programmed.

[*Learning*](http://www.ntsearch.com/search.php?q=Learning&v=56) is a an area of AI that focusses on processes of self-improvement.

Information processes that improve their performance or enlarge their knowledge bases are said to *learn*.

*Why is it hard?*

* + Intelligence implies that an organism or machine must be able to adapt to new situations.
	+ It must be able to learn to do new things.
	+ This requires knowledge acquisition, inference, updating/refinement of knowledge base, acquisition of heuristics, applying faster searches, *etc.*

How can we learn?

Many approaches have been taken to attempt to provide a machine with [learning](http://www.ntsearch.com/search.php?q=learning&v=56) capabilities. This is because [learning](http://www.ntsearch.com/search.php?q=learning&v=56) tasks [cover](http://www.ntsearch.com/search.php?q=cover&v=56) a wide range of phenomena.

Listed below are a few examples of how one may learn. We will look at these in detail shortly

## Skill refinement

-- one can learn by practicing, *e.g playing the piano*.

## Knowledge acquisition

-- one can learn by experience and by storing the experience in a knowledge base. One basic example of this type is rote [learning.](http://www.ntsearch.com/search.php?q=learning&v=56)

## Taking advice

-- Similar to rote [learning](http://www.ntsearch.com/search.php?q=learning&v=56) although the knowledge that is input may need to be transformed (or *operationalised*) in order to be used effectively.

## Problem Solving

-- if we solve a problem one may learn from this experience. The next [time](http://www.ntsearch.com/search.php?q=time&v=56) we see a similar problem we can solve it more efficiently. This does not usually involve gathering new knowledge but may involve reorganisation of data or remembering how to achieve to solution.

## Induction

-- One can learn from *examples*. Humans often classify things in the world without knowing explicit rules. Usually involves a teacher or trainer to aid the classification.

## Discovery

* + - Here one learns knowledge without the aid of a teacher.

## Analogy

* + - If a system can recognise similarities in information already stored then it may be able to transfer some knowledge to improve to solution of the task in hand.

## General model:

* **Learning is essential for unknown environments, i.e., when designer lacks omniscience. Learning is useful as a system construction method, i.e., expose the agent to reality rather than trying to write it down.**

## Learning modifies the agent's decision mechanisms to improve performance



•

Learning element

## Design of a learning element is affected by

* **Which components of the performance element are to be learned**

## What feedback is available to learn these components

* **What representation is used for the components**

## Type of feedback:

* **Supervised learning: correct answers for each example**

## Unsupervised learning: correct answers not given

* **Reinforcement learning: occasional rewards 3.Explain Inductive Learning**

Inductive learning is a kind of [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) in which, given a set of examples an agent tries to estimate or create an evaluation function. Most inductive learning is supervised [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning), in which examples provided with classifications. (The alternative is *clustering*.) More formally, an

*example* is a pair *(x, f(x))*, where *x* is the input and *f(x)* is the output of the function applied to *x*. The task of *pure inductive inference* (or *induction*) is, given a set of examples of *f*, to find a hypothesis *h* that approximates *f*.

* Given an example, A pair <x, f(x)> where x is the input and f(x) is the result
* Generate a hypothesis, A function h(x) that approximates f(x)
* That will generalize well, Correctly predict values for unseen samples

## How do we do this?

\_Must determine a hypothesis space...

* The set of all hypotheses we are willing to consider
* \_e.g. All functions of degree less than 10

That is realizable, Contains the true function

## Inductive learning method

Simplest form: learn a function from examples (tabula rasa) f is the target function

An example is a pair x, f(x), e.g., 

Problem: find a(n) hypothesis h such that h≈f

given a training set of examples

(This is a highly simpli\_ed model of real learning:

{ Ignores prior knowledge

{ Assumes a deterministic, observable \environment"

{ Assumes examples are given

{ Assumes that the agent wants to learn f|why?)

## Construct/adjust *h* to agree with *f* on training set (*h* is consistent if it agrees with *f*

**on all examples)**

## E.g., curve fitting:



•

## Construct/adjust *h* to agree with *f* on training set (*h* is consistent if it agrees with *f*

**on all examples)**

## E.g., curve fitting:



* **Construct/adjust *h* to agree with *f* on training set (*h* is consistent if it agrees with *f***

## on all examples)

* **E.g., curve fitting:**



## Construct/adjust *h* to agree with *f* on training set (*h* is consistent if it agrees with *f*

**on all examples)**

## E.g., curve fitting:

The most likely hypothesis is the simplest one consistent with the data." Since there can be noise in the measurements, in practice we need to make a tradeoff between simplicity of the hypothesis and how well it fits the data.

1. **Explain Reinforcement** [**Learning**](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning)

As opposed to *supervised learning*, reinforcement [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) takes place in an environment where the agent cannot directly compare the results of its action to a desired result. Instead, it is given some *reward* or *punishment* that relates to its actions. It may win or lose a game, or be told it has made a good move or a poor one. The job of reinforcement [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) is to find a successful function using these rewards.

The reason reinforcement [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) is harder than supervised learning is that the agent is never told what the right action is, only whether it is doing well or poorly, and in some cases (such as chess) it may only receive feedback after a long string of actions.

There are two basic kinds of information an agent can try to learn.

* + **utility function** -- The agent learns the utility of being in various states, and chooses actions to maximize the expected utility of their outcomes. This requires the agent keep a model of the environment.
	+ **action-value** -- The agent learns an action-value function giving the expected utility of performing an action in a given state. This is called *Q-learning*. This is the *model-free* approach.

Passive [Learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) in a Known Environment

Assuming an environment consisting of a set of states, some terminal and some nonterminal, and a model that specifies the probabilities of transition from state to state, an agent learns passively by observing a set of *training sequences*, which consist of a set of state transitions followed by a reward.

The goal is to use the reward information to learn the expected utility of each of the nonterminal states. **An important simplifying assumption is that the utility of a sequence is the sum of the rewards accumulated in the states of the sequence**. That is, the utility function is *additive*.

A passive [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) agent keeps an estimate *U* of the utility of each state, a table *N* of how many times each state was seen, and a table *M* of transition probabilities. There are a variety of ways the agent can update its table *U*.

### Naive Updating

One simple updating method is the *least mean squares* (LMS) approach [Widrow and Hoff, 1960]. It assumes that the observed reward-to-go of a state in a sequence provides direct evidence of the actual [reward-to-go.](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#reward-to-go) The approach is simply to keep the utility as a running average of the rewards based upon the number of times the state has been seen. This approach minimizes the mean square error with respect to the observed data.

This approach converges very slowly, because it ignores the fact that **the actual utility of a state is the probability-weighted average of its successors' utilities, plus its own reward**. LMS disregards these probabilities.

### Adaptive Dynamic Programming

If the transition probabilities and the rewards of the states are known (which will usually happen after a reasonably small set of training examples), then the actual utilities can be computed directly as

U(i) = R(i) + SUMj MijU(j)

where *U(i)* is the utility of state *i*, *R* is its reward, and *Mij* is the probability of transition from state *i* to state *j*. This is identical to a single *value determination* in the policy iteration algorithm for Markov decision processes. *Adaptive dynamic programming* is any kind of reinforcement learning method that works by solving the utility equations using a dynamic programming algorithm. It is exact, but of course highly inefficient in large state spaces.

### Temporal Difference [Learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning)

[Richard Sutton] Temporal difference [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) uses the difference in utility values between successive states to adjust them from one epoch to another. The key idea is to use the observed transitions to adjust the values of the observed states so that they agree with the ADP constraint equations. Practically, this means updating the utility of state *i* so that it agrees better with its successor *j*. This is done with the *temporal-difference* (TD) equation:

U(i) <- U(i) + a(R(i) + U(j) - U(i))

where *a* is a *learning rate* parameter.

Temporal difference [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) is a way of approximating the ADP constraint equations without solving them for all possible states. The idea generally is to define conditions that hold over local transitions when the utility estimates are correct, and then create update rules that nudge the estimates toward this equation.

This approach will cause *U(i)* to converge to the correct value if the [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) rate parameter decreases with the number of times a state has been visited [Dayan, 1992]. In general, as the number of training sequences tends to infinity, TD will converge on the same utilities as ADP.

Passive [Learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) in an Unknown Environment

Since neither temporal difference [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) nor LMS actually use the model *M* of state transition probabilities, they will operate unchanged in an unknown environment. The ADP approach, however, updates its estimated model of an unknown environment after each step, and this model is used to revise the utility estimates.

Any method for [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) stochastic functions can be used to learn the environment model; in particular, in a simple environment the transition probability *Mij* is just the percentage of times state *i* has transititoned to *j*.

The basic difference between TD and ADP is that TD adjusts a state to agree with the observed successor, while ADP makes a state agree with all successors that might occur, weighted by their probabilities. More importantly, ADP's adjustments may need to be propagated across all of the utility equations, while TD's affect only the current equation. TD is essentially a crude first approximation to ADP.

A middle-ground can be found by bounding or ordering the number of adjustments made in ADP, beyond the simple one made in TD. The *prioritized-sweeping* heuristic prefers only to make adjustments to states whose *likely* successors have just undergone *large* adjustments in their utility estimates. Such approximate ADP systems can be very nearly as efficient as ADP in terms of convergence, but operate much more quickly.

Active [Learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) in an Unknown Environment

The difference between active and passive agents is that passive agents learn a fixed policy, while the active agent must decide what action to take and how it will affect its rewards. To represent an active agent, the environment model *M* is extended to give the probability of a transition from a state *i* to a state *j*, *given an action a*. Utility is modified to be the reward of the state plus the maximum utility expected depending upon the agent's action:

U(i) = R(i) + maxa x SUMj MaijU(j)

An ADP agent is extended to learn transition probabilities given actions; this is simply another dimension in its transition table. A TD agent must similarly be extended to have a model of the environment.

Learning Action-Value Functions

An action-value function assigns an expected utility to the result of performing a given action in a given state. If *Q(a, i)* is the value of doing action *a* in state *i*, then

U(i) = maxa Q(a, i)

The equations for [Q-learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#Q-learning) are similar to those for state-based learning agents. The difference is that [Q-learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#Q-learning) agents do not need models of the world. The equilibrium equation, which can be used directly (as with ADP agents) is

Q(a, i) = R(i) + SUMj Maij maxa' Q(a', j)

The temporal difference version does not require that a model be learned; its update equation is Q(a, i) <- Q(a, i) + a(R(i) + maxa' Q(a', j) - Q(a, i))

Applications of Reinforcement [Learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning)

The first significant reinforcement [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) system was used in Arthur Samuel's checker-playing program. It used a weighted linear function to evaluate positions, though it did not use observed rewards in its [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) process.

TD-gammon [Tesauro, 1992] has an evaluation function represented by a fully-connected [neural](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#neural%20network) [network](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#neural%20network) with one hidden layer of 80 nodes; with the inclusion of some precomputed board features in its input, it was able to reach world-class play after about 300,000 training games.

A case of reinforcement [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) in robotics is the famous *cart-pole* balancing problem. The problem is to control the position of the cart (along a single axis) so as to keep a pole balanced on top of it upright, while staying within the limits of the track length. Actions are usually to jerk left or right, the so-called *bang-bang control* approach. The first work on this problem was the

BOXES system [Michie and Chambers, 1968], in which state space was partitioned into boxes, and reinforcement propogated into the boxes. More recent simulated work using [neural networks](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#neural%20network) [Furuta et al., 1984] simulated the *triple-inverted-pendulum* problem, in which three poles balance one atop another on a cart

1. **Discuss Neural Net learning and Genetic learning**

Neural Networks

More specifically, a [neural network](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#neural%20network) consists of a set of *nodes* (or *units*), *links* that connect one node to another, and *weights* associated with each link. Some nodes receive inputs via links; others directly from the environment, and some nodes send outputs out of the network. [Learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) usually occurs by adjusting the weights on the links.

Each unit has a set of weighted inputs, an *activation level*, and a way to compute its activation level at the next time step. This is done by applying an *activation function* to the weighted sum of the node's inputs. Generally, the weighted sum (also called the *input function*) is a strictly linear sum, while the activation function may be nonlinear. If the value of the activation function is above a *threshold*, the node "fires."

Generally, all nodes share the same activation function and threshold value, and only the topology and weights change.



* g is a non-linear function which takes as input a weighted sum of the input link signals (as well as an intrinsic bias weight) and outputs a certain signal strength.
* g is commonly a threshold function or sigmoid function. Network Structures

The two fundamental types of network structure are *feed-forward* and *recurrent*. A feed-forward network is a directed acyclic graph; information flows in one direction only, and there are no cycles. Such networks cannot represent internal state. Usually, [neural networks](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#neural%20network) are also *layered*, meaning that nodes are organized into groups of layers, and links only go from nodes to nodes in adjacent layers.

Recurrent networks allow loops, and as a result can represent state, though they are much more complex to analyze. *Hopfield networks* and *Boltzmann machines* are examples of recurrent networks; Hopfield networks are the best understood. All connections in Hopfield networks are bidirectional with symmetric weights, all units have outputs of 1 or -1, and the activation function is the sign function. Also, all nodes in a Hopfield network are both input and output nodes. Interestingly, it has been shown that a Hopfield network can reliably recognize *0.138N* training examples, where *N* is the number of units in the network.

Boltzmann machines allow non-input/output units, and they use a stochastic evaluation function that is based upon the sum of the total weighted input. Boltzmann machines are formally equivalent to a certain kind of belief network evaluated with a stochastic simulation algorithm.

One problem in building [neural networks](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#neural%20network) is deciding on the initial topology, e.g., how many nodes there are and how they are connected. Genetic algorithms have been used to explore this problem, but it is a large search space and this is a computationally intensive approach. The

*optimal brain damage* method uses information theory to determine whether weights can be removed from the network without loss of performance, and possibly improving it. The alternative of making the network larger has been tested with the *tiling algorithm* [Mezard and Nadal, 1989] which takes an approach similar to induction on [decision trees](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#decision%20trees); it expands a unit by adding new ones to cover instances it misclassified. *Cross-validation* techniques can be used to determine when the network size is right.

Perceptrons



Perceptrons are single-layer, feed-forward networks that were first studied in the 1950's. They are only capable of [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) *linearly separable* functions. That is, if we view *F* features as defining an *F*-dimensional space, the network can recognize any class that involves placing a single hyperplane between the instances of two classes. So, for example, they can easily represent **AND**, **OR**, or **NOT**, but cannot represent **XOR**.

Perceptrons learn by updating the weights on their links in response to the difference between their output value and the correct output value. The updating rule (due to Frank Rosenblatt, 1960) is as follows. Define *Err* as the difference between the correct output and actual output. Then the [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) rule for each weight is

Wj <- Wj + A x Ij x Err

where *A* is a constant called the *learning rate*.

Of course, this was too good to last, and in *Perceptrons* [Minsky and Papert, 1969] it was observed how limited linearly separable functions were. Work on perceptrons withered, and neural networks didn't come into vogue again until the 1980's, when multi-layer networks became the focus.



Multi-Layer Feed-Forward Networks

The standard method for [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) in multi-layer feed-forward networks is *back-propagation* [Bryson and Ho, 1969]. Such networks have an input layer, and output layer, and one or more *hidden layers* in between. The difficulty is to divide the blame for an erroneous output among the nodes in the hidden layers.

The back-propagation rule is similar to the perceptron [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) rule. If *Erri* is the error at the output node, then the weight update for the link from unit *j* to unit *i* (the output node) is

Wj,i <- Wj,i + A x aj x Erri x g'(ini)

where *g'* is the derivative of the activation function, and *aj* is the activation of the unit *j*. (Note that this means the activation function must have a derivative, so the sigmoid function is usually used rather than the step function.) Define *Di* as *Erri x g'(ini)*.

This updates the weights leading to the output node. To update the weights on the interior links, we use the idea that the hidden node *j* is responsible for part of the error in each of the nodes to which it connects. Thus the error at the output is divided according to the strength of the connection between the output node and the hidden node, and propogated backward to previous layers. Specifically,

Dj = g'(inj) x SUMi Wj,i Di

Thus the updating rule for internal nodes is

Wj,i <- Wj,i + A x aj x Di.

Lastly, the weight updating rule for the weights from the input layer to the hidden layer is is Wk,j <- Wk,j + A x Ik x Dj

where *k* is the input node and *j* the hidden node, and *Ik* is the input value of *k*.

A [neural network](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#neural%20network) requires *2n/n* hidden units to represent all Boolean functions of *n* inputs. For *m* training examples and *W* weights, each epoch in the [learning](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#learning) process takes *O(mW)* time; but in the worst case, the number of [epochs](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#epoch) can be exponential in the number of inputs.

In general, if the number of hidden nodes is too large, the network may learn only the training examples, while if the number is too small it may never converge on a set of weights consistent with the training examples.

Multi-layer feed-forward networks can represent any continuous function with a single hidden layer, and any function with two hidden layers [Cybenko, 1988, 1989].

Applications of [Neural Networks](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#neural%20network)

John Denker remarked that "neural networks are the second best way of doing just about anything." They provide passable performance on a wide variety of problems that are difficult to solve well using other methods.

NETtalk [Sejnowski and Rosenberg, 1987] was designed to learn how to pronounce written text. Input was a seven-character centered on the target character, and output was a set of Booleans controlling the form of the sound to be produced. It learned 95% accuracy on its training set, but had only 78% accuracy on the [test set](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#test%20set). Not spectacularly good, but important because it impressed many people with the potential of [neural networks.](http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/learning.html#neural%20network)

Other applications include a ZIP code recognition [Le Cun et al., 1989] system that achieves 99% accuracy on handwritten codes, and driving [Pomerleau, 1993] in the ALVINN system at CMU. ALVINN controls the NavLab vehicles, and translates inputs from a video image into

steering control directions. ALVINN performs exceptionally well on the particular road-type it learns, but poorly on other terrain types. The extended MANIAC system [Jochem et al., 1993] has multiple ALVINN subnets combined to handle different road types.

# What is conditional planning?Explain.















**\**

# Describe continuous planning.











