Brief Notes

on

Artificial Intelligence
and
Neural Network

Prepared by:

Sujan Tamrakar

Intelligence: It is the ability to acquire and apply knowledge and skills. Intelligence is defined as general cognitive problem-solving skills. A mental ability involved in reasoning, perceiving relationships and analogies, calculating, learning quickly… etc.

Artificial intelligence is the branch of computer science concerned with making computers behave like humans. The term was coined in 1956 by John McCarthy at the Massachusetts Institute of Technology. Artificial intelligence (AI) is the intelligence exhibited by machines or software. It is also the name of the academic field of study which studies how to create computers and computer software that are capable of intelligent behavior. Major AI researchers and textbooks define this field as "the study and design of intelligent agents", in which an intelligent agent is a system that perceives its environment and takes actions that maximize its chances of success.

Artificial intelligence includes the following areas of specialization (objectives):

- Games playing: programming computers to play games against human opponents
- Expert systems: programming computers to make decisions in real-life situations (for example, some expert systems help doctors diagnose diseases based on symptoms)
- Natural language: programming computers to understand natural human languages
- Neural networks: Systems that simulate intelligence by attempting to reproduce the types of physical connections that occur in animal brains
- Robotics: programming computers to *see* and *hear* and react to other sensory stimuli

AI prehistory

• Philosophy	Logic, methods of reasoning, mind as physical system foundations of learning, language, rationality
• Mathematics	Formal representation and proof algorithms, computation, (un)decidability, (in)tractability, probability
• Economics	utility, decision theory
• Neuroscience	physical substrate for mental activity
• Psychology	phenomena of perception and motor control, experimental techniques
• Computer engineering	building fast computers
• Control theory design	systems that maximize an objective function over time
• Linguistics	knowledge representation, grammar

State of the art

- Deep Blue defeated the reigning world chess champion Garry Kasparov in 1997
- Proved a mathematical conjecture (Robbins conjecture) unsolved for decades

- No hands across America (driving autonomously 98% of the time from Pittsburgh to San Diego)
- During the 1991 Gulf War, US forces deployed an AI logistics planning and scheduling program that involved up to 50,000 vehicles, cargo, and people
- NASA's on-board autonomous planning program controlled the scheduling of operations for a spacecraft
- Proverb solves crossword puzzles better than most humans

Views of AI fall into four categories:

1. Thinking humanly
2. Thinking rationally
3. Acting humanly
4. Acting rationally

Acting humanly: The Turing Test approach

The Turing Test, proposed by Alan Turing (Turing, 1950), was designed to provide a satisfactory operational definition of intelligence. Turing defined intelligent behavior as the ability to achieve human-level performance in all cognitive tasks, sufficient to fool an interrogator. Roughly speaking, the test he proposed is that the computer should be interrogated by a human via a teletype, and passes the test if the interrogator cannot tell if there is a computer or a human at the other end. The computer would need to possess the following capabilities:

- natural language processing to enable it to communicate successfully in English (or some other human language);
- knowledge representation to store information provided before or during the interrogation;
- automated reasoning to use the stored information to answer questions and to draw new conclusions;
- machine learning to adapt to new circumstances and to detect and extrapolate patterns.

Turing's test deliberately avoided direct physical interaction between the interrogator and the computer, because *physical* simulation of a person is unnecessary for intelligence. However, the so-called total Turing Test includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects ``through the hatch.'' To pass the total Turing Test, the computer will need

- computer vision to perceive objects, and
- robotics to move them about.

Within AI, there has not been a big effort to try to pass the Turing test. The issue of acting like a human comes up primarily when AI programs have to interact with people, as when an expert system explains how it came to its diagnosis, or a natural language processing system has a dialogue with a user. These programs must behave according to certain normal conventions of human interaction in order to make themselves understood. The underlying representation and reasoning in such a system may or may not be based on a human model.

Thinking humanly: The cognitive modelling approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think. We need to get *inside* the actual workings of human minds. There are two ways to do this: through introspection--trying to catch our own thoughts as they go by--or through psychological experiments. Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input/output and timing behavior matches human behavior, that is evidence that some of the program's mechanisms may also be operating in humans. For example, Newell and Simon, who developed GPS, the ``General Problem Solver'' (Newell and Simon, 1961), were not content to have their program correctly solve problems. They were more concerned with comparing the trace of its reasoning steps to traces of human subjects solving the same problems. This is in contrast to other researchers of the same time (such as Wang (1960)), who were concerned with getting the right answers regardless of how humans might do it. The interdisciplinary field of cognitive science brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind. Although cognitive science is a fascinating field in itself, we are not going to be discussing it all that much in this book. We will occasionally comment on similarities or differences between AI techniques and human cognition. Real cognitive science, however, is necessarily based on experimental investigation of actual humans or animals, and we assume that the reader only has access to a computer for experimentation. We will simply note that AI and cognitive science continue to fertilize each other, especially in the areas of vision, natural language, and learning.

Thinking rationally: The laws of thought approach

The Greek philosopher Aristotle was one of the first to attempt to codify ``right thinking,'' that is, irrefutable reasoning processes. His famous syllogisms provided patterns for argument structures that always gave correct conclusions given correct premises. For example, ``Socrates is a man; all men are mortal; therefore Socrates is mortal.'' These laws of thought were supposed to govern the operation of the mind, and initiated the field of logic.

There are two main obstacles to this approach. First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain. Second, there is a big difference between being able to solve a problem ``in principle'' and doing so in practice. Even problems with just a few dozen facts can exhaust the computational resources of any computer unless it has some guidance as to which reasoning steps to try first. Although both of these obstacles apply to *any* attempt to build computational reasoning systems, they appeared first in the logicist tradition because the power of the representation and reasoning systems are well-defined and fairly well understood.

Acting rationally: The rational agent approach

Acting rationally means acting so as to achieve one's goals, given one's beliefs. An agent is just something that perceives and acts. (This may be an unusual use of the word, but you will get used to it.) In this approach, AI is viewed as the study and construction of rational agents.

In the ``laws of thought'' approach to AI, the whole emphasis was on correct inferences. Making correct inferences is sometimes *part* of being a rational agent, because one way to act rationally is to reason logically to the conclusion that a given action will achieve one's goals, and then to act on that conclusion. On the other hand, correct inference is not *all* of rationality, because there are often situations where there is no provably correct thing to do, yet something must still be done. There are also ways of acting rationally that cannot be reasonably said to involve inference. For

example, pulling one's hand off of a hot stove is a reflex action that is more successful than a slower action taken after careful deliberation.

All the ``cognitive skills'' needed for the Turing Test are there to allow rational actions. Thus, we need the ability to represent knowledge and reason with it because this enables us to reach good decisions in a wide variety of situations. We need to be able to generate comprehensible sentences in natural language because saying those sentences helps us get by in a complex society. We need learning not just for erudition, but because having a better idea of how the world works enables us to generate more effective strategies for dealing with it. We need visual perception not just because seeing is fun, but in order to get a better idea of what an action might achieve--for example, being able to see a tasty morsel helps one to move toward it.

The study of AI as rational agent design therefore has two advantages. First, it is more general than the ``laws of thought'' approach, because correct inference is only a useful mechanism for achieving rationality, and not a necessary one. Second, it is more amenable to scientific development than approaches based on human behavior or human thought, because the standard of rationality is clearly defined and completely general.

Applications of AI

Game playing

> You can buy machines that can play master level chess for a few hundred dollars. There is some AI in them, but they play well against people mainly through brute force computation--looking at hundreds of thousands of positions.

Speech recognition

> In the 1990s, computer speech recognition reached a practical level for limited purposes. Thus United Airlines has replaced its keyboard tree for flight information by a system using speech recognition of flight numbers and city names. It is quite convenient.

Understanding natural language

> Just getting a sequence of words into a computer is not enough. Parsing sentences is not enough either. The computer has to be provided with an understanding of the domain the text is about, and this is presently possible only for very limited domains.

Computer vision

> The world is composed of three-dimensional objects, but the inputs to the human eye and computers' TV cameras are two dimensional. Some useful programs can work solely in two dimensions, but full computer vision requires partial three-dimensional information that is not just a set of two-dimensional views.

Expert systems

> A ``knowledge engineer'' interviews experts in a certain domain and tries to embody their knowledge in a computer program for carrying out some task. How well this works depends on whether the intellectual mechanisms required for the task are within the present state of AI. One of the first expert systems was MYCIN in 1974, which diagnosed bacterial infections of the blood and suggested treatments. It did better than medical students or practicing doctors, provided its limitations were observed. In the present state of AI, this has to be true.

Intelligent Agent

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. An intelligent agent is a software that assists people and act on their behalf. Intelligent agents work by allowing people to delegate work that they could have done, to the agent software. Agents can perform repetitive tasks, remember things you forgot, intelligently summarize complex data, learn from you and even make recommendations to you.

Characteristic of Intelligent Agent

All agents are autonomous, which means that an agent has control over its own actions. All agents are also goal-driven. Agents have a purpose and act accordance with that purpose. There are several ways of making goals known to an agent, and are listed below:

An agent could be driven by a script with pre-defines action which would then define the agent's goals.

An agent could also be a program and as long as the program is driven by goals and has other characteristics of agents.

An agent could also be driven by rules, and the rules would define the agent's goals.

There is also embedded agent goals, such as "planning" methodologies, and in some cases the agent could change its own goals over time. An agent could also senses changes in its environment and responds to these changes. This characteristic of the agent is at the core of delegation and automation. For example, you tell your assistant "when x happens, do y" and the agent is always waiting for x to happen. An agent continue to work even when the user is gone, which means that an agent could run on a server, but in some cases, an agent run on the user systems.

	Human agent	Software agent	Robotic agent
Sensors	Eyes, ears, skin, nose	Keystrokes, clicks, n/w packets	Cameras, infrared, mic
Actuators	Hands, legs, mouth	Display, sounds, send packets	Motors, display

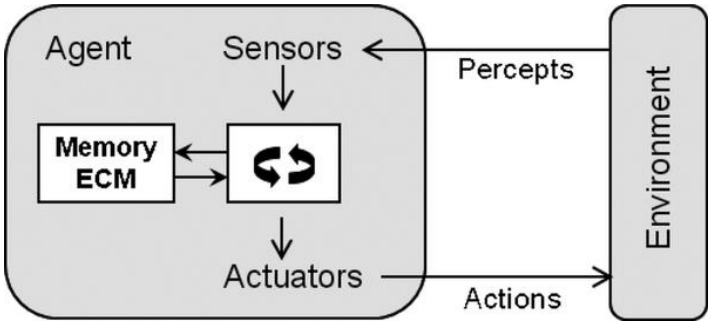


Figure 1 An Intelligent Agent

Structure of an Intelligent Agent:

Agent = architecture + agent program

Architecture = computing device in which program will run

Agent program = function that implements the agent mapping from percepts to actions

PAGE of an agent are: Percepts, Actions, Goals, Environment

Example of and Intelligent Agent (Auto Taxi)

Percepts = camera, microphone, different sensors, GPS

Actions = accelerate, brake, steer, blow horn, speech

Goal = safe, fast, legal, comfort, maximize profit

Environment = road, pedestrians, signals, traffic, customer

Types of Agent (based on Agent program)

Simple Reflex agent: if- then rule / reactive approach

- are really not very bright
- will not accumulate experience
- will not learn from its experience
- If the percept is not in the reflex agent's database, the agent cannot react appropriately to the situation
- Ex: simple mercury type thermostat/ heater

Model based Reflex agent: "how the world works"

- keeping track of the part of the world it can see now
- does this by keeping an internal state that depends on what it has seen before so it holds information on the unobserved aspects of the current state.
- Ex: Mars Lander after picking up its first sample, it stores this in the internal state of the world around it so when it come across the second same sample it passes it by and saves space for other samples.

Goal based agent:

- has a representation of the current state of the environment and how that environment generally works
- These agents **consider different scenarios before acting** on their environments, **to see which action will probably attain a goal**. This consideration of different scenarios is called **search and planning**. It makes the agent **proactive**, not just reactive.

- Another interesting feature of the goal-based agent is that it already has some model of how the objects in its environment usually behave, so it can perform searches and identify plans based on this knowledge about the world.
- Ex: Vacuum world problem

Utility based agent:

- add one more feature to our goal-based agent, to make it even more adaptive
- Agents so far have had a single goal.
- Agents may have to juggle conflicting goals.
- Need to optimise utility over a range of goals.
- Utility: measure of goodness (a real number).
- Combine with probability of success to get expected utility.
- a utility measure is applied to the different possible actions that can be performed in the environment. This sophisticated planner is a utility-based agent
- utility-based agent will rate each scenario to see how well it achieves certain criteria with regard to the production of a good outcome.
- Things like the probability of success, the resources needed to execute the scenario, the importance of the goal to be achieved, the time it will take, might all be factored in to the utility function calculations.
- how desirable a particular state is == *utility function* which maps a state to a measure of the utility of the state
- ex: Auto taxi (Google car)

Problem Solving
An important aspect of intelligence is goal-based problem solving.

The solution of many problems (e.g. noughts and crosses, timetabling, chess) can be described by finding a sequence of actions that lead to a desirable goal. Each action changes the *state* and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.

A well-defined problem can be described by:

- **Initial state**
- **Operator or successor function** - for any state x returns s(x), the set of states reachable from x with one action
- **State space** - all states reachable from initial by any sequence of actions
- **Path** - sequence through state space
- **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
- **Goal test** - test to determine if at goal state

Example Problems

The real art of problem solving is in deciding the description of the states and the operators.

- The 8-puzzle
- 8 Queen problem
- Route finding

Linear Planning

Basic Idea: *Work on one goal until completely solved before moving on to the next goal*

Implications:
– No interleaving of goal achievement
– Efficient search if goals do not interact (much)

• **Advantages**
– Reduced search space, since goals are solved one at a time
– Advantageous if goals are (mainly) independent
– Linear planning is *sound*
• **Disadvantages**
– Linear planning may produce *suboptimal* solutions
(based on the number of operators in the plan)
– Linear planning is *incomplete*

Non-Linear Planning

• Basic Idea
– Use goal *set* instead of goal stack
– Include in the search space all possible subgoal orderings
• Handles goal interactions by *interleaving*
• **Advantages**
– Non-linear planning is *sound*
– Non-linear planning is *complete*
– Non-linear planning may be *optimal* with respect to plan
length (depending on search strategy employed)
• **Disadvantages**
– Larger search space, since all possible goal orderings may
have to be considered
– Somewhat more complex algorithm; More bookkeeping

Problem-solving agents: find sequence of actions that achieve goals.

1. Problem-Solving Agents
    - Problem-Solving Steps:
        1. *Goal transformation*: where a goal is set of acceptable states.
        2. *Problem formation*: choose the operators and state space.
        3. *search*
        4. *execute solution*
2. Formulating Problems

- Types of problems:
    1. *Single state problems*: state is always known with certainty.
    2. *Multi state problems*: know which states might be in.
    3. *Contingency problems*: constructed plans with conditional parts based on sensors.
    4. *Exploration problems*: agent must learn the effect of actions.
- Formal definition of a problem:
    - **Initial state** (or set of states)
    - **set of operators**
    - **goal test on states**
    - **path cost**
- Measuring performance:
    - Does it find a solution?
    - What is the **search cost**?
    - What is the **total cost**?
        - (total cost = path cost + search cost)
- Choosing states and actions:
    - **Abstraction**: remove unnecessary information from representation; makes it cheaper to find a solution.

3. Example Problems:
    - Toy problems:
        - 8-puzzle
        - 8-queen/n-queen
        - crypto arithmetic
        - vacuum world
        - missionaries and cannibals
    - Real World
        - Traveling Salesperson (NP hard)
        - VLSI layout
        - robot navigation
        - assembly sequencing
4. Searching for Solutions
    - Operators **expand** a state: generate new states from present ones.
    - **Search strategy**: tells which state to expand next.
    - **fringe** or **frontier**: discovered states to be expanded
5. Evaluation criteria:
        - **Completeness**: will it find a solution if one exists?
        - **time efficiency**:
        - **space efficiency**:
        - **optimality of solution**:

Well-defined problems

A **problem** can be defined formally by five components:

1. The **initial state** that Agent starts in.
2. A description of the possible **actions** available to Agent.

3. A description of what each action does; the formal name for this the **transition model**, specified by a function that returns the state that results from doing action. The term **successor** refers to any state reachable from a given state by a single action.

   Together, the initial state, actions and transition model implicitly define the **state space** of the problem. The **state space** is the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed network of **graphs** in which the nodes are states and the links between nodes are actions.
4. The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.
5. A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

Constraint Satisfaction

A constraint satisfaction problem (CSP) consists of

- a set of variables,
- a domain for each variable, and
- a set of constraints.

The aim is to choose a value for each variable so that the resulting possible world satisfies the constraints; we want a model of the constraints.
A finite CSP has a finite set of variables and a finite domain for each variable. Many of the methods considered in this chapter only work for finite CSPs, although some are designed for infinite, even continuous, domains.

Constraint satisfaction is the process of finding a solution to a set of constraints that impose conditions that the variables must satisfy. A solution is therefore a set of values for the variables that satisfies all constraints—that is, a point in the feasible region.

The techniques used in constraint satisfaction depend on the kind of constraints being considered. Often used are constraints on a finite domain, to the point that constraint satisfaction problems are typically identified with problems based on constraints on a finite domain. Such problems are usually solved via search, in particular a form of backtracking or local search. Constraint propagation are other methods used on such problems; most of them are incomplete in general, that is, they may solve the problem or prove it unsatisfiable, but not always. Constraint propagation methods are also used in conjunction with search to make a given problem simpler to solve. Other considered kinds of constraints are on real or rational numbers; solving problems on these constraints is done via variable elimination or the simplex algorithm.

Ex: Sudoku, crypt arithmetic

3.5.14.1 Theory of game playing

Let us discuss in brief some basics of theory of game playing from the view point of AI. Game playing was one of the first tasks undertaken in AI. The development of chess game program started in 1950. The AI games are usually considered as deterministic (i.e. deriving some conclusion) and fully observable, in which there are two agents and there actions alternate. In this case, the utility values (like objective function), which have some mathematical value determining goodness of a move, are equal and opposite, e.g. if player A in chess gains 1 point, player B loses 1 point. The name adversial search is derived because of this adverse situation created by A and B. Multiplayer games, non-zero sum games, and stochastic games come under this category. In the game playing strategy, one player tries to maximize his points and minimize other player's points. Because of alternate maximization and minimization of objective function by players, this search strategy is also called *'mini-max' search.*

Formally, a game is defined as a kind of search problem with:

(i) *Initial state*: It includes starting state (board position), and identifies the player to apply a move.

(ii) *An operator selector*: which selects an 'operator' from valid list of operators; and returns a (move, state) pair, each indicating a legal move and the resulting state.

(iii) *Final state or terminal state*: which determines when the game is over.

(iv) *Objective function*: also called utility function, which associates some numeric value with a move to decide which move should be taken in the situation when multiple moves are simultaneously valid. In the game of chess, the outcome is a win, loss or draw. These are given values +1, -1 or 0. Some games have wider variety of possible outcomes, e.g. the payoffs in backgammon range from +192 to -192.

Various moves of a game are represented by game tree. It has the root that represents the initial state and children of root represent various other states. The partial game tree of tic- tac- toe game is shown in Chapter 2.

In a normal search problem, the optimal solution is found by a sequence of valid moves leading to goal state, but in game playing search, the moves alternate between two players and each player while deciding his move, pays attention to following two points:

(i) win his game

(ii) stop the opponent from winning

Accordingly, the moves alternately try to maximize the utility function and minimize the utility function. Moreover, the strategies are contingent, means, while deciding next move every player will check his own move also.

What is Search?

Search is the systematic examination of states to find path from the start/root state to the goal state.

The set of possible states, together with *operators* defining their connectivity constitute the *search space*.

The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

In real life search usually results from a lack of knowledge. In AI too search is merely a offensive instrument with which to attack problems that we can't seem to solve any better way.

Problem Space

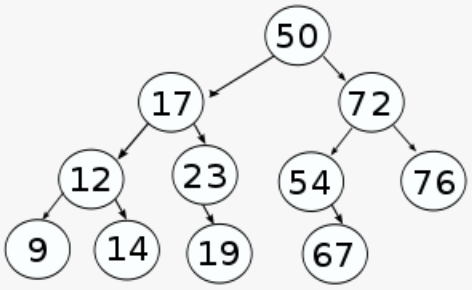


Image via Wikipedia

Problem space is a set of states and a set of operators. The operators map from one state to another state. There will be one or more states that can be called initial states, one or more states which we need to reach what are known as goal states and there will be states in between initial states and goal states known as intermediate states. So what is the solution? The solution to the given problem is nothing but a sequence of operators that map an initial state to a goal state. This sequence forms a solution path. What is the best solution? Obviously, the shortest path from the initial state to the goal state is the best one. Shortest path has only a few operations compared to all other possible solution paths. Solution path forms a tree structure where each node is a state. So searching is nothing but exploring the tree from the root node.

Types of AI search Techniques

Solution can be found with less information or with more information. It all depends on the problem we need to solve. Usually when we have more information it will be easy to solve the problem. There are two kinds of AI search techniques:

Uninformed search and Informed search.

Uninformed Search

Sometimes we may not get much relevant information to solve a problem. Suppose we lost our car key and we are not able to recall where we left, we have to search for the key with some information such as in which places we used to place it. It may be our pant pocket or may be the table drawer. If it is not there then we have to search the whole house to get it. The best solution would be to search in the places from the table to the wardrobe. Here we need to search blindly with fewer clues. This type of search is called uninformed search or blind search. There are two popular AI search techniques in this category:

Breadth first search and Depth first search.

Informed search

We can solve the problem in an efficient manner if we have relevant information, clues or hints. The clues that help solve the problem constitute heuristic information. So informed search is also call heuristic search. Instead of searching one path or many paths, just like that informed search uses the given heuristic information to decide whether to explore the current state further. Hill climbing is an AI search algorithm that explores the neighboring states, chooses the most promising state as successor, and continues searching for the subsequent states. Once a state is explored, hill climbing algorithm simply discards it. Hill climbing search technique can make substantial savings if it has reliable information. It has to face three challenges: foothill, ridge and plateau. Best first search is a heuristic search technique that stores the explored states as well so that it can backtrack if it realizes that the present path proves unworthy.

In summary; there are 2 types of basic search techniques:

1. Uninformed Search techniques (Blind)
   - Depth first search
   - Breadth first search
   - Depth limit search
2. Informed Search techniques (Heuristic)
   - Hill climbing
   - Best first search
   - Greedy search
   - A* search
   - Minimax procedure
   - Alpha Beta procedure

Search Techniques

Iterative Deepening A* Search

Modification of A* search use f-cost limit as depth bound

- Increase threshold as minimum of f(.) of previous cycle
- Each iteration expands all nodes inside the contour for current f-cost
- complete and optimal
- Same order of node expansion
- Storage Efficient – practical
  IDA* do not remember history

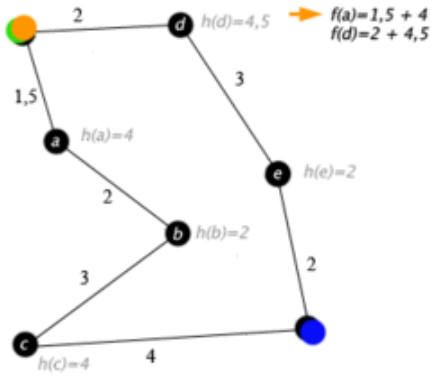
Therefore cannot avoid repeated states.

A* Search

A* uses a best-first search and finds the least-cost path from a given initial node to one goal node (out of one or more possible goals).

It uses a distance-plus-cost heuristic function (usually denoted f(x)) to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions:

- The path-cost function, which is the cost from the starting node to the current node (usually denoted g(x))
- In addition, an admissible "heuristic estimate" of the distance to the goal (usually denoted h(x)).



Best-First Search (Greedy Search)

Best-first search is a search algorithm, which explores a graph by expanding the most promising node chosen according to a specified rule.

Judea Pearl described best-first search as estimating the promise of node n by a "heuristic evaluation function f(n) which, in general, may depend on the description of n, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain."

Some authors have used "best-first search" to refer specifically to a search with a heuristic that attempts to predict how close the end of a path is to a solution, so that paths, which are judged closer to a solution, are extended first. This specific type of search is called greedy best-first search.

Efficient selection of the current best candidate for extension is typically implemented using a priority queue.

Bidirectional search

Bidirectional search is a graph search algorithm that finds a shortest path from an initial vertex to a goal vertex in a directed graph. It runs two simultaneous searches: one forward from the initial state and one backward from the goal, stopping when the two meet in the middle. The reason for this approach is that in many cases it is faster: for instance, in a simplified model of search problem complexity in which both searches expand a tree with branching factor b, and the distance from start to goal is d, each of the two searches has complexity $O(b^{d/2})$ (in Big O notation), and the sum of these two search times is much less than the $O(b^d)$ complexity that would result from a single search from the beginning to the goal.

Iterative Deepening Search

Iterative deepening depth-first search (IDDFS) is a state space search strategy in which a depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches d, the depth of the shallowest goal state. On each iteration, IDDFS visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited, assuming no pruning, is effectively breadth-first.

Depth-Limited Search

Like the normal depth-first search, depth-limited search is an uninformed search. It works exactly like depth-first search, but avoids its drawbacks regarding completeness by imposing a maximum limit on the depth of the search. Even if the search could still expand a vertex beyond that depth, it will not do so and thereby it will not follow infinitely deep paths or get stuck in cycles. Therefore depth-limited search will find a solution if it is within the depth limit, which guarantees at least completeness on all graphs.

Breadth First Search Vs Depth First Search

Breadth First Search	Depth First Search
Advantages	
Optimal solutions are always found Multiple solutions found early	May arrive at solutions without examining much of search space
Will not go down blind alley for solution	Needs little memory (only node in current path needs to be stored)
Disadvantages	
If solution path is long, the whole tree must be searched up to that depth	May settle for non-optimal solution
All of the tree generated must be stored in memory	May explore single unfruitful path for a long time (forever if loop exists!)

Depth First Search

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it has not finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

Uniform Cost Search

Uniform-cost search (UCS) is a tree search algorithm used for traversing or searching a weighted tree, tree structure, or graph. The search begins at the root node. The search continues by visiting the next node, which has the least total cost from the root. Nodes are visited in this manner until a goal state is reached.

Typically, the search algorithm involves expanding nodes by adding all unexpanded neighboring nodes that are connected by directed paths to a priority queue. In the queue, each node is associated with its total path cost from the root, where the least-cost paths are given highest priority. The node at the head of the queue is subsequently expanded, adding the next set of connected nodes with the total path cost from the root to the respective node.

Breadth First Search

BFS is an uninformed search method that aims to expand and examine all nodes of a graph or combination of sequences by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it. It does not use a heuristic algorithm.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO (i.e., First In, First Out) queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called “open” and then once examined are placed in the container “closed”.

Hill Climbing

It is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

For example, hill climbing can be applied to the travelling salesman problem. It is easy to find an initial solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much shorter route is likely to be obtained.

Hill climbing is good for finding a local optimum (a solution that cannot be improved by considering a neighboring configuration) but it is not necessarily guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space).

Fuzzy Logic

Fuzzy logic is a form of many-valued logic in which the truth values of variables may be any real number between 0 and 1. By contrast, in Boolean logic, the truth values of variables may only be 0 or 1. Fuzzy logic has been extended to handle the concept of partial truth, where the truth value may range between completely true and completely false. A type of logic that recognizes more than simple true and false values. For example, the statement, *today is sunny,* might be 100% true if there are no clouds, 80% true if there are a few clouds, 50% true if it's hazy and 0% true if it rains all day.

Fuzzy logic has proved to be particularly useful in expert system and other artificial intelligence applications. It is also used in some spell checkers to suggest a list of probable words to replace a misspelled one.

Means End analysis

It is a problem-solving technique in which the current state is compared to the goal state, and the difference between them is divided up into sub goals in order to achieve the goal state by the use of the available operators. Means-End analysis is one of many weak search methods that have been utilized in both cognitive architectures and more general artificial intelligence research. Most of the search strategies either reason forward or backward however, often a mixture of the two directions is appropriate. Such mixed strategy would make it possible to solve the major parts of problem first and solve the smaller problems that arise when combining them together. Such a technique is called "Means - Ends Analysis".

The means -ends analysis process centers around finding the difference between current state and goal state. The problem space of means - ends analysis has an initial state and one or more goal state, a set of operate with a set of preconditions their application and difference functions that computes the difference between two state a(i) and s(j).

How MEA works

The MEA technique is a strategy to control search in problem-solving. Given a current state and a goal state, an action is chosen which will reduce the difference between the two. The action is performed on the current state to produce a new state, and the process is recursively applied to this new state and the goal state.

Note that, in order for MEA to be effective, the goal-seeking system must have a means of associating to any kind of detectable difference those actions that are relevant to reducing that difference. It must also have means for detecting the progress it is making (the changes in the differences between the actual and the desired state), as some attempted sequences of actions may fail and, hence, some alternate sequences may be tried.

When knowledge is available concerning the importance of differences, the most important difference is selected first to further improve the average performance of MEA over other brute-force search strategies. However, even without the ordering of differences according to importance, MEA improves over other search heuristics (again in the average case) by focusing the problem solving on the actual differences between the current state and that of the goal.

Production systems

A Knowledge representation formalism consists of collections of condition-action rules (Production Rules or Operators), a database which is modified in accordance with the rules, and a Production System Interpreter which controls the operation of the rules i.e. the 'control mechanism' of a Production System, determining the order in which Production Rules are fired.

A system that uses this form of knowledge representation is called a production system. A production system is a tool used in artificial intelligence and especially within the applied AI domain known as expert systems. Production systems consist of a database of rules, a working memory, a matcher, and a procedure that resolves conflicts between rules.

A production system consists of rules and factors. Knowledge is encoded in a declarative from which comprises of a set of rules of the form. Productions consist of two parts: a sensory precondition (or "IF" statement) and an action (or "THEN"). If a production's precondition matches the current state of the world, then the production is said to be triggered. If a production's action is executed, it is said to have fired. A production system also contains a database, sometimes called working memory, which maintains data about current state or knowledge, and a rule interpreter. The rule interpreter must provide a mechanism for prioritizing productions when more than one is triggered.

Situation ---------- Action

SITUATION that implies ACTION.

Example:-
IF the initial state is a goal state THEN quit.

The major components of an AI production system are

i. A global database

ii. A set of production rules and

iii. A control system

The goal database is the central data structure used by an AI production system. The production system. The production rules operate on the global database. Each rule has a precondition that is either satisfied or not by the database. If the precondition is satisfied, the rule can be applied. Application of the rule changes the database. The control system chooses which applicable rule should be applied and ceases computation when a termination condition on the database is satisfied. If several rules are to fire at the same time, the control system resolves the conflicts.

Four classes of production systems:-

1. A monotonic production system

2. A non-monotonic production system

3. A partially commutative production system

4. A commutative production system.

Advantages of production systems:-

1. Production systems provide an excellent tool for structuring AI programs.

2. Production Systems are highly modular because the individual rules can be added, removed or modified independently.

Disadvantages of Production Systems:-
One important disadvantage is the fact that it may be very difficult analyze the flow of control within a production system because the individual rules don't call each other.

Ex: Water Jug Problem

You have a 4 liter jug and a 3 liter jug, a supply of water, and a drain. How can you measure exactly 2 liters of water? Here are the rules: You can

- fill either jug from the water source
- pour water from one jug into the other
- empty a jug down the drain

Solution:
Fill the 3 liter jug from the tap (0,3)
Pour this into the 4 liter jug (3,0)
Fill the 3 liter jug from the tap (3,3)
Pour from the 3 liter jug into the 4 liter jug until full. This leaves 2 liters in the 3 liter jug (4,2)
There are other solutions!

The two jugs can be represented by an ordered pair (a,b) where a = 0, 1, 2, 3 or 4, the number of litres in the larger jug, while b = 0, 1, 2 or 3, the number of liters in the smaller jug.
Hence our initial state is (0,0) and the goal state is (2,0).

There are several options at each stage. Here they are:

- Fill the 4l jug
- Fill the 3l jug
- Empty the 4l jug on the ground
- Empty the 3l jug on the ground
- Pour water from the 3l jug to fill the 4l jug
- Pour water from the 4l jug to fill the 3l jug
- Empty the 3l jug into the 4l jug
- Empty the 4l jug into the 3l jug

Solving the problem in this mathematical way is called production rules:

	Condition	New State	Description
1	(a,b) if a < 4	(4,b)	Fill the 4l jug
2	(a,b) if b < 3	(a,3)	Fill the 3l jug
3	(a,b) if a > 0	(0,b)	Empty the 4l jug on the ground
4	(a,b) if b > 0	(a,0)	Empty the 3l jug on the ground
5	(a,b) if a+b >= 4 and b > 0	(4,b-4+a)	Pour water from the 3l jug to fill the 4l jug
6	(a,b) if a+b >= 3 and a > 0	(a-3+b,3)	Pour water from the 4l jug to fill the 3l jug
7	(a,b) if a+b <= 4 and b > 0	(a+b,0)	Empty the 3l jug into the 4l jug
8	(a,b) if a+b <= 3 and a > 0	(0,a+b)	Empty the 4l jug into the 3l jug

Knowledge Representation and reasoning

Knowledge representation and reasoning (KR) is the field of artificial intelligence (AI) dedicated to representing information about the world in a form that a computer system can utilize to solve complex tasks such as diagnosing a medical condition or having a dialog in a natural language. Knowledge representation incorporates findings from psychology about how humans solve problems and represent knowledge in order to design formalisms that will make complex systems easier to design and build. Knowledge representation and reasoning also incorporates findings from logic to automate various kinds of reasoning, such as the application of rules or the relations of sets and subsets.
Examples of knowledge representation formalisms include semantic nets, Frames, Rules, and ontologies. Examples of automated reasoning engines include inference engines, theorem provers, and classifiers.
The main features of such a Knowledge representation language are:
1. *Object-orientedness*. All the information about a specific concept is stored with that concept, as opposed, for example, to rule-based systems where information about one concept may be scattered throughout the rule base.
2. *Generalization/Specialization*. Long recognized as a key aspect of human, KR languages provide a natural way to group concepts in hierarchies in which higher level concepts represent more general, shared attributes of the concepts below.
3. *Reasoning*. The ability to state in a formal way that the existence of some piece of knowledge implies the existence of some other, previously unknown piece of knowledge, is important to KR. Each KR language provides a different approach to reasoning.
4. *Classification*. Given an abstract description of a concept, most KR languages provide the ability to determine if a concept fits that description, this is actually a common special form of reasoning.
Object orientation and generalization help to make the represented knowledge more understandable to humans, reasoning and classification help make a system behave as if it knows what is represented. Frame-based systems thus meet the goals of the KR Hypothesis.

Logic makes statements about the world which are true (or false) if the state of affairs it represents is the case (or not the case). Compared to natural languages (expressive but context sensitive) and programming languages (good for concrete data structures but not expressive) logic combines the advantages of natural languages and formal languages. Logic is:

- concise
- unambiguous
- context insensitive
- expressive
- effective for inferences

A logic is defined by the following:

1. **Syntax** - describes the possible configurations that constitute sentences.
2. **Semantics** - determines what facts in the world the sentences refer to i.e. the interpretation. Each sentence makes a claim about the world.
3. **Proof theory** - set of rules for generating new sentences that are necessarily true given that the old sentences are true. The relationship between sentences is called **entailment**. The semantics link these sentences (representation) to facts of the world. The proof can be used to determine new facts which follow from the old.

We will consider two kinds of logic: **propositional logic** and **first-order logic** or more precisely first-order **predicate calculus**.

Types of logic:

1. Propositional logic (Boolean logic)
2. Predicate logic (First order predicate calculus)

Propositional logic

Propositional logic (also called sentential logic) is the logic the includes sentence letters (A,B,C) and logical connectives, but not quantifiers. The semantics of propositional logic uses truth assignments to the letters to determine whether a compound propositional sentence is true.

- defines a language for symbolic reasoning
- a statement that is either true or false
- Examples of propositions:
  - Pitt is located in the Oakland section of Pittsburgh.
  - France is in Europe.
  - It rains outside.
  - 2 is a prime number and 6 is a prime

Sentences in the propositional logic:

• Atomic sentences: – Constructed from constants and propositional symbols

– True, False are (atomic) sentences

• Composite sentences: – Constructed from valid sentences via connectives. Complex sentences are constructed from simpler sentences using logical connectives: ¬ (not), ∧ (and), ∨ (or), → (implies) [I prefer → to ⇒], and ↔ (iff).

Example:

A := "'Aristotle is dead."'

B := "'Hildesheim is on the Rhine."'

C := "'Logic is fun."'

Syntax:

A proposition or propositional sentence can be formed as follows:
Every propositional symbol is a sentence.
If A is a sentence, then ¬A is a sentence.
If A1 and A2 are sentences, then so are:
A1 ∧ A2, (and, conjunction)
A1 ∨ A2, (or, disjunction)
A1 → A2, (implies, implication)
A1 ↔ A2. (iff, biconditional)

Semantics:

A world or domain is the set of facts we want to represent. An interpretation maps each propositional symbol to the world. A sentence is true if its interpretation in the world is true. A knowledge base is a set of sentences. A world is a model of a KB if the KB is true for that world. For convenience, a model can be thought of as a truth assignment to the symbols.

Deduction:
A sentence S is satisfiable within a KB if S is true in some model of the KB, i.e., some truth assignment makes S and the KB true. A sentence S is entailed by a KB if S is true in all models of the KB (denoted as KB |= S), i.e., every truth assignment that makes KB true also makes S true. Logical inference or deduction is concerned with producing entailed sentences from KBs.

Inference Rules

- Modus Ponens: From $A \rightarrow B$ and A / Infer B
- Unit Resolution: From $A \vee B$ and $\neg B$ / Infer A
- Resolution: From $A \vee B$ and $\neg B \vee C$ / Infer $A \vee C$

Inference Procedures

- An *inference procedure* uses inference rules to produce proofs. Denoted as $KB \vdash S$.
- An inference procedure is *sound* if it can only prove entailed sentences, i.e., every sentence it proves is entailed.
- An inference procedure is *complete* if it can prove any entailed sentence, i.e., every entailed sentence can be proved.

Resolution Inference Procedure

- Resolution applies to *conjunctive normal form*.
  - A KB is a conjunction of sentences.
  - Each sentence is a disjunction of literals.
- Resolution is *refutation complete*. If a KB is in CNF, and if the KB is inconsistent, then resolution will infer inconsistent literals.
- To deduce a sentence S, first temporarily add $\neg S$ to the KB. Then, if resolution infers inconsistent literals, then $\neg S$ is not satisfiable within the KB, which means that S is entailed.

Useful Equivalences for Converting to CNF

Several logical equivalences are handy for converting sentences to CNF

- $(p \rightarrow q) \equiv (\neg p \vee q)$
- $((p \wedge r) \rightarrow (q \vee s)) \equiv (\neg p \vee \neg r \vee q \vee s)$
- $(p \rightarrow (q \wedge r))$
  $\equiv ((p \rightarrow q) \wedge (p \rightarrow r))$
  $\equiv ((\neg p \vee q) \wedge (\neg p \vee r))$
- $((p \vee q) \rightarrow r)$
  $\equiv ((p \rightarrow r) \wedge (q \rightarrow r))$
  $\equiv ((\neg p \vee r) \wedge (\neg q \vee r))$

Contradiction and Tautology

Some composite sentences may always (under any interpretation) evaluate to a single truth value:

- **Contradiction** (always ***False***)

$$P \wedge \neg P$$

- **Tautology** (always ***True***)

$$P \vee \neg P$$

Predicate logic is usually used as a synonym for first-order logic, but sometimes it is used to refer to other logics that have similar syntax. Syntactically, first-order logic has the same connectives as propositional logic, but it also has variables for individual objects, quantifiers, symbols for functions, and symbols for relations. The semantics include a domain of discourse for the variables and quantifiers to range over, along with interpretations of the relation and function symbols. This formal system is distinguished from other systems in that its formulae contain variables which can be quantified. Two common quantifiers are the existential ∃ ("there exists") and universal ∀ ("for all") quantifiers. The variables could be elements in the universe under discussion, or perhaps relations or functions over that universe. For instance, an existential quantifier over a function symbol would be interpreted as modifier "there is a function". Predicate logic is a formal language (like a programming language) with rules for syntax (i.e. how to write expressions) and semantics (i.e. how to formalise the meaning of expressions).

Syntactic Elements:

A symbol in first-order logic can be a predicate, a function, a constant term, or a variable term. First-order logic uses the connectives ¬ (not), ∧ (and), ∨ (or), → (implies), and ↔ (iff). First-order logic also uses the quantifiers ∀ (for all) and ∃ (there exists).

Syntax:

A term is a constant or variable, or a function applied to a sequence of terms. A ground term is a term with no variables. An atomic sentence or atom is a predicate applied to a sequence of terms. A ground atom is an atom with no variables. Connectives can be used to construct more complex sentences in the usual way.
If A is a sentence and x is a variable, then:
– ∀x A is a sentence (universal quantifier).
– ∃x A is a sentence (existential quantifier).
A well-formed formula is a sentence in which all the variables are quantified.

Semantics:

The connectives ∧, ∨, ¬, →, and ↔ are evaluated in the usual way. ∀x A is true if every substitution for x makes A true. ∃x A is true if at least one substitution for x makes A true. An interpretation consists of objects in the world and mappings for terms and predicates. Each ground term is mapped to an object. Each predicate is mapped to a relation (think relational database). A ground atom is true if the predicate's relation holds between the terms' objects.

Translating English to FOL

- Every gardener likes the sun.
  (Ax) gardener(x) => likes(x,Sun)
- You can fool some of the people all of the time.
  (Ex) (person(x) ^ (At)(time(t) => can-fool(x,t)))
- You can fool all of the people some of the time.
  (Ax) (person(x) => (Et) (time(t) ^ can-fool(x,t)))
- All purple mushrooms are poisonous.
  (Ax) (mushroom(x) ^ purple(x)) => poisonous(x)
- No purple mushroom is poisonous.
  ~(Ex) purple(x) ^ mushroom(x) ^ poisonous(x)
  or, equivalently,
  (Ax) (mushroom(x) ^ purple(x)) => ~poisonous(x)
- There are exactly two purple mushrooms.
  (Ex)(Ey) mushroom(x) ^ purple(x) ^ mushroom(y) ^ purple(y) ^ ~(x=y) ^ (Az)
  (mushroom(z) ^ purple(z)) => ((x=z) v (y=z))
- Deb is not tall.
  ~tall(Deb)

Clause Normal Form

Need to convert KB and query to Conjunctive Normal Form (**CNF**).

- Eliminate bi-conditional through equivalence $A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$.
- Eliminate implication through equivalence $A \Rightarrow B \equiv (\neg A) \vee B$.
- Eliminate double negatives: $\neg (\neg A) \equiv A$, $\neg(A \vee B) \equiv \neg A \wedge \neg B$, $\neg (A \wedge B) \equiv \neg A \vee \neg B$.
- Distribute $\vee$ over $\wedge$.

FOPL statement	Clauses
$S1: (\forall x)(kangaroo(x) \rightarrow mammal(x))$	$C1: \sim kangaroo(x) \vee mammal(x)$
$S2: (\forall y)(whale(y) \rightarrow \sim kangaroo(y))$	$C2: \sim whale(y) \vee \sim kangaroo(y)$
$S3: (\exists z)(mammal(z) \wedge \sim furry(z))$	$C3: mammal(M)$
	$C4: \sim furry(M)$
$S4: kangaroo(Kate)$	$C5: kangaroo(Kate)$
$S5: whale(Wally)$	$C6: whale(Wally)$
$S6: (\forall k)(\exists p)(kangaroo(k) \rightarrow (pouch(p) \wedge body\_part(k, p))$	$C7: \sim kangaroo(k) \vee pouch(F1(k))$
	$C8: \sim kangaroo(k) \vee body\_part(k, F1(k))$
$S7: (\forall f)((mammal(f) \wedge \sim furry(f)) \rightarrow whale(f))$	$C9: \sim mammal(f) \vee furry(f) \vee whale(f)$
$S8: (\forall w)(whale(w) \rightarrow mammal(w))$	$C10: \sim whale(w) \vee mammal(w)$

Converting FOL Sentences to Clause Form [in details]

- Every FOL sentence can be converted to a logically equivalent sentence that is in a "normal form" called **clause form**
- Steps to convert a sentence to clause form:
    1. Eliminate all <=> connectives by replacing each instance of the form (P <=> Q) by the equivalent expression ((P => Q) ^ (Q => P))
    2. Eliminate all => connectives by replacing each instance of the form (P => Q) by (~P v Q)
    3. Reduce the scope of each negation symbol to a single predicate by applying equivalences such as converting ~~P to P; ~(P v Q) to ~P ^ ~Q; ~(P ^ Q) to ~P v ~Q; ~(Ax)P to (Ex)~P, and ~(Ex)P to(Ax)~P
    4. Standardize variables: rename all variables so that each quantifier has its own unique variable name. For example, convert (Ax)P(x) to (Ay)P(y) if there is another place where variable x is already used.
    5. Eliminate existential quantification by introducing **Skolem functions**. For example, convert (Ex)P(x) to P(c) where c is a brand new constant symbol that is not used in any other sentence. c is called a **Skolem constant**. More generally, if the existential quantifier is within the scope of a universal quantified variable, then introduce a Skolem function that depend on the universally quantified variable. For example, (Ax)(Ey)P(x,y) is converted to (Ax)P(x, f(x)). f is called

a **Skolem function**, and must be a brand new function name that does not occur in any other sentence in the entire KB.
    - Example: (Ax)(Ey)loves(x,y) is converted to (Ax)loves(x,f(x)) where in this case f(x) specifies the person that x loves. (If we knew that everyone loved their mother, then f could stand for the mother-of function.
6. Remove universal quantification symbols by first moving them all to the left end and making the scope of each the entire sentence, and then just dropping the "prefix" part. For example, convert(Ax)P(x) to P(x).
7. Distribute "and" over "or" to get a conjunction of disjunctions called **conjunctive normal form**. Convert (P ^ Q) v R to (P v R) ^ (Q v R), and convert (P v Q) v R to (P v Q v R).
8. Split each conjunct into a separate **clause**, which is just a disjunction ("or") of negated and un-negated predicates, called **literals**.
9. Standardize variables apart again so that each clause contains variable names that do not occur in any other clause.

- Example

  Convert the sentence (Ax)(P(x) => ((Ay)(P(y) => P(f(x,y))) ^ ~(Ay)(Q(x,y) => P(y))))

  1. Eliminate <=>
     Nothing to do here.
  2. Eliminate =>
     (Ax)(~P(x) v ((Ay)(~P(y) v P(f(x,y))) ^ ~(Ay)(~Q(x,y) v P(y))))
  3. Reduce scope of negation
     (Ax)(~P(x) v ((Ay)(~P(y) v P(f(x,y))) ^ (Ey)(Q(x,y) ^ ~P(y))))
  4. Standardize variables
     (Ax)(~P(x) v ((Ay)(~P(y) v P(f(x,y))) ^ (Ez)(Q(x,z) ^ ~P(z))))
  5. Eliminate existential quantification
     (Ax)(~P(x) v ((Ay)(~P(y) v P(f(x,y))) ^ (Q(x,g(x)) ^ ~P(g(x)))))
  6. Drop universal quantification symbols
     (~P(x) v ((~P(y) v P(f(x,y))) ^ (Q(x,g(x)) ^ ~P(g(x)))))
  7. Convert to conjunction of disjunctions
     (~P(x) v ~P(y) v P(f(x,y))) ^ (~P(x) v Q(x,g(x))) ^ (~P(x) v ~P(g(x)))
  8. Create separate clauses
     - ~P(x) v ~P(y) v P(f(x,y))
     - ~P(x) v Q(x,g(x))
     - ~P(x) v ~P(g(x))
  9. Standardize variables
     - ~P(x) v ~P(y) v P(f(x,y))
     - ~P(z) v Q(z,g(z))
     - ~P(w) v ~P(g(w))

Forward Reasoning

- Works from a set of facts and rules towards a set of conclusions, diagnoses or recommendations.

- When a fact matches the antecedent of a rule
    - Rule fires
    - Conclusion added to facts

Conflict Resolution

- Problem: more than one rule fires at once
- Conflict resolution strategy decides which conclusions to use.
    - Give rules priorities and to use the conclusion that has the highest priority.
    - Apply the rule with the longest antecedent
    - Apply the rule that was most recently added to the database.

Backward Reasoning

- Use backward chaining when proving a particular conclusion
- Works back from a conclusion towards the original facts.
- When a conclusion matches the conclusion of a rule in the database, the antecedents of the rule are compared with facts in the database.

- *Reason forward from the initial states.* Begin building a tree of move sequences that might be solutions by starting with the initial configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose *left* sides match the root node and using their right sides to create the new

configurations. Generate the next level by taking each node generated at the previous level and applying to it all of the rules whose left sides match it. Continue until a configuration that matches the goal state is generated.

- *Reason backward from the goal states.* Begin building a tree of move sequences that might be solutions by starting with the goal configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose *right* sides match the root node. These are all the rules that, if only we could apply them, would generate the state we want. Use the left sides of the rules to generate the nodes at this second level of the tree. Generate the next level of the tree by taking each node at the previous level and finding all the rules whose right sides match it. Then use the corresponding left sides to generate the new nodes. Continue until a node that matches the initial state is generated. This method of reasoning backward from the desired final state is often called *goal-directed reasoning*.

Notice that the same rules can be used both to reason forward from the initial state and to reason backward from the goal state. To reason forward, the left sides (the preconditions) are matched against the current state and the right sides (the results) are used to generate new nodes until the goal is reached. To reason backward, the right sides are matched against the current node and the left sides are used to generate new nodes representing new goal states to be achieved. This continues until one of these goal states is matched by an initial state.

BAYESIAN NETWORK

is a probabilistic graphical model (a type of statistical model) that represents a set of random variables and their conditional dependencies via a directed acyclic graph (DAG). A belief network, also called a Bayesian network, is an acyclic directed graph (DAG), where the nodes are random variables. There is an arc from each element of parents (Xi) into Xi. Associated with the belief network is a set of conditional probability distributions - the conditional probability of each variable given its parents (which includes the prior probabilities of those variables with no parents).

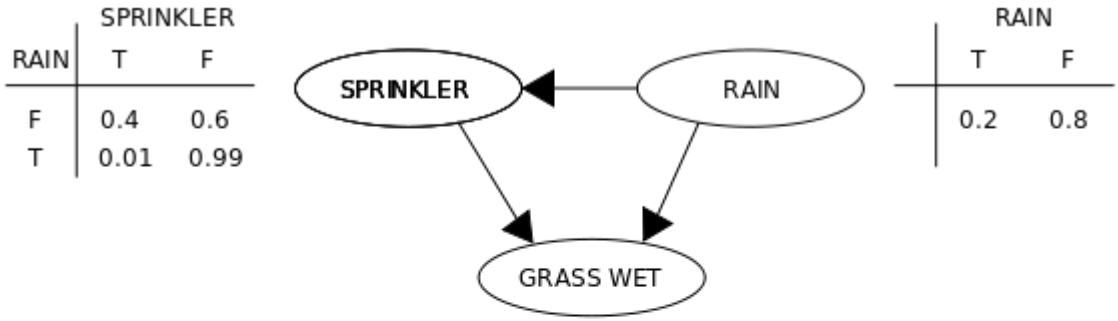
Thus, a belief network consists of

- a DAG, where each node is labeled by a random variable;
- a domain for each random variable; and
- a set of conditional probability distributions giving P(X|parents(X)) for each variable X.

A belief network is acyclic by construction. The way the chain rule decomposes the conjunction gives the ordering. A variable can have only predecessors as parents. Different decompositions can result in different belief networks.

For example, a Bayesian network could represent the probabilistic relationships between diseases and symptoms. Given symptoms, the network can be used to compute the probabilities of the presence of various diseases.

Formally, Bayesian networks are DAGs whose nodes represent random variables in the Bayesian sense: they may be observable quantities, latent variables, unknown parameters or hypotheses. Edges represent conditional dependencies; nodes that are not connected represent variables that are conditionally independent of each other. Each node is associated with a probability function that takes, as input, a particular set of values for the node's parent variables, and gives (as output) the probability (or probability distribution, if applicable) of the variable represented by the node.



		GRASS WET	
SPRINKLER	RAIN	T	F
F	F	0.0	1.0
F	T	0.8	0.2
T	F	0.9	0.1
T	T	0.99	0.01

One advantage of Bayesian networks is that it is intuitively easier for a human to understand (a sparse set of) direct dependencies and local distributions than complete joint distributions.

Top-down and bottom-up reasoning

In the water sprinkler example, we had evidence of an effect (wet grass), and inferred the most likely cause. This is called diagnostic, or "bottom up", reasoning, since it goes from effects to causes; it is a common task in expert systems. Bayes nets can also be used for causal, or "top down", reasoning. For example, we can compute the probability that the grass will be wet given that it is cloudy. Hence Bayes nets are often called "generative" models, because they specify how causes generate effects.

Case based reasoning

Case-based reasoning (CBR), broadly construed, is the process of solving new problems based on the solutions of similar past problems. An auto mechanic who fixes an engine by recalling another car that exhibited similar symptoms is using case-based reasoning. A lawyer who advocates a particular outcome in a trial based on legal precedents or a judge who creates case law is using case-based reasoning. So, too, an engineer copying working elements of nature (practicing bio mimicry), is treating nature as a database of solutions to problems. Case-based reasoning is a prominent kind of analogy making. In case-based reasoning, the training examples - the cases - are stored and accessed to solve a new problem. To get a prediction for a new example, those cases that are similar, or close to, the new example are used to predict the value of the target features of the new example. This is at one extreme of the learning problem where, unlike decision trees and neural networks, relatively little work must be done offline, and virtually all of the work is performed at query time.

It has been argued that case-based reasoning is not only a powerful method for computer reasoning, but also a pervasive behavior in everyday human problem solving; or, more radically, that all reasoning is based on past cases personally experienced. This view is related to prototype theory, which is most deeply explored in cognitive science.

Process

Case-based reasoning has been formalized for purposes of computer reasoning as a four-step process:

1. **Retrieve:** Given a target problem, retrieve from memory cases relevant to solving it. A case consists of a problem, its solution, and, typically, annotations about how the solution was derived. For example, suppose Fred wants to prepare blueberry pancakes. Being a novice cook, the most relevant experience he can recall is one in which he successfully made plain pancakes. The procedure he followed for making the plain pancakes, together with justifications for decisions made along the way, constitutes Fred's retrieved case.
2. **Reuse:** Map the solution from the previous case to the target problem. This may involve adapting the solution as needed to fit the new situation. In the pancake example, Fred must adapt his retrieved solution to include the addition of blueberries.
3. **Revise:** Having mapped the previous solution to the target situation, test the new solution in the real world (or a simulation) and, if necessary, revise. Suppose Fred adapted his pancake solution by adding blueberries to the batter. After mixing, he discovers that the batter has turned blue – an undesired effect. This suggests the following revision: delay the addition of blueberries until after the batter has been ladled into the pan.
4. **Retain:** After the solution has been successfully adapted to the target problem, store the resulting experience as a new case in memory. Fred, accordingly, records his new-found procedure for making blueberry pancakes, thereby enriching his set of stored experiences, and better preparing him for future pancake-making demands.

CBR technology has resulted in the deployment of a number of successful systems, the earliest being Lockheed's CLAVIER, a system for laying out composite parts to be baked in an industrial convection oven. CBR has been used extensively in help desk applications such as the Compaq SMART system and has found a major application area in the health sciences.

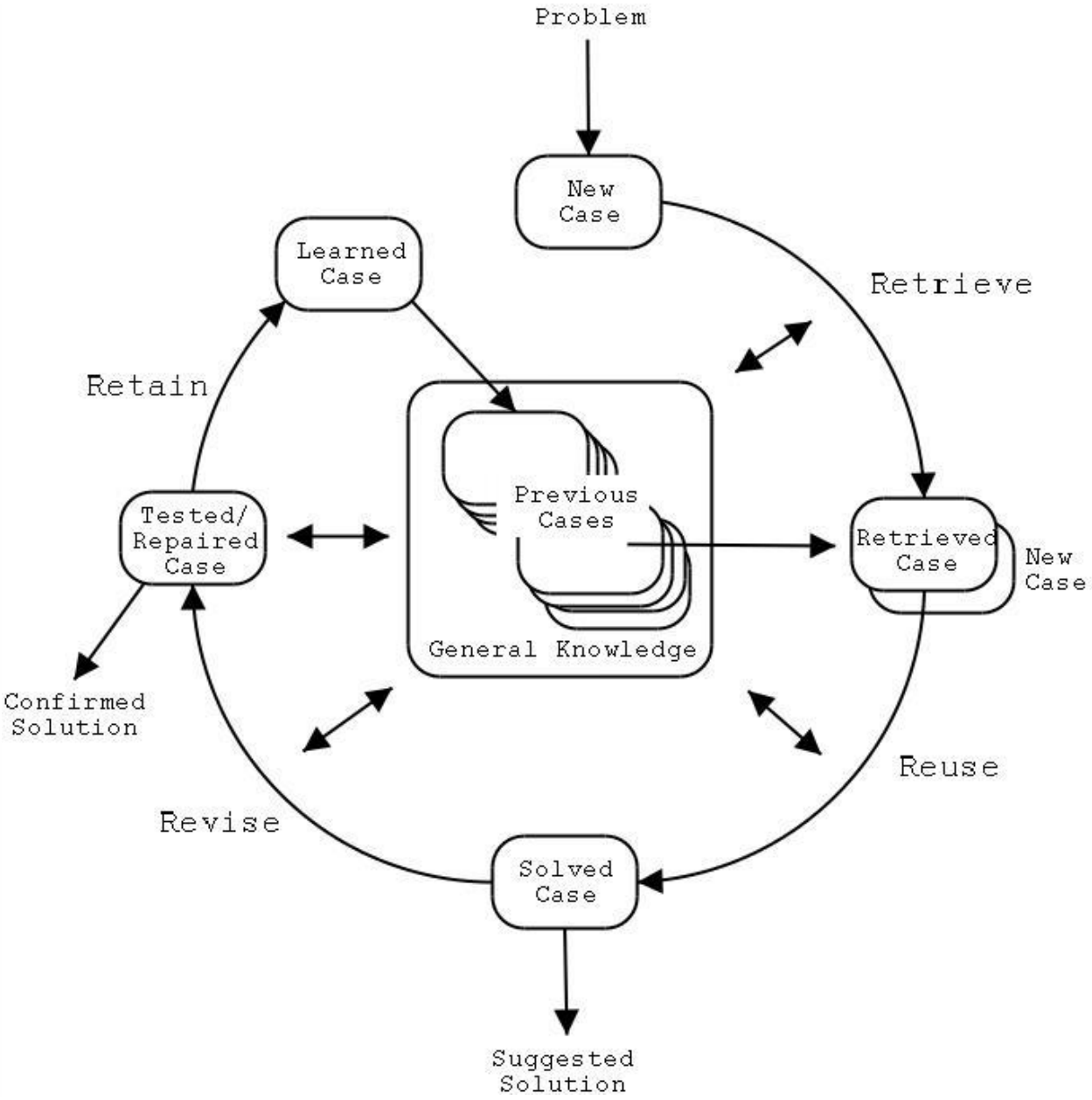


Figure 2 Case Based Reasoning Cycle

Machine learning

Ex: spam filtering, optical character recognition (OCR), search engines and computer vision

- **machine learning is one of many areas in artificial intelligence.**
- "Field of study that gives computers the ability to learn without being explicitly programmed"
- involves development of self-learning algorithms.

- machine is intelligent if it has capabilities of NLP, Knowledge Representation, Automated Reasoning, Machine Learning features.

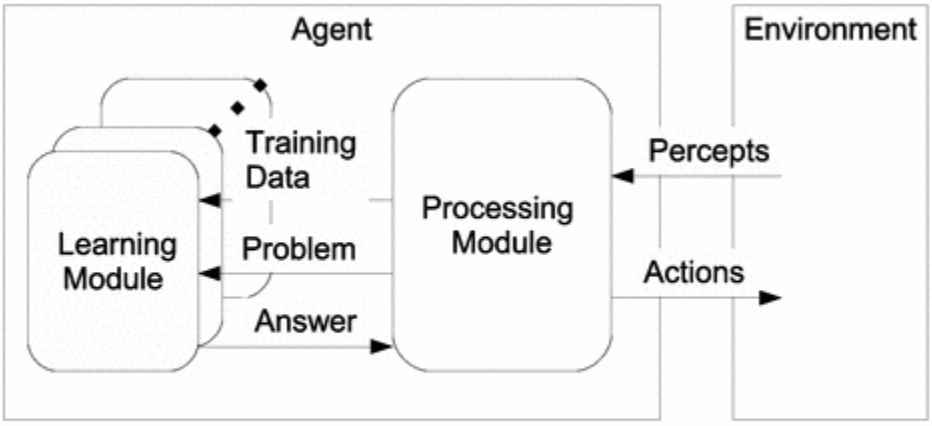


Fig. 1. Learning agent architecture

3 main categories of learning algorithm:

- **Supervised learning**. "supervision from an external source" The computer is presented with example inputs and their desired outputs, given by a "teacher", and the goal is to learn a general rule that maps inputs to outputs.

  - ✓ student taking an exam, having it marked and then being shown which questions they answered incorrectly. After being shown the correct answers, the student should then learn to answer those questions successfully as well
  - ✓ you get bunch of photos **with information what is on them** and you train a model to recognize new photos
  - ✓ you have bunch of molecules and **information which are drugs** and you train a model to answer whether new molecule is also a drug

- **Unsupervised learning**, no labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end.

  - ✓ someone learning to juggle by themselves. The person will start by throwing the balls and attempting to catch them again. After dropping most of the balls initially, they will gradually adjust their technique and start to keep the balls in the air.
  - ✓ you have bunch of photos of 6 people but **without information who is on which one** and want to **divide** this dataset into 6 piles, each with photos of one individual

- In **reinforcement learning**, a computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle, robot navigation learning collision avoiding behaviour), without a teacher explicitly telling it whether it has come close to its goal or not. “Discuss **P.A.G.E.** “ feedback from the environment. Simple reward feedback is required for the agent to learn its behaviour; Another example is learning to play a game by playing against an opponent.

Applications for machine learning include:

- Adaptive websites
- Bioinformatics
- Classifying DNA sequences
- Computational finance
- Computer vision, including object recognition
- Detecting credit card fraud
- Game playing
- Internet fraud detection
- Machine perception
- Medical diagnosis
- Natural language processing
- Recommender systems
- Search engines
- Sentiment analysis (or opinion mining)
- Speech and handwriting recognition
- Stock market analysis
- Structural health monitoring

In 2006, the online movie company Netflix held the first "Netflix Prize" competition to find a program to better predict user preferences and improve the accuracy on its existing Cinematch movie recommendation algorithm by at least 10%. A joint team made up of researchers from AT&T Labs-Research in collaboration with the teams Big Chaos and Pragmatic Theory built an ensemble model to win the Grand Prize in 2009 for $1 million. BUT later found out that everything was not recommendation based so they changed their recommendation engine.

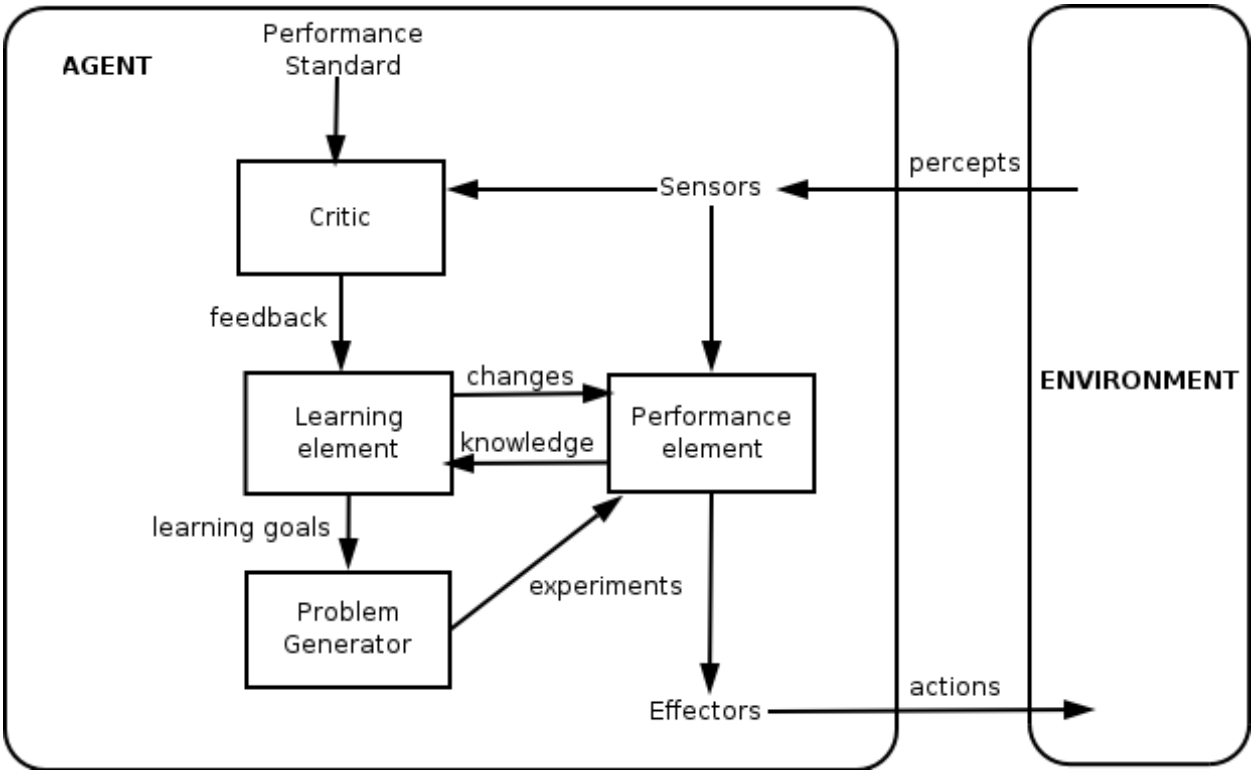


Figure 3 Detailed architecture of Machine learning

Rote learning

Rote learning is a memorization technique based on repetition. The idea is that one will be able to quickly recall the meaning of the material the more one repeats it. Rote methods are routinely used when fast memorization is required, such as learning one's lines in a play or memorizing a telephone number.

Rote learning is widely used in the mastery of foundational knowledge. Examples of school topics where rote learning is frequently used include phonics in reading, the periodic table in chemistry, multiplication tables in mathematics, anatomy in medicine, cases or statutes in law, basic formulae in any science, etc. Rote learning is sometimes disparaged with the derogative terms *parrot fashion*, *regurgitation*, *cramming*, or *mugging* because one who engages in rote learning may give the wrong impression of having understood what they have written or said.

Rote learning is a technique which avoids understanding the inner complexities and inferences of the subject that is being learned and instead focuses on memorizing the material so that it can be recalled by the learner exactly the way it was read or heard. The major practice involved in rote learning techniques is learning by repetition, based on the idea that one will be able to quickly recall the meaning of the material the more it is repeated. Rote learning is used in diverse areas,

from mathematics to music to religion. Although it has been criticized by some schools of thought, rote learning is a necessity in many situations.

What is learning by taking advice?

This is a simple form of learning. Suppose a programmer writes a set of instructions to instruct the computer what to do, the programmer is a teacher and the computer is a student. Once learned (i.e. programmed), the system will be in a position to do new things.

The advice may come from many sources: human experts, internet to name a few. This type of learning requires more inference than rote learning. The knowledge must be transformed into an operational form before stored in the knowledge base. Moreover the reliability of the source of knowledge should be considered.

High level advice needs to be first *operationalized* for a program to use in problem-solving.

There are two basic approaches to advice taking:

- Take high level, abstract advice and convert it into rules that can guide performance elements of the system. *Automate all aspects of advice taking*
- *Develop sophisticated tools* such as knowledge base editors and debugging. These are used to aid an expert to translate his expertise into detailed rules. Here the expert is an *integral* part of the learning system. Such tools are important in *expert systems* area of AI.

Learning by analogy

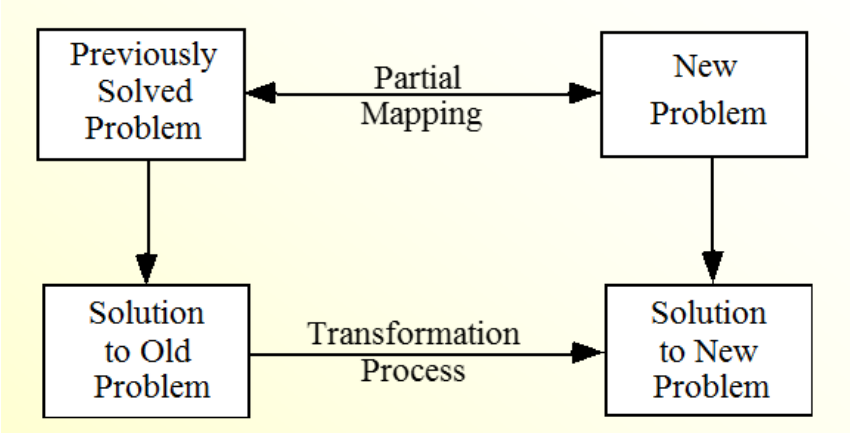
Reasoning by analogy generally involves abstracting details from a particular set of problems and resolving structural similarities between previously distinct problems. Analogical reasoning refers to this process of recognition and then applying the solution from the known problem to the new problem. Such a technique is often identified as case-based reasoning. Analogical learning generally involves developing a set of mappings between features of two instances. Analogical problem solving is mostly described as transfer of a source solution to a target problem based on the structural correspondences (mapping) between source and target. Derivational analogy proposes an alternative view: a target problem is solved by replaying a remembered problem-solving episode. Thus, the experience with the source problem is used to guide the search for the target solution by applying the same solution technique rather than by transferring the complete solution.

Derivational Analogy v/s Transformational Analogy

Detailed history of a problem solving episode is called its derivation. Analogical reasoning that takes these histories into account is called Derivational Analogy. The *history* of the problem solution - the steps involved - are often relevant. It can be used in following scenario:

- In translating Pascal code to LISP -- line by line translation is no use. You will have to reuse the major structural and control decisions.
- One way to do this is to replay a previous derivation and modify it when necessary.
- If initial steps and assumptions are still valid copy them across.
- Otherwise alternatives need to found -- best first search fashion.

Transformational analogy does not look at how the problem was solved -- it only looks at the final solution. Look for a similar solution and *copy* it to the new situation making suitable substitutions where appropriate.



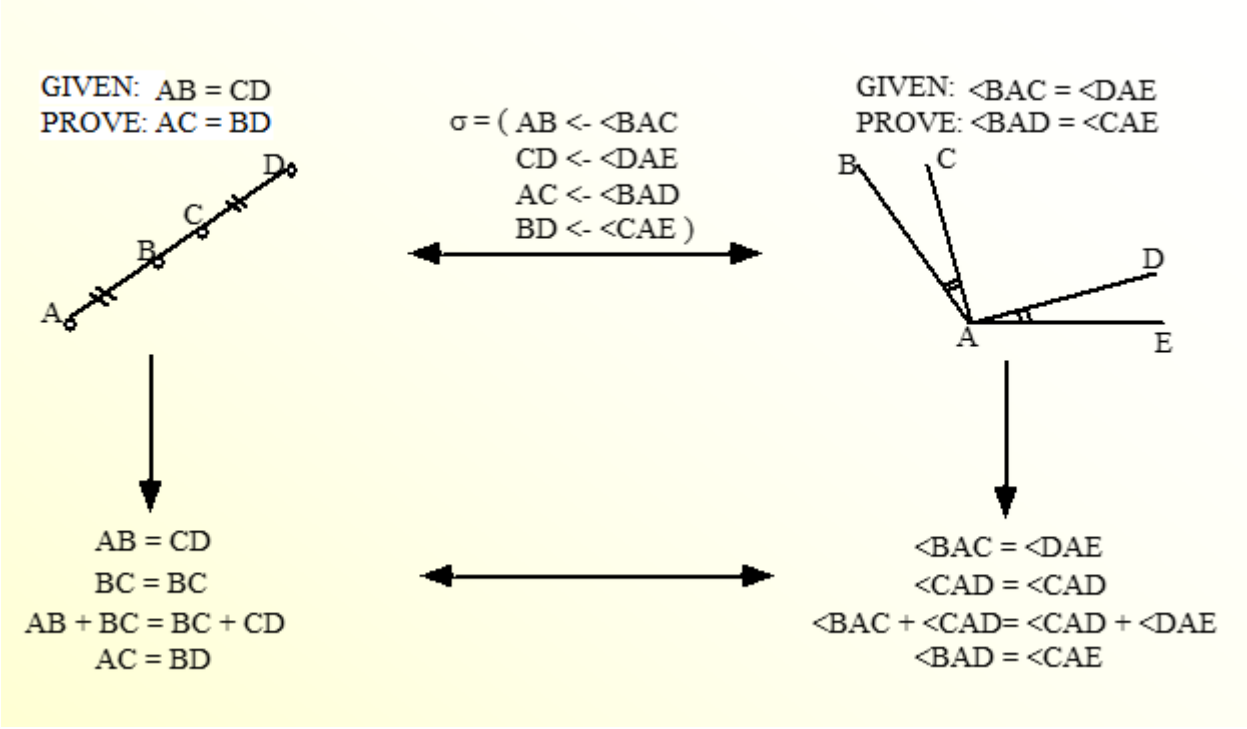


Figure 4 Solving a geometry problem by Transformational Analogy

Explanation based learning

An Explanation-based Learning (EBL) system accepts an example (i.e. a training example) and explains what it learns from the example. The EBL system takes only the relevant aspects of the training. This explanation is translated into particular form that a problem solving program can understand. The explanation is generalized so that it can be used to solve other problems.

This kind of learning occurs when the system finds an explanation of an instance it has seen, and generalizes the explanation. The general rule follows logically from the background knowledge possessed by the system. The entailment constraints for EBL are

Hypothesis ^ Descriptions |= Classifications
Background |= Hypothesis

The agent does not actually learn anything factually new, since the hypothesis was entailed by background knowledge. This kind of learning is regarded as a way to convert first principles into useful specialized knowledge (converting problem-solving search into pattern-matching search).

The basic idea is to construct an explanation of the observed result, and then generalize the explanation. More specifically, while constructing a proof of the solution, a parallel proof is performed, in which each constant of the first is made into a variable. Then a new rule is built in which the left-hand side is the leaves of the proof tree, and the right-hand side is the variabilized goal, up to any bindings that must be made with the generalized proof. Any conditions true regardless of the variables are dropped.

An example of EBL using a perfect domain theory is a program that learns to play chess by being shown examples. A specific chess position that contains an important feature, say, "Forced loss of black queen in two moves," includes many irrelevant features, such as the specific scattering of pawns on the board. EBL can take a single training example and determine what are the relevant features in order to form a generalization.

A domain theory is perfect or complete if it contains, in principle, all information needed to decide any question about the domain. For example, the domain theory for chess is simply the rules of chess. Knowing the rules, in principle it is possible to deduce the best move in any situation. However, actually making such a deduction is impossible in practice due to combinatoric explosion. EBL uses training examples to make searching for deductive consequences of a domain theory efficient in practice.

In essence, an EBL system works by finding a way to deduce each training example from the system's existing database of domain theory. Having a short proof of the training example extends the domain-theory database, enabling the EBL system to find and classify future examples that are similar to the training example very quickly.

EBL software takes four inputs:

- a hypothesis space (the set of all possible conclusions)
- a domain theory (axioms about a domain of interest)
- training examples (specific facts that rule out some possible hypotheses)
- operationality criteria (criteria for determining which features in the domain are efficiently recognizable, e.g. which features are directly detectable using sensors)

Learning from examples: Induction

This involves the process of learning by example -- where a system tries to induce a general rule from a set of observed instances.

This involves classification -- assigning, to a particular input, the name of a class to which it belongs. Classification is important to many problem solving tasks.

A learning system has to be capable of evolving its own class descriptions:

- Initial class definitions may not be adequate.
- The world may not be well understood or rapidly changing.

The task of constructing class definitions is called induction or concept learning. Inductive learning methods can be defined as those methods that systematically produce general descriptions or knowledge from the specific knowledge provided by domain examples.

Winston Learning

- The goal is to construct representation of the definitions of concepts in this domain.
- Concepts such a house - brick (rectangular block) with a wedge (triangular block) suitably placed on top of it, tent - 2 wedges touching side by side, or an arch - two non-touching bricks supporting a third wedge or brick, were learned.
- The idea of *near miss* objects -- similar to actual instances was introduced.
- Input was a line drawing of a blocks world structure.
- Input processed (see VISION Sections later) to produce a semantic net representation of the structural description of the object
- Links in network include *left-of, right-of, does-not-marry, supported-by, has-part,* and *isa*.
- The *marry* relation is important -- two objects with a common touching edge are said to marry. Marrying is assumed unless *does-not-marry* stated.

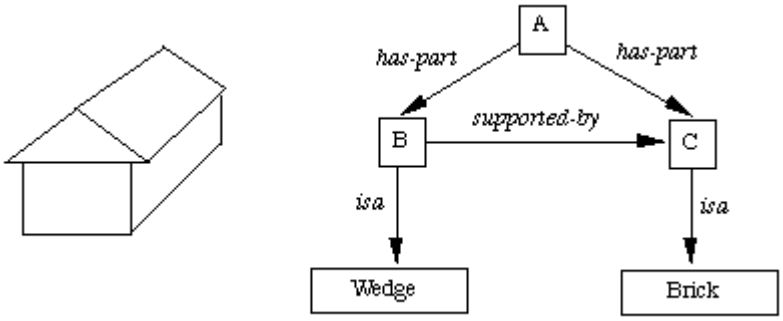


Figure 5 Object house and its Semantic Net/Structural description

There are three basic steps to the problem of concept formulation:

1. Select one know instance of the concept. Call this the *concept definition*.
2. Examine definitions of other known instance of the concept. *Generalize* the definition to include them.

3. Examine descriptions of *near misses*. *Restrict* the definition to *exclude* these.

Both steps 2 and 3 rely on comparison and both similarities and differences need to be identified.

Genetic Algorithm

Genetic Algorithms were invented to mimic some of the processes observed in natural evolution. Many people, biologists included, are astonished that life at the level of complexity that we observe could have evolved in the relatively short time suggested by the fossil record. The idea with GA is to use this power of evolution to solve optimization problems. Genetic Algorithms (GAs) are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimization problems. Although randomized, GAs are by no means random, instead they exploit historical information to direct the search into the region of better performance within the search space. The basic techniques of the GAs are designed to simulate processes in natural systems necessary for evolution, especially those follow the principles first laid down by Charles Darwin of "survival of the fittest.". Since in nature, competition among individuals for scanty resources results in the fittest individuals dominating over the weaker ones.

Genetic Algorithms Overview

GAs simulate the survival of the fittest among individuals over consecutive generation for solving a problem. Each generation consists of a population of character strings that are analogous to the chromosome that we see in our DNA. Each individual represents a point in a search space and a possible solution. The individuals in the population are then made to go through a process of evolution.

GAs are based on an analogy with the genetic structure and behavior of chromosomes within a population of individuals using the following foundations:

Individuals in a population compete for resources and mates.

Those individuals most successful in each 'competition' will produce more offspring than those individuals that perform poorly.

Genes from `good' individuals propagate throughout the population so that two good parents will sometimes produce offspring that are better than either parent.

Thus each successive generation will become more suited to their environment.

Search Space

A population of individuals are is maintained within search space for a GA, each representing a possible solution to a given problem. Each individual is coded as a finite length vector of components, or variables, in terms of some alphabet, usually the binary alphabet {0,1}. To continue the genetic analogy these individuals are likened to chromosomes and the variables are analogous to genes. Thus a chromosome (solution) is composed of several genes (variables).

A fitness score is assigned to each solution representing the abilities of an individual to `compete'. The individual with the optimal (or generally near optimal) fitness score is sought. The GA aims to use selective `breeding' of the solutions to produce `offspring' better than the parents by combining information from the chromosomes.

Based on Natural Selection

After an initial population is randomly generated, the algorithm evolves the through three operators:

1. selection which equates to survival of the fittest;
2. crossover which represents mating between individuals;
3. mutation which introduces random modifications.

1. Selection Operator

Key idea: give preference to better individuals, allowing them to pass on their genes to the next generation. The goodness of each individual depends on its fitness. Fitness may be determined by an objective function or by a subjective judgment.

2. Crossover Operator

Prime distinguished factor of GA from other optimization techniques. Two individuals are chosen from the population using the selection operator. A crossover site along the bit strings is randomly chosen. The values of the two strings are exchanged up to this point. If S1=000000 and s2=111111 and the crossover point is 2 then S1'=110000 and s2'=001111. The two new offspring created from this mating are put into the next generation of the population. By recombining portions of good individuals, this process is likely to create even better individuals.

3. Mutation Operator

With some low probability, a portion of the new individuals will have some of their bits flipped. Its purpose is to maintain diversity within the population and inhibit premature convergence. Mutation alone induces a random walk through the search space. Mutation and selection (without crossover) create a parallel, noise-tolerant, hill-climbing algorithms.

Applications

Genetic algorithms are a very effective way of quickly finding a reasonable solution to a complex problem. Granted they aren't instantaneous, or even close, but they do an excellent job of searching through a large and complex search space. Genetic algorithms are most effective in a search space for which little is known. You may know exactly what you want a solution to do but have no idea how you want it to go about doing it. This is where genetic algorithms thrive. They produce solutions that solve the problem in ways you may never have even considered. Then again, they can also produce solutions that only work within the test environment and flounder once you try to use them in the real world. Put simply: use genetic algorithms for everything you cannot easily do with another algorithm.

Figure 6 The 2006 NASA ST5 spacecraft antenna. This complicated shape was found by an evolutionary computer design program to create the best radiation pattern. It is known as an Evolved antenna.

1. Introduction to neural networks

1.1 What is a Neural Network?

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well.

1.2 Historical background

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least one major setback and several eras.

Many important advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few researchers. The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pits. But the technology available at that time did not allow them to do too much.

1.3 Why use neural networks?

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyze. This expert can then be used to provide projections given new situations of interest and answer "what if" questions. Other advantages include:

Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.

Self-Organization: An ANN can create its own organization or representation of the information it receives during learning time.

Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.

Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

1.4 Neural networks versus conventional computers

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing elements (neurons) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problem by itself, its operation can be unpredictable.

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is to solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high level language program and then into machine code that the computer can understand. These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

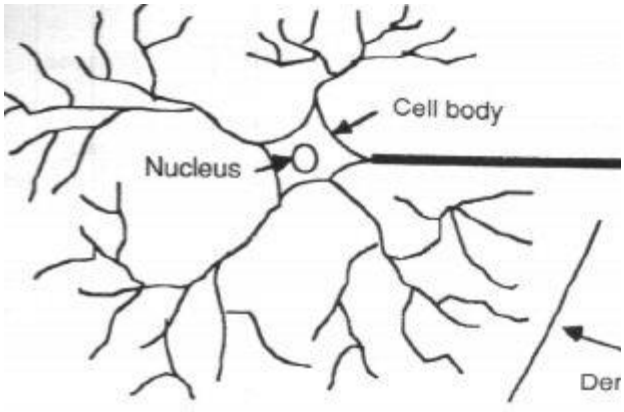
Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks are more suited to an algorithmic approach like arithmetic operations and tasks that are more suited to neural networks. Even more, a large number of tasks, require systems that use a combination of the two approaches (normally a conventional computer is used to supervise the neural network) in order to perform at maximum efficiency.

Neural networks do not perform miracles. But if used sensibly they can produce some amazing results.

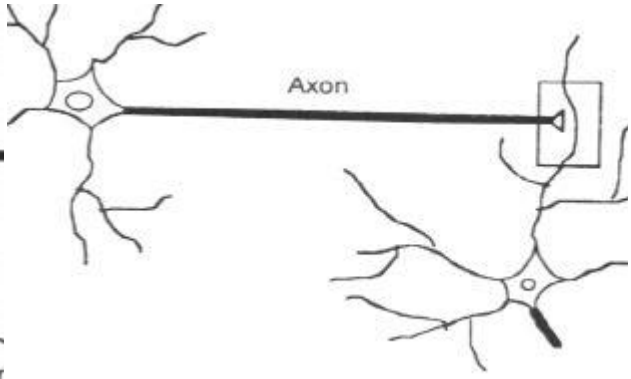
2. Human and Artificial Neurons - investigating the similarities

2.1 How the Human Brain Learns?

Much is still unknown about how the brain trains itself to process information, so theories abound. In the human brain, a typical neuron collects signals from others through a host of fine structures called dendrites. The neuron sends out spikes of electrical activity through a long, thin stand known as an axon, which splits into thousands of branches. At the end of each branch, a structure called a synapse converts the activity from the axon into electrical effects that inhibit or excite activity from the axon into electrical effects that inhibit or excite activity in the connected neurons. When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes.



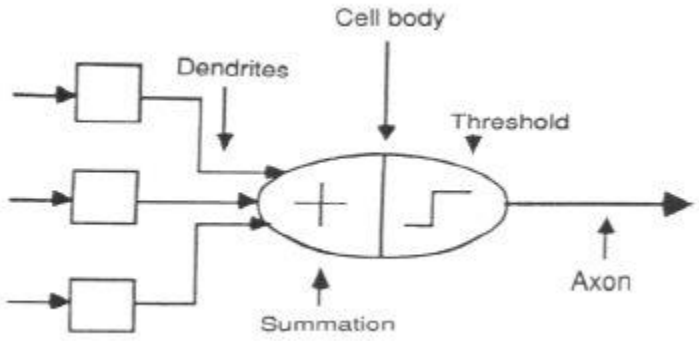
Components of a neuron



The synapse

2.2 From Human Neurons to Artificial Neurons

We conduct these neural networks by first trying to deduce the essential features of neurons and their interconnections. We then typically program a computer to simulate these features. However because our knowledge of neurons is incomplete and our computing power is limited, our models are necessarily gross idealizations of real networks of neurons.

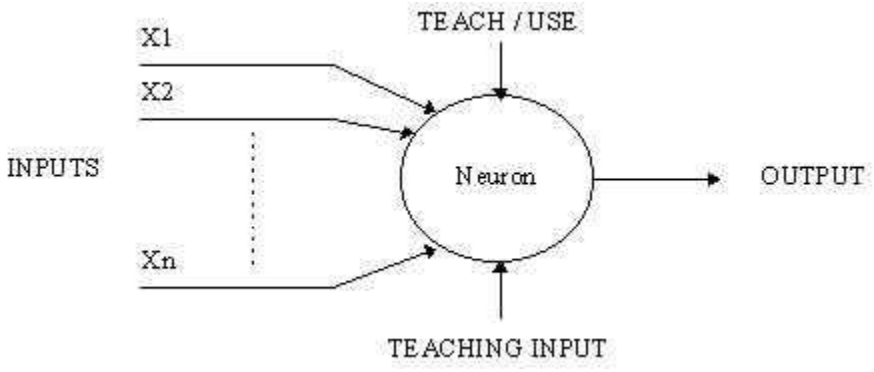


The neuron model

3. An engineering approach

3.1 A simple neuron

An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the training mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire or not.



A simple neuron

3.2 Firing rules

The firing rule is an important concept in neural networks and accounts for their high flexibility. A firing rule determines how one calculates whether a neuron should fire for any input pattern. It relates to all the input patterns, not only the ones on which the node was trained.

A simple firing rule can be implemented by using Hamming distance technique. The rule goes as follows:

Take a collection of training patterns for a node, some of which cause it to fire (the 1-taught set of patterns) and others which prevent it from doing so (the 0-taught set). Then the patterns not in the collection cause the node to fire if, on comparison, they have more input elements in common with the 'nearest' pattern in the 1-taught set than with the 'nearest' pattern in the 0-taught set. If there is a tie, then the pattern remains in the undefined state.

For example, a 3-input neuron is taught to output 1 when the input (X1, X2 and X3) is 111 or 101 and to output 0 when the input is 000 or 001. Then, before applying the firing rule, the truth table is;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0/1	0/1	0/1	1	0/1	1

As an example of the way the firing rule is applied, take the pattern 010. It differs from 000 in 1 element, from 001 in 2 elements, from 101 in 3 elements and from 111 in 2 elements. Therefore, the 'nearest' pattern is 000 which belongs in the 0-taught set. Thus the firing rule requires that the neuron should not fire when the input is 001. On the other hand, 011 is equally distant from two taught patterns that have different outputs and thus the output stays undefined (0/1).

By applying the firing in every column the following truth table is obtained;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0	0/1	0/1	1	1	1

The difference between the two truth tables is called the generalization of the neuron. Therefore the firing rule gives the neuron a sense of similarity and enables it to respond 'sensibly' to patterns not seen during training.

3.3 Pattern Recognition - an example

An important application of neural networks is pattern recognition. Pattern recognition can be implemented by using a feed-forward (figure 1) neural network that has been trained accordingly.

During training, the network is trained to associate outputs with input patterns. When the network is used, it identifies the input pattern and tries to output the associated output pattern. The power of neural networks comes to life when a pattern that has no output associated with it, is given as an input. In this case, the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.

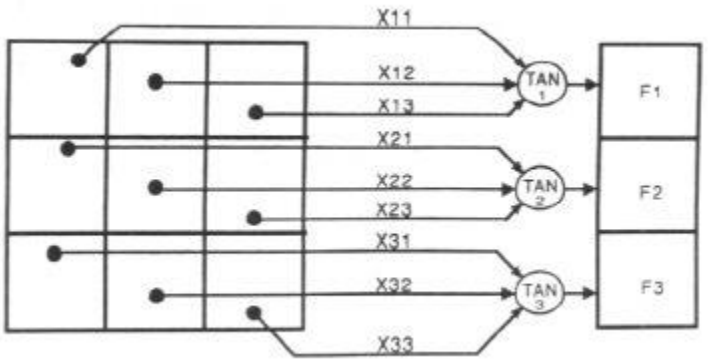


Figure 1.

For example:
The network of figure 1 is trained to recognize the patterns T and H. The associated patterns are all black and all white respectively as shown below.

If we represent black squares with 0 and white squares with 1 then the truth tables for the 3 neurons after generalization are;

X11:		0	0	0	0	1	1	1	1
X12:		0	0	1	1	0	0	1	1
X13:		0	1	0	1	0	1	0	1
OUT:		0	0	1	1	0	0	1	1

Top neuron

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1

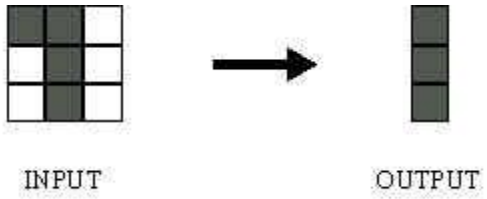
X23:		0	1	0	1	0	1	0	1
OUT:		1	0/1	1	0/1	0/1	0	0/1	0

Middle neuron

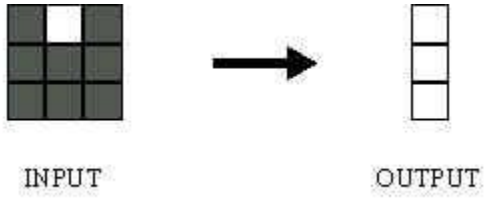
X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0	1	1	0	0	1	0

Bottom neuron

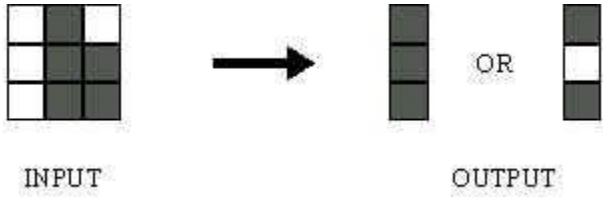
From the tables it can be seen the following associations can be extracted:



In this case, it is obvious that the output should be all blacks since the input pattern is almost the same as the 'T' pattern.



Here also, it is obvious that the output should be all whites since the input pattern is almost the same as the 'H' pattern.



Here, the top row is 2 errors away from the a T and 3 from an H. So the top output is black. The middle row is 1 error away from both T and H so the output is random. The bottom row is 1 error

away from T and 2 away from H. Therefore the output is black. The total output of the network is still in favour of the T shape.

3.4 A more complicated neuron

The previous neuron doesn't do anything that conventional computers don't do already. A more sophisticated neuron (figure 2) is the McCulloch and Pitts model (MCP). The difference from the previous model is that the inputs are 'weighted', the effect that each input has at decision making is dependent on the weight of the particular input. The weight of an input is a number which when multiplied with the input gives the weighted input. These weighted inputs are then added together and if they exceed a pre-set threshold value, the neuron fires. In any other case the neuron does not fire.

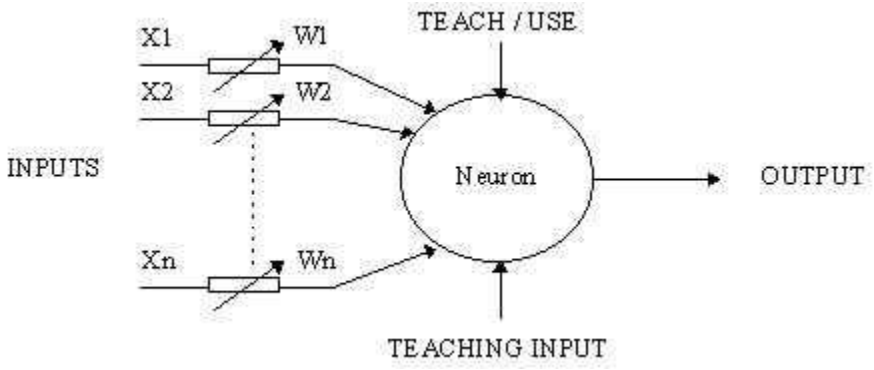


Figure 2. An MCP neuron

In mathematical terms, the neuron fires if and only if;

X1W1 + X2W2 + X3W3 + ... > T

The addition of input weights and of the threshold makes this neuron a very flexible and powerful one. The MCP neuron has the ability to adapt to a particular situation by changing its weights and/or threshold. Various algorithms exist that cause the neuron to 'adapt'; the most used ones are the Delta rule and the back error propagation. The former is used in feed-forward networks and the latter in feedback networks.

4 Architecture of neural networks

4.1 Feed-forward networks

Feed-forward ANNs (figure 1) allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not affect that same layer. Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organization is also referred to as bottom-up or top-down.

4.2 Feedback networks

Feedback networks (figure 1) can have signals travelling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an

equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organizations.

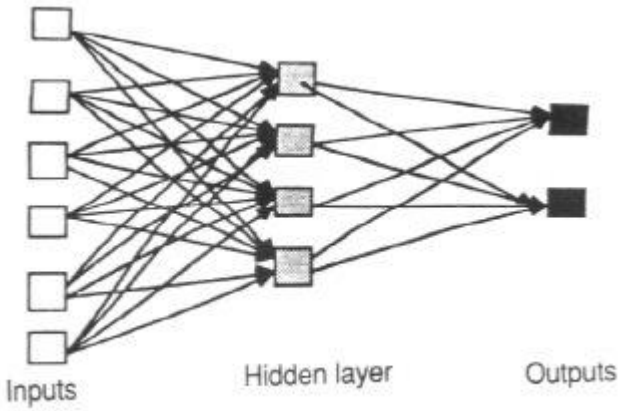


Figure 4.1 An example of a simple feed forward network

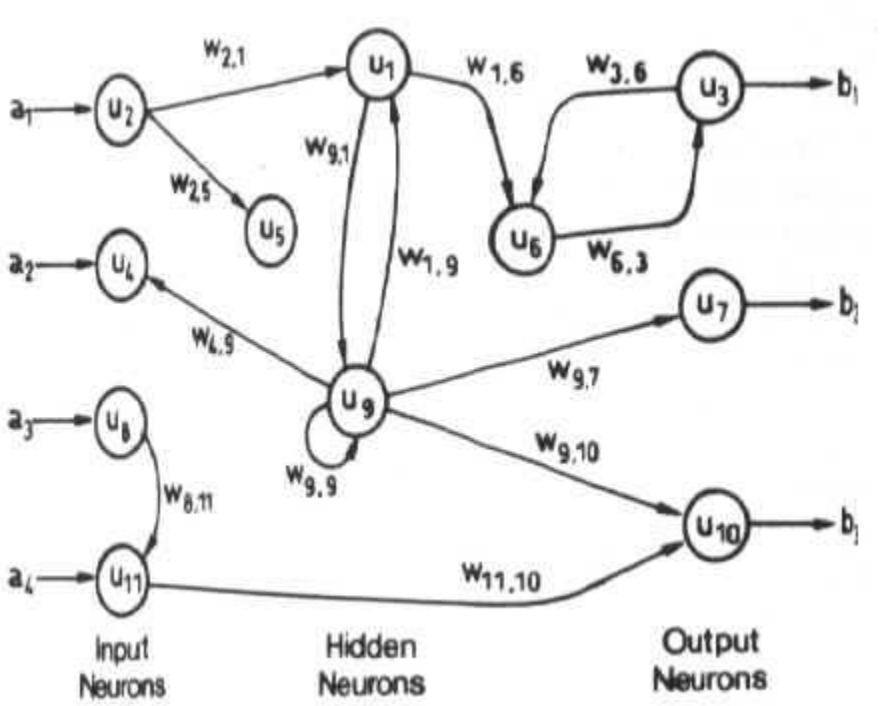


Figure 4.2 An example of a complicated network

4.3 Network layers

The commonest type of artificial neural network consists of three groups, or layers, of units: a layer of "input" units is connected to a layer of "hidden" units, which is connected to a layer of "output" units. (see Figure 4.1)

- The activity of the input units represents the raw information that is fed into the network.

- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
- The behavior of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

We also distinguish single-layer and multi-layer architectures. The single-layer organization, in which all units are connected to one another, constitutes the most general case and is of more potential computational power than hierarchically structured multi-layer organizations. In multi-layer networks, units are often numbered by layer, instead of following a global numbering.

4.4 Perceptrons

The most influential work on neural nets in the 60's went under the heading of 'perceptrons' a term coined by Frank Rosenblatt. The perceptron (figure 4.4) turns out to be an MCP model (neuron with weighted inputs) with some additional, fixed, preprocessing. Units labelled A1, A2, Aj , Ap are called association units and their task is to extract specific, localized featured from the input images. Perceptrons mimic the basic idea behind the mammalian visual system. They were mainly used in pattern recognition even though their capabilities extended a lot more.

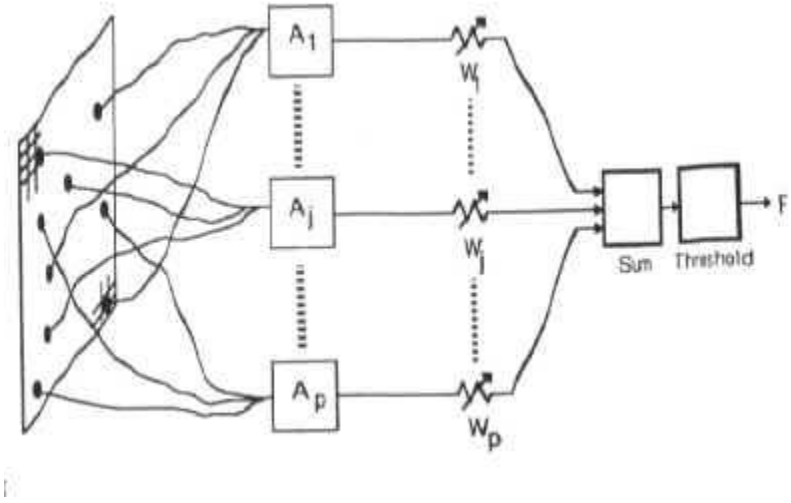


Figure 4.4

5.3 The Back-Propagation Algorithm

In order to train a neural network to perform some task, we must adjust the weights of each unit in such a way that the error between the desired output and the actual output is reduced. This process requires that the neural network compute the error derivative of the weights (EW). In other words, it must calculate how the error changes as each weight is increased or decreased slightly. The back propagation algorithm is the most widely used method for determining the EW.

The back-propagation algorithm is easiest to understand if all the units in the network are linear. The algorithm computes each EW by first computing the EA, the rate at which the error changes as the activity level of a unit is changed. For output units, the EA is simply the difference between the actual and the desired output. To compute the EA for a hidden unit in the layer just before the output layer, we first identify all the weights between that hidden unit and the output units to which it is connected. We then multiply those weights by the EAs of those output units and add the products. This sum equals theEA for the chosen hidden unit. After calculating all the EAs in the hidden layer just before the output layer, we can compute in like fashion the EAs for other layers, moving from layer to layer in a direction opposite to the way activities propagate through the network. This is what gives back propagation its name. Once the EA has been computed for a unit, it is straight forward to compute the EW for each incoming connection of the unit. The EW is the product of the EA and the activity through the incoming connection.

Note that for non-linear units, (see Appendix C) the back-propagation algorithm includes an extra step. Before back-propagating, the EA must be converted into the EI, the rate at which the

6. Applications of neural networks

6.1 Neural Networks in Practice

Given this description of neural networks and how they work, what real world applications are they suited for? Neural networks have broad applicability to real world business problems. In fact, they have already been successfully applied in many industries.

Since neural networks are best at identifying patterns or trends in data, they are well suited for prediction or forecasting needs including:

- sales forecasting
- industrial process control
- customer research
- data validation
- risk management
- target marketing

But to give you some more specific examples; ANN are also used in the following specific paradigms: recognition of speakers in communications; diagnosis of hepatitis; recovery of telecommunications from faulty software; interpretation of multimeaning Chinese words; undersea mine detection; texture analysis; three-dimensional object recognition; hand-written word recognition; and facial recognition.

6.2 Neural networks in medicine

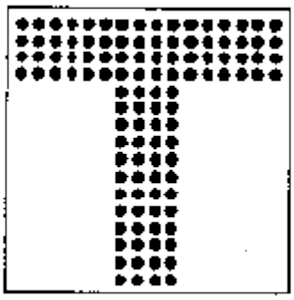
Artificial Neural Networks (ANN) are currently a 'hot' research area in medicine and it is believed that they will receive extensive application to biomedical systems in the next few years. At the moment, the research is mostly on modelling parts of the human body and recognising diseases from various scans (e.g. cardiograms, CAT scans, ultrasonic scans, etc.).

6.3 Neural Networks in business

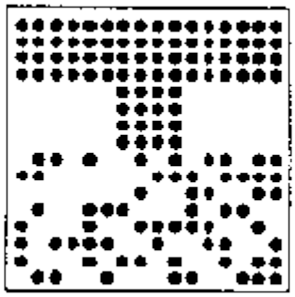
Business is a diverted field with several general areas of specialization such as accounting or financial analysis. Almost any neural network application would fit into one business area or financial analysis. There is some potential for using neural networks for business purposes, including resource allocation and scheduling. There is also a strong potential for using neural networks for database mining, that is, searching for patterns implicit within the explicitly stored information in databases. Most of the funded work in this area is classified as proprietary. Thus, it is not possible to report on the full extent of the work going on. Most work is applying neural networks, such as the Hopfield-Tank network for optimization and scheduling.

Hopfield Network by John Hopfield in 1982

The purpose of a Hopfield net is to store 1 or more patterns and to recall the full patterns based on partial input. For example, consider the problem of optical character recognition. The task is to scan an input text and extract the characters out and put them in a text file in ASCII form. Okay, so what happens if you spilled coffee on the text that you want to scan? Now some of the characters are not quite as well defined, though they're mostly closer to the original characters than any other character:



Original 'T'

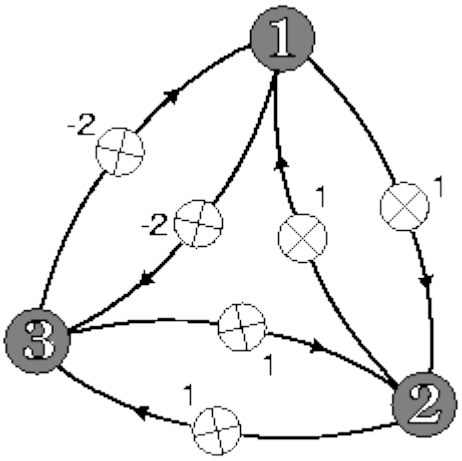


half of image corrupted by noise

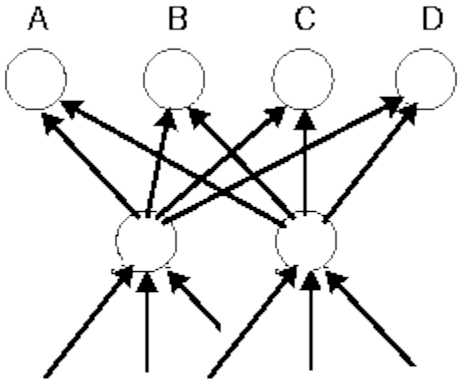
So here's the way a Hopfield network would work. You map it out so that each pixel is one node in the network. You train it (or just assign the weights) to recognize each of the 26 characters of the alphabet, in both upper and lower case (that's 52 patterns). Now if your scan gives you a pattern like something on the right of the above illustration, you input it to the Hopfield network, and it chugs away for a few iterations, and eventually reproduces the pattern on the left, a perfect "T".

Note that this could work with higher-level chunks; for example, it could have an array of pixels to represent the whole word. It could also be used for something more complex like sound or facial images. The problem is, the more complex the things being recalled, the more pixels you need, and as you will see, if you have N pixels, you'll be dealing with N^2 weights, so the problem is very computationally expensive (and thus slow).

All the nodes in a Hopfield network are both inputs and outputs, and they are fully interconnected. That is, each node is an input to every other node in the network. You can think of the links from each node to itself as being a link with a weight of 0. Here's a picture of a 3-node Hopfield network:



3 node Hopfield net



Flow is always in a forward direction

In the previous neural models you've seen, the processing (ignoring training) only goes in one direction: you start from the input nodes, do a sum & threshold of those values to get the outputs of the first layer, and possibly pass those values to a second layer of summing & thresholding, but nothing gets passed back from layer 2 to layer 1 or even passed between the nodes in layer 1:

In a Hopfield network, all the nodes are inputs to each other, and they're also outputs. As I stated above, how it works in computation is that you put a distorted pattern onto the nodes of the network, iterate a bunch of times, and eventually it arrives at one of the patterns we trained it to know and stays there. So, what you need to know to make it work are:

- How to "train" the network
- How to update a node in the network
- How the overall sequencing of node updates is accomplished, and
- How can you tell if you're at one of the trained patterns

Boltzmann Machine

A Boltzmann machine is an extension of a Hopfield network. As the latter, a Boltzmann machine is a set of neurons, which are active or not, and of weighted connections between them. While operating, units are chosen at random and updated. Boltzmann machines can be seen as the stochastic, generative counterpart of Hopfield nets. They were one of the first examples of a neural network capable of learning internal representations, and are able to represent and (given sufficient time) solve difficult combinatoric problems.

A Hopfield network evolves deterministically towards a lower energy state of the system. The final stable state will be a minimum, but not necessarily a globally optimal one. This is because moving from one energy minimum to the other would require increasing the energy of the system. A Boltzmann machine allows for this by a changed mechanism of activating a unit.

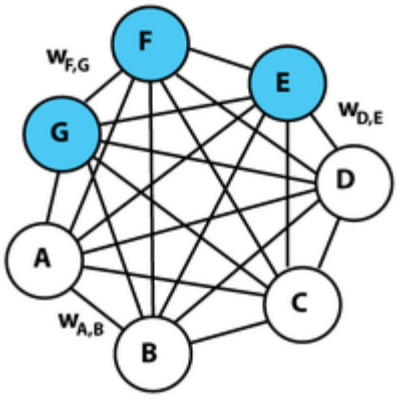


Figure 7 A graphical representation of a Boltzmann machine with a few weights labeled. Each undirected edge represents dependency and is weighted with weight w_{ij}. In this example there are 3 hidden units (blue) and 4 visible units (white).

A Boltzmann machine is a network of symmetrically connected, neuron-like units that make stochastic decisions about whether to be on or off. Boltzmann machines have a simple learning algorithm (Hinton & Sejnowski, 1983) that allows them to discover interesting features that represent complex regularities in the training data. The learning algorithm is very slow in networks with many layers of feature detectors, but it is fast in "restricted Boltzmann machines" that have a single layer of feature detectors. Many hidden layers can be learned efficiently by composing restricted Boltzmann machines, using the feature activations of one as the training data for the next.

Boltzmann machines are used to solve two quite different computational problems. For a search problem, the weights on the connections are fixed and are used to represent a cost function. The stochastic dynamics of a Boltzmann machine then allow it to sample binary state vectors that have low values of the cost function.

For a learning problem, the Boltzmann machine is shown a set of binary data vectors and it must learn to generate these vectors with high probability. To do this, it must find weights on the connections so that, relative to other possible binary vectors, the data vectors have low values of the cost function. To solve a learning problem, Boltzmann machines make many small updates to their weights, and each update requires them to solve many different search problems.

Boltzmann machines can serve both as content-accessible memories (character and face recognition) and a way of solving constraint satisfaction problems (decision making). For example, a constraint satisfaction problem can be solved by introducing a unit for a hypothesis about a variable. Conflicting hypotheses are connected by negatively weighted connections, compatible ones by positive edges. A solution can be found by letting the network operate and consulting the probability distribution of states of the system after reaching thermal equilibrium.

Kohonen Network

The Self-Organizing Map (SOM), commonly also known as Kohonen network (Kohonen 1982, Kohonen 2001) is a computational method for the visualization and analysis of high-dimensional data, especially experimentally acquired information. A self-organizing map (SOM) or self-organizing feature map (SOFM) is a type of artificial neural network (ANN) that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples, called a map. This makes SOMs useful for visualizing low-dimensional views of high-dimensional data, akin to multidimensional scaling.

Kohonen technique creates a network that stores information in such a way that any topological relationships within the training set are maintained. Kohonen's networks are one of basic types of self-organizing neural networks. The ability to self-organize provides new possibilities - adaptation to formerly unknown input data. It seems to be the most natural way of learning, which is used in our brains, where no patterns are defined. Those patterns take shape during the learning process, which is combined with normal work. Kohonen's networks are a synonym of whole group of nets which make use of self-organizing, competitive type learning method. We set up signals on net's inputs and then choose winning neuron, the one which corresponds with input vector in the best way. Precise scheme of rivalry and later modifications of synaptic wages may have various forms. There are many sub-types based on rivalry, which differ themselves by precise self-organizing algorithm.

Functioning of self-organizing neural network is divided into three stages:

- construction
- learning
- identification

The objective of a Kohonen network is to map input vectors (patterns) of arbitrary dimension N onto a discrete map with 1 or 2 dimensions. Patterns close to one another in the input space should be close to one another in the map: they should be topologically ordered. A Kohonen network is composed of a grid of output units and N input units. The input pattern is fed to each output unit. The input lines to each output unit are weighted. These weights are initialized to small random numbers.

Learning in Kohonen Networks

The learning process is as roughly as follows:

- initialise the weights for each output unit
- loop until weight changes are negligible
    - for each input pattern
        - present the input pattern
        - find the winning output unit
        - find all units in the neighbourhood of the winner
        - update the weight vectors for all those units
    - reduce the size of neighbourboods if required

The winning output unit is simply the unit with the weight vector that has the smallest Euclidean distance to the input pattern. The neighbourhood of a unit is defined as all units within some distance of that unit on the map (not in weight space).

A common example used to help teach the principals behind SOMs is the mapping of colours from their three dimensional components - red, green and blue, into two dimensions. Figure 1 shows an example of a SOM trained to recognize the eight different colours shown on the right. The colours have been presented to the network as 3D vectors - one dimension for each of the colour components - and the network has learnt to represent them in the 2D space you can see. Notice that in addition to clustering the colours into distinct regions, regions of similar properties are usually found adjacent to each other. This feature of Kohonen maps is often put to good use as you will discover later.

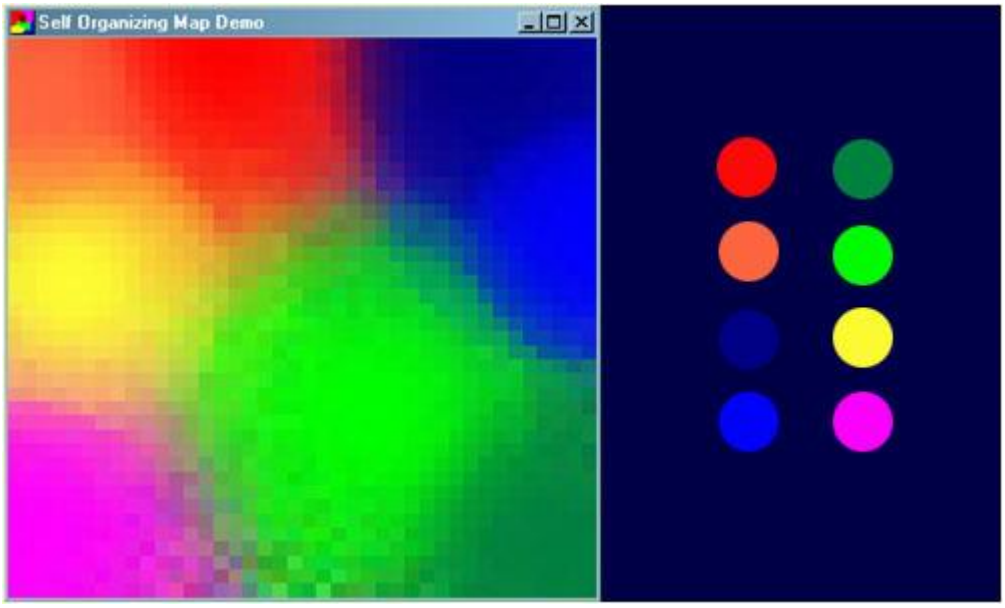


Figure 8 Screenshot of the demo program (left) and the colors it has classified (right).

One of the most interesting aspects of SOMs is that they learn to classify data without supervision. You may already be aware of supervised training techniques such as back propagation where the training data consists of vector pairs - an input vector and a target vector. With this approach an input vector is presented to the network (typically a multilayer feed forward network) and the output is compared with the target vector. If they differ, the weights of the network are altered slightly to reduce the error in the output. This is repeated many times and with many sets of vector pairs until the network gives the desired output. Training a SOM however, requires no target vector. A SOM learns to classify the training data without any external supervision whatsoever.

Expert System

An **expert system** is a computer system that emulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning about knowledge, represented primarily as if–then rules rather than through conventional procedural code. An expert system is divided into two sub-systems: the inference engine and the knowledge base. The knowledge base represents facts and rules. The inference engine applies the rules to the known facts to deduce new facts. Inference engines can also include explanation and debugging capabilities. An ***expert system*** (ES) is a knowledge-based system that employs knowledge about its application domain and uses an inferencing (reason) procedure to solve problems that would

otherwise require human competence or expertise. The power of expert systems stems primarily from the specific knowledge about a narrow domain stored in the expert system's ***knowledge base***. Expert systems do not have human capabilities. They use a knowledge base of a particular domain and bring that knowledge to bear on the facts of the particular situation at hand. The knowledge base of an ES also contains ***heuristic knowledge -*** rules of thumb used by human experts who work in the domain. Application areas include classification, diagnosis, monitoring, process control, design, scheduling and planning, and generation of options.

Classification - identify an object based on stated characteristics

Diagnosis Systems - infer malfunction or disease from observable data

Monitoring - compare data from a continually observed system to prescribe behaviour

Process Control - control a physical process based on monitoring

Design - configure a system according to specifications

Scheduling & Planning - develop or modify a plan of action

Generation of Options - generate alternative solutions to a problem

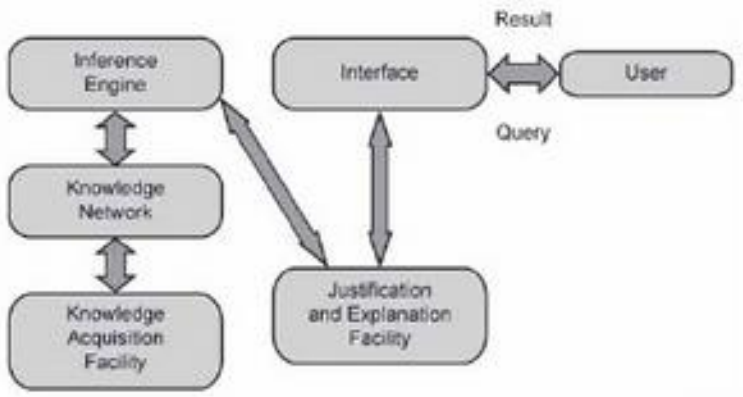


Figure 8.3: Basic components of an Expert System

How Expert Systems Work

The strength of an ES derives from its ***knowledge base*** - an organized collection of facts and heuristics about the system's domain. An ES is built in a process known as ***knowledge engineering***, during which knowledge about the domain is acquired from human experts and other sources by knowledge engineers. The accumulation of knowledge in knowledge bases, from which conclusions are to be drawn by the inference engine, is the hallmark of an expert system. Expert systems offer both tangible and important intangible benefits to owner companies. These benefits should be weighed against the development and exploitation costs of an ES, which are high for large, organizationally important ESs.

Benefits of Expert Systems

An ES is no substitute for a knowledge worker's overall performance of the problem-solving task. But these systems can dramatically reduce the amount of work the individual must do to solve a problem, and they do leave people with the creative and innovative aspects of problem solving.

Some of the possible organizational benefits of expert systems are:

1. An ES can complete its part of the tasks much faster than a human expert.

2. The error rate of successful systems is low, sometimes much lower than the human error rate for the same task.

3. ESs make consistent recommendations

4. ESs are a convenient vehicle for bringing to the point of application difficult-to-use sources of knowledge.

5. ESs can capture the scarce expertise of a uniquely qualified expert.

6. ESs can become a vehicle for building up organizational knowledge, as opposed to the knowledge of individuals in the organization.

7. When use as training vehicles, ESs result in a faster learning curve for novices.

8. The company can operate an ES in environments hazardous for humans.

Limitations of Expert Systems

No technology offers an easy and total solution. Large systems are costly and require significant development time and computer resources. ESs also have their limitations which include:

1. Limitations of the technology

2. Problems with knowledge acquisition

3. Operational domains as the principal area of ES application

4. Maintaining human expertise in organizations

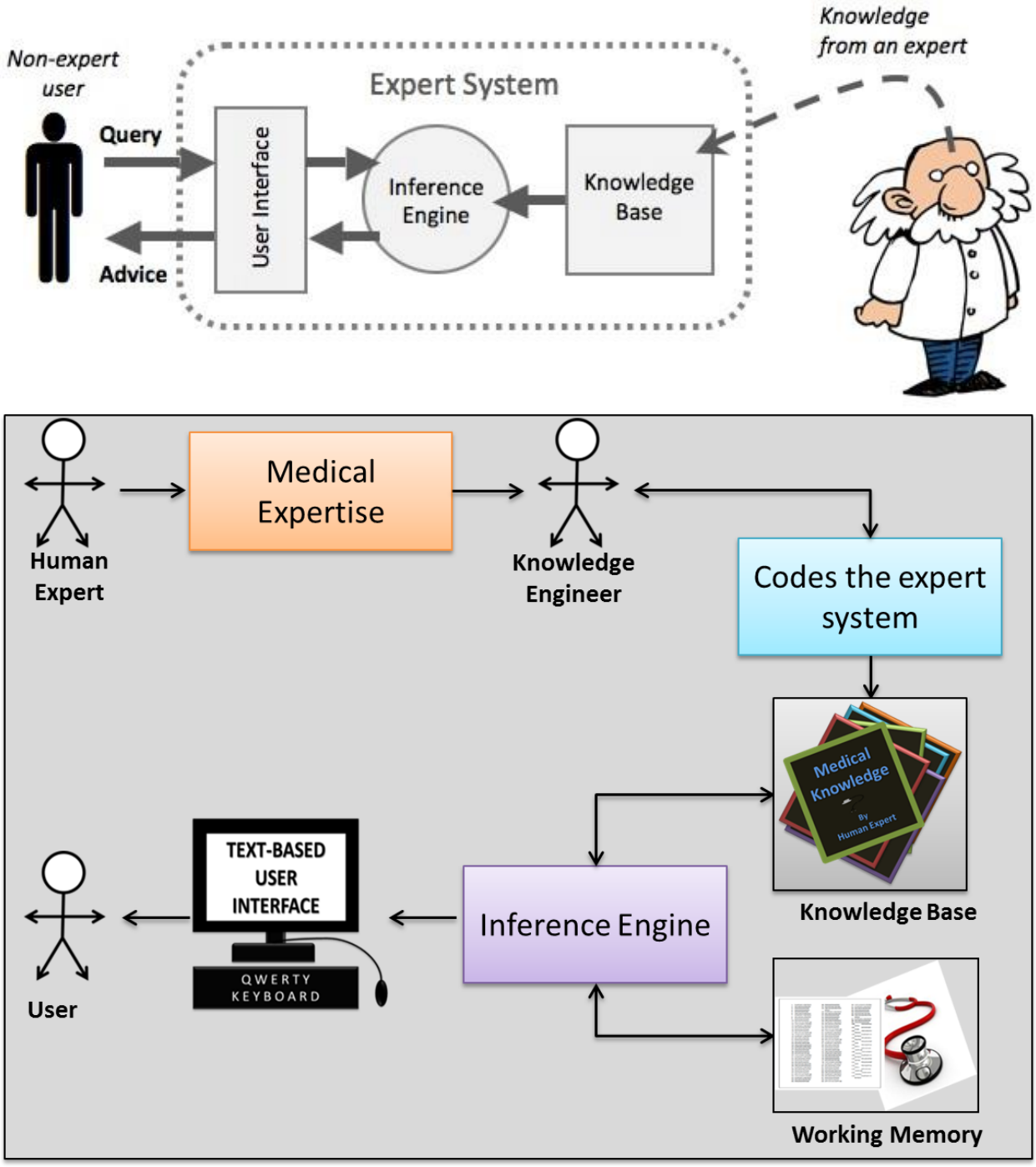


Figure 9 Medical Expert System

Natural Language Processing

Being able to talk to computers in conversational human languages and have them understand us is a goal of AI researchers. Natural language processing systems are becoming common. The main application for natural language systems at this time is as a user interface for expert and

database systems. Natural language processing (NLP) is a field of computer science, artificial intelligence, and computational linguistics concerned with the interactions between computers and human (natural) languages. As such, NLP is related to the area of human–computer interaction. Many challenges in NLP involve natural language understanding, that is, enabling computers to derive meaning from human or natural language input, and others involve natural language generation.

Figure 10 Steps of NLP

Different level of analysis required:

Morphological analysis - concerns how words are constructed from more basic meaning units called morphemes. A morpheme is the primitive unit of meaning in a language. Individual words are analyzed into their components and non-word tokens, such as punctuation are separated from the words.

Syntactic analysis - concerns how words can be put together to form correct sentences and determines what structural role each word plays in the sentence and what phrases are subparts of other phrases. Linear sequences of words are transformed into structures that show how the words relate each other.

Semantic analysis- concerns what words mean and how these meaning combine in sentences to form sentence meaning. The study of context-independent meaning.

Discourse analysis- concerns how the immediately preceding sentences affect the interpretation of the next sentence. For example, interpreting pronouns and interpreting the temporal aspects of the information.

Pragmatic analysis- concerns how sentences are used in different situations and how use affects the interpretation of the sentence.

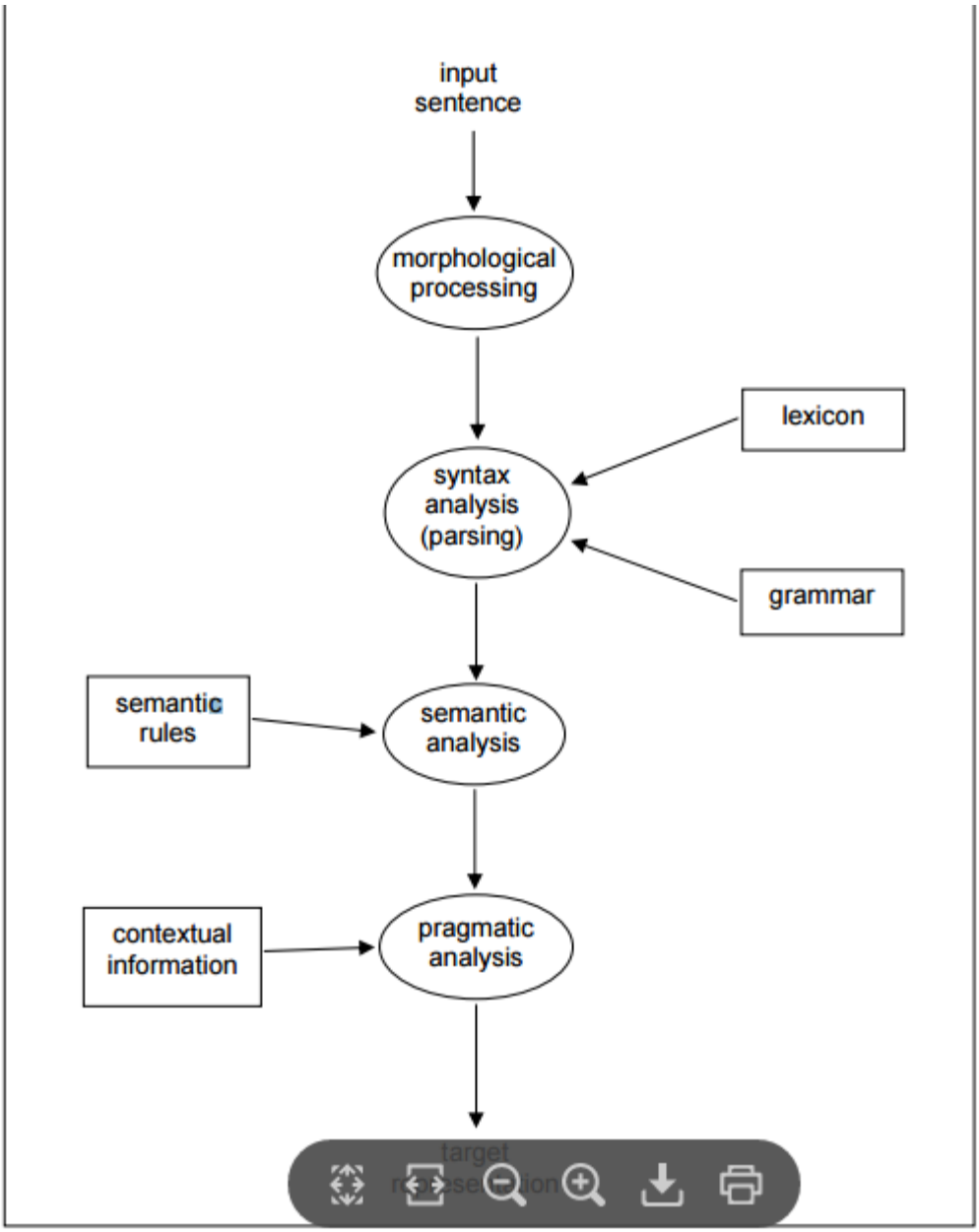


Fig. 2.1. The logical steps in Natural language Processing

Morphological Analysis

Morphological analysis gives a basic insight into natural language by studying how to distinguish and generate grammatical forms of words arising through inflection. This involves considering a set of tags describing grammatical categories of the word form concerned, most notably, its base form and paradigm. Automatic analysis of word forms in free text can be used for instance in grammar checker development, and can aid corpus tagging, or semi-automatic dictionary compiling.

Syntactic Analysis

The goal of syntactic analysis is to determine whether the text string on input is a sentence in the given (natural) language. If it is, the result of the analysis contains a description of the syntactic structure of the sentence, for example in the form of a derivation tree. Such formalizations are aimed at making computers "understand" relationships between words (and indirectly between corresponding people, things, and actions). Syntactic analysis can be utilized for instance when developing a punctuation corrector, dialogue systems with a natural language interface, or as a building block in a machine translation system.

Examples of a simple lexicon and grammar could be:

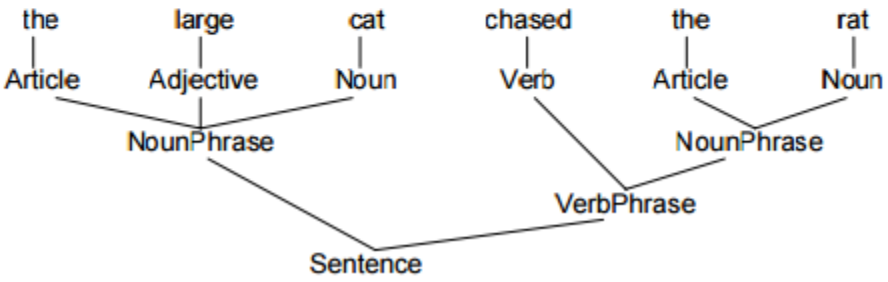
Lexicon

word	category
cat	Noun
chased	Verb
large	Adjective
rat	Noun
the	Article

Grammar

Sentence → NounPhrase, VerbPhrase[2]
VerbPhrase → Verb, NounPhrase
NounPhrase → Article, Noun
NounPhrase → Article, Adjective, Noun

This grammar-lexicon combination could destructure the sentence "*The large cat chased the rat*" as follows:



Semantic Analysis

Semantic and pragmatic analysis make up the most complex phase of language processing as they build up on results of all the above mentioned disciplines. Based on the knowledge about the structure of words and sentences, the meaning of words, phrases, sentences and texts is stipulated, and subsequently also their purpose and consequences. From the computational point of view, no general solutions that would be adequate have been proposed for this area. There are many open theoretical problems, and in practice, great problems are caused by errors on lower processing levels.

Discourse Analysis

The execution of current sentence may depend on the previous sentence and may influence the coming sentence. Thus such words should be analyzed to give out the useful meaning in later part. Like the use of pronoun in second sentence that refers to the noun from first sentence.

Pragmatic Analysis

Pragmatics is the study of language in use. Studying language in use entails an ability to match the formal aspect of language with the appropriate context, that is, utterances with their situations. To deal with language from this perspective, is to account for the relationship holding between language forms and language users.

Applications:

Robotics

AI, engineering, and physiology are the basic disciplines of robotics. This technology produces robot machines with computer intelligence and computer-controlled, human like physical capabilities, robotics applications

Computer Vision

The simulation of human senses is a principal objective of the AI field. The most advanced AI sensory system is compute vision, or visual scene recognition. The task of a vision system is to interpret the picture obtained. These systems are employed in robots or in satellite systems. Simpler vision systems are used for quality control in manufacturing.

Speech Recognition

The ultimate goal of the corresponding AI area is computerized speech recognition, or the understanding of connected speech by an unknown speaker, as opposed to systems that recognize words or short phrases spoken one at a time or systems are trained by a specific speaker before use.

For Genetic algorithm Youtube videos, search for:

EVOLUTION [V6] - Learn to Walk (genetic algorithm

Genetic Algorithm - explained in 4 minutes

Genetic Algorithms

Genetic evolution of a wheeled vehicle with Box2d

Introduction to Genetic Algorithms